NASA Technical Memorandum 104807

Making Intelligent Systems Team Players

A Guide to Developing Intelligent Monitoring Systems

Sherry A. Land
Jane T. Malin
Carroll Thronesberry
Debra L. Schreckenghost

July 1995

NASA Technical Memorandum 104807

Making Intelligent Systems Team Players

A Guide to Developing Intelligent Monitoring Systems

Sherry A. Land
Jane T. Malin
Lyndon B. Johnson Space Center
Houston, Texas

Carroll Thronesberry
Debra L. Schreckenghost
Metrica, Inc.
Houston, Texas

July 1995

Contents

Tables

Figures

Introduction

Purpose

The purpose of this Developers' Guide is to assist developers of intelligent systems who work in the space industry and related industries in their development of complete and reliable monitoring systems. In preparing this guide, we have drawn from the experiences of 5 years of work in the evolution of the DEcision Support SYstem (DESSY), a real-time application that supports monitoring and fault detection. This guide documents our strategies and lessons learned so that other developers of other systems can work more productively.

This guide is not intended to provide theoretical methodology discussions of various development issues or to act as a complete documentary on the development process. It is, rather, intended to be a source of hands-on experiences from which other developers can draw to gain a better understanding of the problems they face. A cookbook approach is used, with step-by-step instructions and examples clarifying the stages of developing the system. Because DESSY was developed using the G2 programming tool, the examples provided are in G2. However, we discuss all examples in a generic way to allow developers using other development tools to find the guide useful.

The true value of this guide lies in examples provided from a real case study (DESSY), which developers can tailor to their own projects. Our goal is to eliminate time wasted in reinventing the wheel, and to make the development process as smooth and efficient as possible.

Scope

This guide is based solely on experiences in developing the DESSY expert system in the G2 programming environment offered by Gensym Corporation. To understand its scope, the reader must understand the scope of DESSY itself, as well as have some understanding of G2 or a G2-like tool. DESSY monitors Space Shuttle telemetry data in real time and uses the real-time capabilities offered by the G2 package. DESSY also takes advantage of the object-oriented capabilities provided by the G2 software. In reading the guide, it will be helpful to be familiar with G2, but it is not required. The guide is written with assumption that the user has a familiarity with object-oriented programming and rule-based systems. Throughout this guide, the software system will be referred to as intelligent system or expert system or knowledge base interchangeably. Although much of the discussion focuses on rule-based systems, usually described by the term expert system, many of the principles could be applied to any type of monitoring intelligent system.

It is also important to understand the boundaries of DESSY's monitoring capabilities. DESSY monitors telemetry data and makes inferences about commands, state transitions, and simple failures. It is not a failure analysis system capable of performing in-depth failure diagnostics. It is the authors' opinion that real-time monitoring and failure analysis are separate functions that should remain separate. DESSY does, however, identify simple failures detected from telemetry data. In short, DESSY performs failure detection, rather than failure diagnostics.

The key phrases that summarize the characteristics of DESSY are expert system, real-time, data monitoring, rule-based, object oriented, and failure detection.

Expected Use

This guide should be used by expert system developers who want to expedite the development process. Although an initial scanning of the entire guide is recommended, it is primarily intended to be used throughout the development process as a reference guide. We have outlined the steps in building DESSY that worked well for that particular project, and we feel the sequence is general enough to be applied to other expert systems development tasks. It may be that the organization of this manual does not suit your specific development needs. In that case, we ask that you remain open to the specific task examples found throughout the sections. The guide can be used in sequence or in a purely reference manner. Although we recommend proceeding with your development in the sequenced steps, the examples that are provided will usually stand alone and should be helpful at any point in your development.

Organization

The organization of this guide has been set up to reflect what the authors believe to be the organization of the complete development process. The guide begins with a philosophical discussion of preliminary issues related to development and various development approaches. The first section, entitled "Real Time Monitoring Systems," should be read first to enable the reader to understand the attitudes and orientations of the authors.

The remaining sections deal with the process of producing the expert system. Section 2, "Determining the System Requirements," deals with initial project tasks such as the development of the project planas well as the process of understanding the existing tasks of the people involved with the system to be supported, i.e., the expert systems domain. Also involved in these preliminary activities is the definition of the project scope and the identification of requirements.

Once the developer has the preliminary tasks of section 2 underway, he can move on to section 3, "Building the System." In this section, design and development phases have been merged. This approach was used primarily because of the nature of the development tool. The G2 tool and others that have a graphical development environment support the combination of design and development rather well because design requirements can be quickly captured within the tool. Once the design is encoded, the development process is greatly simplified. In fact, at times encoding the design actually accomplishes parts of the development as well. In any case, these concepts have been combined as a single (but iterative) phase and are presented in section 3.

Another area of great concern in developing intelligent monitoring systems is understanding the data source. Section 4, "Working with Real-Time Data," addresses issues software developers face when tying monitoring systems to real data sources. These issues center on noisy and unreliable real-time data. The problems are discussed in this section, along with a methodology used to overcome them.

Section 5, "Evolving the System through the User Interface," relates the close tie of operational prototyping and software evolution with the user interface of the intelligent system. The user interface is the window into the intelligent system during design and development as well as during use. This section provides insight in using the intelligent system interface, beginning with a storyboard, to guide software evolution to its final state.

Section 6, "Testing the System," covers a crucial phase in the life of the project. Although testing is among the iterative steps of design and development, we chose to treat it separately because it differs somewhat from the other steps in that the end users (flight controllers in the case of DESSY) must be highly involved in the testing phase. This contrasts with the design and development phase, which is more a programmer's task. (although the end user certainly must be involved in all project phases.) Section 6 discusses verification and validation, and covers various types of testing that are necessary throughout the system development.

Documentation and training are also important parts of an intelligent system project, particularly since the system will almost certainly evolve as new information is learned or as the physical system being monitored changes. Thus we have included Section 7, "Documentation and User Training," which discusses the types of documentation and training we felt were necessary during DESSY development.

Finally several appendices are included which contain specific DESSY documentation. Appendix A, "DESSY End Effector Failures," includes listing of rules from DESSY, and Appendix B, "DESSY Cue Cards," contains the cue cards from both DESSY subsystems. These appendices are intended to provide software developers with further detailed information about DESSY.

Additional Information

This Developer's Guide is actually only part of a larger body of information designed to assist intelligent systems developers. Also available are an electronic library, software demonstrations, and additional documentation. The Control Center Library For Application Reuse and Exchange (CLARE) is an electronic library containing sample applications or their documentation and on-line literature, including this Developer's Guide, to support control center software developers. CLARE has been developed in the hypertext mark-up language (html) for Mosaic browsing. The goal of CLARE is to provide interactive, on-line support for requirements definition, concurrent development and improvement of advanced software. CLARE is available at the following URL: http://tommy.jsc.nasa.gov/~clare.

Demonstrations of DESSY and other intelligent systems software are also available currently through the Intelligent Systems Branch at Johnson Space Center, and eventually will be through CLARE. For information about obtaining a demonstration or for further information about CLARE or other related software, contact the authors of this document.

Finally, additional literature is available to support software developers. The "Making Intelligent Systems Team Players" (Malin and Schreckenghost, 1991) document set provides case studies and software design information related to DESSY. This document is available electronically through CLARE.

Section 1
Real-Time Monitoring Systems

Because the definition of real-time monitoring systems may vary from source to source, the definition as applied to the DESSY project is provided. DESSY monitors a set of Shuttle telemetry data which has an update rate of once per second. Any reasoning done in DESSY involving time uses the second as the unit. Because G2, and therefore DESSY, runs on the UNIX operating system, which is not a true real-time operating system, real-time processing is limited. Thus for DESSY, real-time monitoring means monitoring data which is periodically updated once a second, processing that data in the allotted time, and reasoning over that data in that same second. It is recommended that the reader gain a thorough understanding of real-time processing; however, that discussion is beyond the scope of this document.

The following sections cover topics germane to understanding real-time monitoring systems. Section 1.1, "Types of Monitoring Systems," addresses different system classifications. Section 1.2, "Scope of Monitoring Systems," covers trade-offs associated with limiting an expert system's scope. Finally, "Supported Development Approach" (section 1.3) explains the operational prototyping approach used throughout this guide.

1.1 Types of Monitoring Systems

Monitoring systems can be broken into categories by their logical structures and the classification of their domains. Logically, a system may contain rules, procedures, or both, depending upon the nature of the monitored activities. Likewise, the system may be used for monitoring passive or inactive systems in which the goal is to observe fairly static sensor readings and flag anomalies, or the software may monitor very active systems in which changes are expected and normal. Challenges of working with real-time data apply in any case.

1.1.1 Rule-based vs. Procedural Systems

DESSY is primarily a rule-based system, but does contain procedures. Although the focus of this guide is on rule-based systems, there are similarities in the ways in which rules and procedures are used, and there are many systems, like DESSY, that use a mix of rules and procedures.

When should rules be chosen over procedures and vice versa? Procedures are lines of code that are sequentially executed. When activities take place that are known to be sequential, procedures are the more appropriate model. It is often the case in monitoring data and anomalies, however, that there are many paths that may occur in a given set of data over a specified time period or activity sequence. In these situations, rules are the more appropriate model. It will be up to the developer to use the appropriate tool for a particular scenario. In any case, both techniques are available and can be used separately or together. (Bachant and Soloway, 1989)

This guide focuses primarily on developing rules since DESSY contains mostly rules. However many of the steps presented apply to writing either rules or procedures, and many of the issues dealing with handling real-time data may be applicable to both rules and procedures alike.

1.1.2 Passive vs. Active Systems

Another distinction between types of monitoring expert systems is whether a system is passive or active. If a system monitors operations in which little or no change in the data is expected, the system is passive. The goal of a passive system is to remain in its current state. Any change from this state indicates an error. The original Bus Loss Smart System (BLSS) is an example of an expert system that is normally passive. This G2 system monitors Shuttle power buses, which normally remain in a static configuration.

In contrast, an active system expects frequent data changes, and therefore configuration changes, as a part of normal activities. These expected changes cause monitoring to be more complex, due in part to the multiple configurations that are expected to occur. Additionally, it can be challenging to monitor a state transition in which multiple data values change simultaneously or over a period of time. (Gabrielian and Franklin, 1988)

DESSY is an example of an active system, although DESSY operates passively for long periods of time. Most systems have both passive and active stages during the monitoring process. The software developer should become aware of all system configurations and transitions when planning the expert system. Only when these have been identified and thoroughly understood can development begin.

1.2 Scope of Monitoring Systems

Another factor in planning a real-time monitoring system is the scope of the system. Real-time systems have the specific job of keeping up with a data stream in real time. Any further analysis of the data uses valuable CPU resources. Thus it is important to realize that there is a significant trade-off between strict data monitoring and data analysis. This trade-off is due to limited performance and will always exist given a real-world situation. Although by necessity any monitoring system performs some analysis, the authors believe it is necessary to limit the analysis functionality of a real-time monitoring system. In an ideal scenario, a monitoring system and an analysis (or diagnostic) system are working side by side. The monitoring system watches the data, makes basic state and status conclusions, and then feeds those conclusions into the diagnostic system. The diagnostic system, which might serve as a database or also include recommended actions, is run off-line, i.e., not real-time. This is necessary because this type of system will have many information sources, including human inputs and other data parameters, that are not part of the telemetry stream. Figure 1 depicts the scenario the authors feel will most adequately meet the needs of the complete monitoring and diagnostic system.

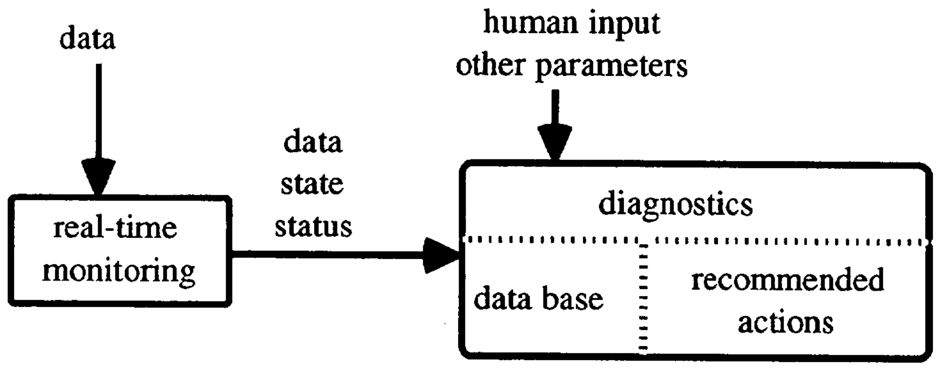


Figure 1. Complete monitoring and diagnostic system.

1.3 Supported Development Approach

This developer's guide supports operational prototyping, a software development approach that originated from observations of successful intelligent system projects at Johnson Space Center, although similar approaches have been employed by other organizations. (Jordan, et. al., 1989) The approach is called operational because the mature products are capable of functioning in an operational environment—receiving data from operational sources and providing conclusions in a timely fashion in conjunction with human operators. The approach is called prototyping because it is informal and iterative. The key features of operational prototyping are participatory design, iterative development, and refinement through team interaction with the prototype.

1.3.1 Participatory Design

Participatory design refers to the practice of including users on the software development team. Users have an intimate understanding of the human task being supported by the software system. While systems analysts and programmers could usually learn to perform the users' tasks, the training time would often be considerable and their competency of task performance would still be lower than that of a highly experienced user. Consequently, including a user on the design team is usually more efficient and allows a more thorough consideration of the supported human task. These advantages become stronger as the human task becomes more complex.

For example, DESSY is an intelligent system designed to support monitoring of the Space Shuttle payload bay arm by ground-based flight controllers. The monitoring task is difficult, complex, and dependent on extensive engineering knowledge of the Shuttle arm. By including a flight controller, the development team avoids the need to perform an exhaustive and expensive task analysis. The risks of overlooking some of the users' needs, or of making the system unusable within the context of the users' other ongoing tasks, are also avoided.

The value of participatory design can be summarized by the following statement taken from Space Operations Seminar held October 20, 1992, at the University of Houston at Clear Lake. "The users know what they don't want, but not necessarily what they do want." Having a user play an active role in development greatly increases the chances of producing a system that the user both wants and can use.

1.3.2 Iterative Development

Iterative development is an approach whereby the software system is developed as a series of successive approximations to the final product rather than as a single, monolithic, integrated system. Iterative development is especially good for projects with unstable requirements. Examples are software systems that provide innovative support of user tasks and those that include innovative applications of new technology. Because these projects are innovative, developers are likely to make discoveries of new requirements after they have begun the project. If they have adopted an iterative approach, it will be easier to respond to those discoveries by revising the current system and revising plans for future enhancements. If, on the other hand, they have adopted a monolithic approach, these discoveries will probably come too late to be of any value to the current system.

For example, if the first iteration of development involves the core of the basic architecture, the developers can verify whether that architecture has the desired properties before designing the remainder of the system in detail. If the architecture proves to have unanticipated problems, it is much easier to change the architecture and redirect the remainder of the project with a minimal cost in development resources. Because developers have the least amount of experience with innovative projects, they are more likely to need redirecting, thereby making iterative development especially valuable in these cases.

1.3.3 Refinement Through Team Interaction with Prototype

The third key feature of operational prototyping is refinement through team interaction with the prototype. Because the team interacts with a prototype, each person is able to see the implications of design decisions. What You See Is What You Get, or WYSIWYG, is a descriptor for applications that immediately show the full implications of user input. Prototyping is the WYSIWYG of analysis and design. Also, because all team members interact with the prototype, several viewpoints can be considered simultaneously. If the team has expertise in intelligent systems development, Space Shuttle subsystem engineering, user task performance, and human factors engineering, then the implications of a proposed change can be evaluated from each of those perspectives. Because these viewpoints can be considered simultaneously, the team is able to make dynamic trade-offs. For example, if a proposed change seems ideal except for implementation feasibility, the intelligent system developer can voice an objection along with the reason behind it. At that point, a brainstorming session can ensue in which potential alternatives are proposed by all participants. In this situation, with all experts present, a solution that satisfies all participants is much more likely to be found. This is the impetus behind concurrent engineering in systems development.

1.3.4 Evolution of the System (Adaptation of Boehm's Spiral)

A common problem with rapid prototyping is a type of wandering project development, in which the project has no defined goals and tends to follow the interests of the developer. The objective is to balance goal-directed development with the ability to respond to unexpected discoveries during development. An adaptation of Boehm's spiral model is the best way to achieve that balance. This adaptation is illustrated in figure 2. As a spiral, this approach has two types of components, a cyclic component and an outward progression component.

The component illustrated in panel A of figure 2 is the cyclic component. Starting with the upper left quadrant, each development iteration begins with a consideration of objectives, alternatives, and constraints. This is followed by a risk analysis in which alternatives are evaluated and a strategy for resolving those risks is formulated. The third step is to plan the next prototype iteration. The final step is to develop and evaluate the next prototype iteration. Each prototype iteration is expected to be an enhancement of the previous iteration.

The second component of the spiral is an outward progression, across cycles. It is illustrated in panel B of figure 2. According to Boehm's spiral model, the issues posing the greatest risk to the project are to be addressed first. If there is an aspect of that application that cannot be accomplished as planned, it is important to discover that fact early. If the project must be scrapped, it can be scrapped before many resources have been expended. If the application must be redesigned as the result of unanticipated discoveries, it is better to make those discoveries before the remainder of the application has been built. As a rule of thumb, the risky aspects of the project are addressed early, reserving the better understood portions of the project for subsequent development iterations.

In the DESSY project, Boehm's spiral has been adapted for rule-based systems to support real-time monitoring of space systems by flight controllers via telemetry downlinks. The sequencing of system development is based on a combination of perceived risk and a need to develop a useful software architecture and object structure. The progression of development for DESSY modules is presented as an example. These recommendations are not presented so that developers will follow them without deviation. Instead, active consideration is recommended based on the specific risks of the project at hand—and active reconsideration as the project continues. The reasoning behind recommendations is shown as an example of how to make the sequencing decisions rather than as a justification for sequencing all projects in exactly this manner.

The objectives addressed by successive DESSY iterations are:

1. Displaying relevant telemetry values in a readily understood fashion

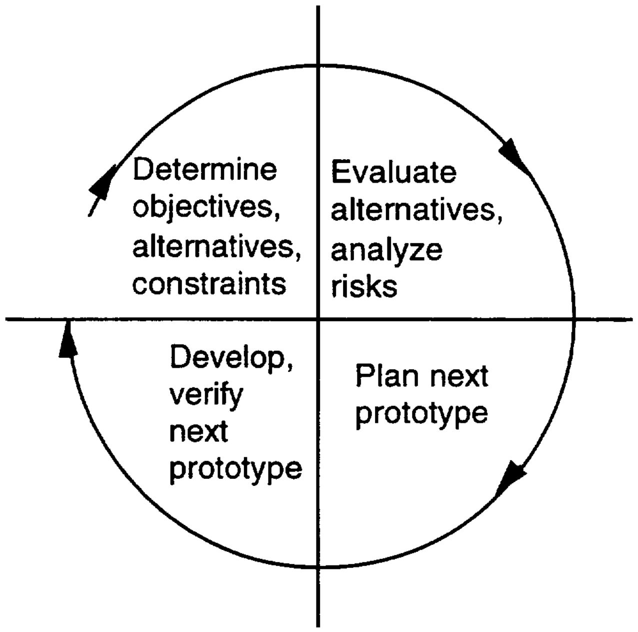
2. Inferring (from telemetry data) and displaying the current state (usually the physical position) of the subsystem being monitored

3. Making the rules resilient to noisy and occasionally missing telemetry data values

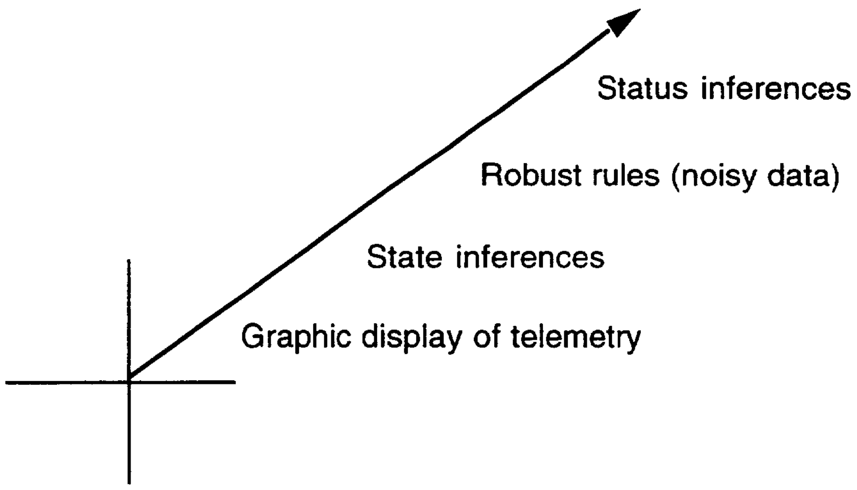
4. Making inferences about the status (health, failure configurations) of the monitored subsystem

This sequence of development is based primarily on risk. The displays currently in use by flight controllers show telemetry values. The worst way in which the replacement systems could fail would be to prevent the display of telemetry in a reliable, understandable manner. If the telemetry can still be seen even if the state and status inferencing fails, the software can still be used. Furthermore, since the state and status inferences also use telemetry values, it is desirable to develop the telemetry software objects before writing state and status rules. Consequently, the objective of displaying relevant telemetry values in readily understood fashion must be addressed before anything else.

Next, the state (position or orientation) inferences need to be developed. Once the state rules have been written, enough complexity has been incorporated that testing with real, noisy data streams is challenging. If the noisy data can be accommodated, then the status rules have a stable foundation. Otherwise, status inferences based on faulty state conclusions are useless. Status rules are developed last because they are based on expectations established by the state conclusions. Thus, the sequencing of development should be driven by a consideration of risks to the project.


Determine objectives, alternatives, constraints
Evaluate alternatives, analyze risks
Develop, verify next prototype
Plan next prototype

Panel A. Cyclic component of spiral development.



Panel B. Outward progression of spiral -- across development cycles.

Figure 2. Adaptation of Boehm's spiral model to operational prototyping.

Section 2
Understanding the System and Determining Requirements

The first step in building any successful software application is to understand the system and determine the software requirements. To ensure this step is achieved, several steps must take place. First, a preliminary project plan should be developed and agreed upon by all team members. Section 2.1, "Project Plan," provides some suggestions for activities in developing the plan. Next, the developer(s) should understand the human tasks in order to produce a correct and useful representation of those tasks. This includes understanding existing tasks, how these tasks relate to the supported operations, and how the tasks will be changed after the new system is in place. These issues are discussed in Section 2.2, "Understanding the Human Task," and Section 2.3, "Understanding Supported Operations." Once these initial steps are taken, the project scope and definition can be determined and included in the project plan. Finally, information requirements, discussed in section 3.4, should be well understood before programming takes place. Meeting each of the subtasks will provide order to the project and increase the chances of success.

2.1 Project Plan

The project plan begins during the first development iteration, shown in figure 2, panel A. in the "Determine Objectives, Alternatives, and Constraints" quadrant, and should be reconsidered with each development cycle. During each cycle, project team members decide whether the plan should be updated, depending on the formality of the plan, the importance of changes, and the resources available. The formality of the plan depends on the project team. Some managers will require a formal document. Others will request a briefing of the project plan. At any rate, the following information should be included in the project plan so it can assist in guiding the project and building consensus about project objectives.

2.1.1 Goals and Objectives

One of the most important measures to prevent "wandering project development" is to build consensus among managers and the development team on project goals and objectives and to clearly document these goals and objectives. An important consideration for truly innovative projects is that the project direction should occasionally change, based on new discoveries. Consequently, it is important to distinguish goals that will define project success from those which are secondary and concern implementation.

2.1.2 Risks

The explicit identification of risks and constraints helps to identify the best development sequence, the people to include on the development team, and the amount of effort to spend in avoiding the risks. Knowing up front the types of risks likely to occur can prevent common pitfalls and greatly enhance the development process.

2.1.3 Roles of Team Members

In operational prototyping, people with critical expertise interact with a prototype to improve its design. To achieve a successful software product, it is helpful to have a development team composed of the right mix of individuals, each having a key role throughout the project. The determination of critical expertise will depend on project objectives and project risks, but it will usually include user task expertise, softwaredevelopment expertise, and human factors expertise. While the number of people on an operational prototyping team should probably be between two and six, the specific number is less important than the following:

- All needed critical expertise must be represented.
- Every viewpoint must be discussed openly when interacting with the prototype.
- The whole team must meet regularly around the current prototype.

- Team discussions should usually be directed toward improvements, rather than acceptance or rejection.

- Each proposed improvement must be evaluated interactively by the team so that solutions can be negotiated which are reasonably optimal for each viewpoint.

Throughout the development process, each team member should be aware of the roles he or she needs to play. Specifically, there is the role played in software design and development, and later in construction of software test cases. This role is filled primarily by a software engineer. In addition, technical expertise is needed about the physical system that is to be monitored and the human tasks involved in monitoring that system. Development, user evaluation, and system testing are all important parts of the technical expert roles. Finally, a human factors engineer must oversee usability issues, ensuring that the final product meets the user requirements and improves the overall process. The following discussion highlights the roles of each team member and provides some insight obtained during the DESSY project.

2.1.3.1 *Software Engineer*

The software engineer (SE) should be knowledgeable about the development software and environment. He or she is both the designer and developer of the complete software system and will be responsible for ensuring that adequate testing is performed throughout the project life cycle. In the DESSY case, this role was shared by various individuals throughout the project.

Although it is useful for the SE be familiar with the development tool, in reality this person may be learning the tool during the development process. If this is the case for your project, exploit help provided from your tool's vendor. A willingness to aggressively pursue vendor help can be a great payoff to the project, while providing good feedback to the vendor on the usability of their product.

Also of particular importance for the software engineer is understanding performance constraints of the tool. Because the final application will be working in real time, it is important to know how to maximize performance. This is not a strength of DESSY, and in retrospect the developers feel a greater emphasis should have placed on performance.

2.1.3.2 *Technical or Task Expert*

The technical expert (TE) provides knowledge about the physical system being monitored, as well as the monitoring process. This person, who is usually an end user of the software system as well, must fully understand both the system hardware and the human monitoring process. Because other members of the team are unlikely to understand these aspects, it is crucial to have an experienced expert who is willing to be an active participant in the project. A predetermined number of hours per week should be set aside for the TE and SE to work jointly on the project and exchange ideas.

Besides providing technical knowledge and possibly filling the role of user-evaluator, the TE also has responsibilities in software testing. Although the software engineer can create the tests to check the program logic (verification), the technical expert must verify that the system correctly reflects the real process; i.e., the technical expert must check the model and associated rules and procedures that the programmer has constructed. Thus the TE is responsible for the validation part of the testing process.

2.1.3.3 *Human Factors Engineer*

The third major player in the software development team is the human factors engineer (HFE). The HFE is responsible for ensuring that the system meets usability requirements including not only the graphical interface, but the usability of the system as a whole. The HFE would likely present an initial interface design from which other team members can work, and would be responsible for overseeing the interface design throughout the development process.

Additionally, this team member must interface between the software engineer and the technical expert/end user, both (or all) of whom may not be trained in human computer interaction issues. The HFE must work with the SE in developing an interface that the software will support, while working with the TE to develop a meaningful interface. It is the HFE's responsibility to make sure the final software interface satisfies all team members.

2.1.4 Schedule and Resources

A schedule with resource allocations is a necessary part of any project. The schedule sets expectations about the project duration and the amount of effort that will probably be expended. Lack of resources and other responsibilities of team members, however, can lead to schedule slips. If your development team is similar to that of DESSY, it will be composed of dedicated people, but people who have other important responsibilities that by necessity often take precedence over software development. Be prepared to adjust the project schedule to meet the needs of the entire team.

Schedules for operational programming have some special features because of the exploratory nature of the approach. First, if some milestones are not negotiable, they should be specifically identified. This will allow contingency planning when surprises are encountered during development. Second, while major events and expected completion dates should be identified through the completion of the project, it is probably not reasonable to plan a detailed schedule throughout the project. Detailed scheduling should go through the next development cycle. After that, the project may need to respond to new discoveries by changing directions, thus rendering obsolete any detailed plans for subsequent iterations. Developers must strike a balance, sticking to the important objectives without unnecessarily restricting their explorations of ways to accomplish them efficiently and effectively.

2.2 Understanding the Human Task

Once a preliminary project plan has been developed, some effort to understand the human task(s) should be put forth to ensure that the intelligent system will accurately automate or supplement these tasks. (Malin and Schreckenghost, 1991) This includes both existing tasks that the potential users perform and new or altered tasks after the system has been implemented. The following sections provide helpful hints on accomplishing this understanding and outlines some specific tools that can be helpful.

2.2.1 Existing Task(s)

Understanding the existing tasks of the user (flight controller) can be very challenging, especially when you must attempt to emulate those tasks. The need to have a team member who understands the user tasks emphasizes the strength of participatory design in operational prototyping. In ideal circumstances, the end user is always available to assist the programmer in developing software that precisely fits in the user's workplace. Because circumstances are rarely ideal, however, the following recommendations are made to assist the developer in understanding the tasks.

Regular meetings should be held with the end user (technical expert) to discuss the tasks. Initial discussion will cover existing tasks; eventually the development team will need feedback from the user on the software's model of existing tasks; finally, the team will need to evaluate the impact of the software on the tasks.

Work should be done in the end user's work environment when possible. This will allow all team members to objectively observe the user performing the task, and the user is likely to do a better job at evaluating the software if software inspection is done within the user's own environment. Finally, the system will eventually need to be integrated into the user's work environment. Occasional developments in this area will greatly expedite the ultimate integration process.

In addition to working closely with the end user, the development team may wish to map out the existing tasks. Section 2.2.3 discusses common tools available for understanding the user's tasks. This is particularly important if the end user in unavailable throughout much of the development process.

2.2.2 The Impact of the New System

Once the existing human tasks are well understood, the development team should evaluate how the new intelligent system will effect those tasks. This is important to ensure that the right problems are being solved and the new software will provide a true enhancement over the existing system. It is also important to provide some up front insight to the end user on upcoming changes to existing tasks.

Because the applications covered in this guide are of the automated monitoring type, a key element of the new task is that of human supervision of the new system. It must be understood how this supervisory task impacts other responsibilities of the user. For example, will the intelligent system display replace existing displays or will the user have an additional screen to monitor? Will the user be required to interact with the intelligent system or will it require only monitoring? In any case, the impact of the system should be analyzed up front and the user should be made aware of this impact as soon as possible.

2.2.3 Tools for Understanding the Task

Understanding the human monitoring task may be difficult for members of the development team not familiar with the users' work environment. Interviews with end users, both those on the development team and other users, are recommended. If possible, it is also helpful for the software developers to sit in with users during operations. This allows objective observations of the work environment and determination of how the software system will fit in.

Other tools available that might help the team understand the users' tasks are data flow diagrams and state diagrams. A data-flow diagram reflects changes in data as the system changes. A state diagram reflects changes in system state due to data changes. Both these tools might be used as a communications medium when communicating with the user. In any case, building a good relationship with the end user, whether or not the user is a member of the development team, will help the team better understand the task at hand and lead to a better end product.

2.3 Understanding Supported Operations

To successfully establish the project scope and definition, the supported operations must first be understood by the development team. This includes familiarity with both the hardware system and the telemetry downlink parameters. Elements of the operations are composed of system components, telemetry data, system configurations, failure space, and relationships between the system being modeled and other systems. Outlining these elements will prevent wandering of the project and keep the development team focused.

2.3.1 System Components

The first step in understanding supported operations is the identification of the pieces and subpieces of the hardware systems that are being monitored. Identification of the hardware to be supported will help in establishing preliminary boundaries for the software system. As with other steps in building the system, even hardware identification may be iterative. As the system evolves, the scope of the hardware may change to reflect interfaces with other systems or lack of available telemetry data.

A goal of DESSY is to eventually monitor telemetry associated with all parts of the Shuttle arm. To manage such a large set of data, the original DESSY designers divided the arm into several functional subsystems, two of which are covered in the existing DESSY and thus used throughout this guide. The first subsystem to be modeled was the MPM/MRL subsystem. The MPMs, or manipulator positioning mechanisms, are the pedestals the arm rests on when it is cradled in the payload bay. MPMs can be stowed or deployed, respectively indicating whether the arm is in the rolled in or rolled out position in the payload bay. The MRLs, or manipulator retention latches, are the latches that latch the arm down when it is cradled in the payload bay. The MRLs can be latched or released. There are four MPMs and three MRLs. They are depicted in figure 3. (Collins, 1988)

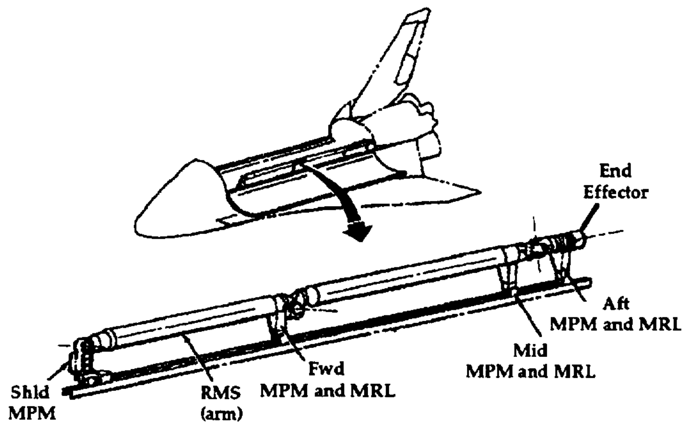


Figure 3. Remote manipulator system (Shuttle arm).

Hardware components are hierarchial in nature, and once the primary hardware is identified, the team must delve deeper into the system to find additional hardware components. For example, each MRL and the shoulder MPM (where the arm is attached) contain two motors. Because there is telemetry available for each of these components, they were modeled as separate objects within DESSY. Other hardware components identified were various switches and power sources or paths. Not all hardware components have sensors, and although they might be modeled within the software system, they cannot be directly monitored.

The end effector system, the second RMS subsystem to be modeled, was also broken down into its hardware components. At the highest level, the system has two functional subpieces: the snare mechanism and the rigidizing mechanism. The snare mechanism can be in an open or closed position. The rigidizing mechanism can be rigid, derigid, or extended. The end effector also has two motors that drive both these mechanisms. Other hardware includes switches and several external systems. The location of the end effector on the Shuttle arm is also depicted in figure 3.

Identification of system hardware is crucial because not only does it allow the developer to understand the functional system, it provides the initial template for the expert system model. The next section, Data, will further refine that template. Identification of data may be thought of as the hardware model at the sensor level for which telemetry is available. How to choose the appropriate data to include in the model is discussed below.

2.3.2 Data

2.3.2.1 *Finding the Data*

Once you have an idea of what hardware will be monitored, finding the data associated with that hardware should be straightforward. As with other steps, finding data will probably be an iterative process. Telemetry that is flagged for use may drive additional hardware to be included in the design and vice versa. For the mission control environment, most existing data is displayed on the console displays. Obtaining a copy of all the associated console displays for the subsystem being modeled is a good place to start.

Other important sources for gaining information about data include drawings of the Space Shuttle panels, schematics, and the SDRS DTE Display Description Report (SDRS, 1992). The technical expert or end user will need to assist the developer with data identification. Obtaining a copy of SDRS is of particular importance because this document contains the MSID numbers needed by the application to get the individual pieces of data. Schematics will also have specific telemetry information in them and are important to use in understanding the data flow. Beware of questionable information, however.

Documentation can contain mistakes, particularly when it includes much numeric data. As a rule of thumb, when in doubt ask the expert.

2.3.2.2 *Understanding the Data*

Once the data sources have been located, the challenge of working with data just begins. The next important step is to understand the data and how changes in system hardware configuration lead to data changes. A good place to start is with the schematics. All team members should learn to read schematics. This is a crucial step in thoroughly understanding the system and related data. Determine the data set to be used and be prepared to stay within the monitoring/diagnostic bounds that it provides, at least initially. You may want to identify other data that would be good candidates for expansion of the knowledge base, or perhaps will be needed when the existing set just can't to the monitoring job.

Other considerations necessary in DESSY involved working with binary telemetry data. The data state (1 or 0) must be understood; i.e., it must be determined which state implies active and which inactive. For example, in DESSY there were several key types of data available. Depending on the type, some data had an active state of 1 and others had an active state of 0. Table 1 displays some types of DESSY data and their states. This information can be obtained from system schematics or from the expert.

Table 1. Active And Inactive States Of Binary DESSY Telemetry

Data	Active	Inactive
microswitches	1	0
opstats	0	1
enables	0	1
command	1	0
mech pwr	1	0

2.3.2.3 *Subtleties in Working with Data*

Finally, there are some subtleties in working with telemetry data of which the development team should be aware. Below are some specific questions and issues to keep in mind when looking for telemetry data and understanding how it works. With these are included experiences with the MPM/MRL and end effector subsystems.

Telemetry Overlap Within the System—Is there any telemetry used in more than one operation, providing different indications at different times?

> *Some opstats (motor data) are used in both MRL and MPM operations. This same data indicates either MRL or MPM motor activity, depending on the operation. (Operation must be determined by context.)*

Data Overlap with Outside Systems—Is there any telemetry used also in other systems?

> *Some DESSY opstats are also used for the KU band antenna. This same data indicates either DESSY activity or KU band activity, depending on the operation. (Operation must be determined by context.)*

Lack of Data for Symmetric Operations—What are the cases where there is telemetry for one operation, but no telemetry for the reverse (symmetric) operation? Data will not always be symmetric for symmetric operations.

> *In the MPM system, there is a stow command indicator, but no deploy command indicator. Likewise for MRLs, there is a latch command indicator, but no release command indicator. Additional data had to be used to determine command.*

Single Telemetry Point—What are the cases where there is only a single piece of telemetry on which a conclusion can be based?

> *Even though we have an MPM stow command indicator, that single data point might fail. Therefore we look at additional data related to stowing to make the conclusion. Whenever possible, we want to look at multiple data when making a conclusion.*

2.3.3 System Configurations

The process of understanding system configurations involves knowing both the hardware and related data that are used in each configuration, and understanding the relationship between changes in the hardware and changes in its corresponding data. Not only does the developer need to identify system states, which identify configurations, but he or she also needs to understand the procedural operations involved in making a transition from one state to another. Two tools are particularly useful in understanding system configurations and procedures. Each is discussed in the following subsections.

2.3.3.1 *State Diagrams*

An excellent tool for mapping changes in state of the hardware system is the state diagram, where the states and transitions are represented as a directed graph. Each system state, or configuration, is illustrated as a state node, and the transitions between them are illustrated as edges. For the purposes of modeling the system based on telemetry, transitions between nodes are driven by changes in the telemetry.

The example below illustrates MPM system states and transitions during the MPM stow procedure. Table 2 identifies each MPM state involved in a stow and shows the related data states (active/inactive) during each phase. Note that the transition state is identified as a valid state. This is appropriate because data changes are involved both entering and leaving this state. Furthermore, even though it is a transition, i.e., a change from one static state to another, there is a finite time period that the physical system remains in this state. Thus, in the example a deployed state, an in-transit state, and a stowed state are identified. Explanations of the data presented in this example are as follows. Table 2 provides the data values of the steps of the MPM stow procedure.

When identifying state changes for your system, we recommend constructing a table similar to that shown above. To construct the table follow these steps:

- Identify the valid states, including transition states.
- Identify all data associated with these states.
- Identify data values for each state.
- Include new states if data changes indicate you should do so.

After a table like that shown in table 2 is constructed, you should be prepared to construct the state diagram. Developing the state diagram will lead to better understanding of the monitored process. Figure 4 shows a first pass for a state diagram for the MPM stow procedure. (It did not include the State 2 of table 2.) All stable (or static) states, the beginning (deployed) state and the ending (stowed) state were identified. The transition state was initially represented as an edge. Since the stowing process takes 32 seconds, the edge was labeled accordingly.

Table 2. Data Changes During the MPM Stow Procedure

MPM Stow Procedure

State 1 MPM deployed	deploy microswitches	active
	stow microswitches	inactive
	stow opstats	inactive
	stow command	inactive

State 2 MPM stow-in-transit	deploy microswitches	inactive
	stow microswitches	inactive
	stow opstats	active
	stow command	active

State 3 MPM stowed	deploy microswitches	inactive
	stow microswitches	active
	stow opstats	inactive
	stow command	inactive

Key: deploy microswitches: indicates a deployed state
stow microswitches: indicates a stowed state
stow opstats: indicates motors are stowing
stow command: indicates a stow command is being given

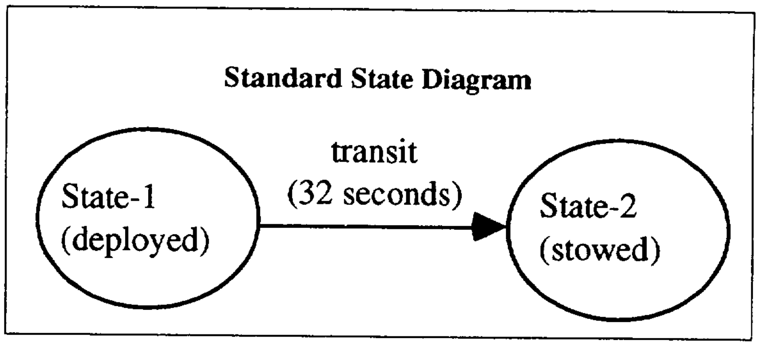


Figure 4. State diagram with transit edge between static states.

However, the "transit" state concept is important. After the initial diagram was complete, it was then revised to include the transition state as a valid state in the diagram. Note that both state diagrams are equivalent representations of the same process. The transition edge of figure 4 became a new state with "epsilon" edges going between it and the states it previously connected. The epsilon notation, commonly found in state diagrams, simply means there is an instantaneous transition between states with no time delay involved. Figure 6 illustrates the revised state diagram used throughout DESSY development. Although either diagram could be used, the one depicted in figure 5 represents the states and transitions more precisely. (Hopcroft, 1979)

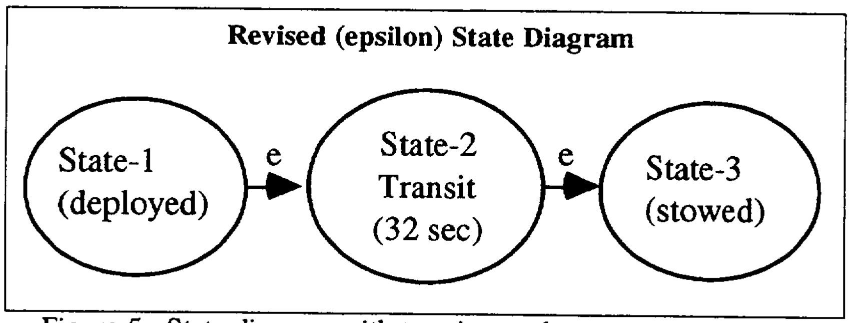


Figure 5. State diagram with transit state between static states.

Once the procedural data changes have been identified and the state diagram developed, the developer will have a fairly complete view of the operation. At this point, the developer should construct a state diagram covering all procedures, with data changes labeled directly on it. Figure 6 provides a more complete state diagram, an example of the end effector snaring system. Six states have been identified, along with the three microswitches that indicate state changes. In this example motor (opstat) and commanding data that affect the changes are not shown. The procedures shown include the snare closing with capture, closing without capture, and opening. The snare closing with capture procedure is represented in the bolded states. The important information to note from this figure is the states and their relationships to one another, the data values, and the timing information indicating how long the system would nominally remain in the given transition state.

Note that there is no failure information represented in figure 6. State diagrams may be extended to include non-nominal operations. Figure 7 shows a section of the snare state diagram from figure 6 with an "aborted close" state added. The concept illustrates a simple extension to the state diagram as defined above. It will be covered in more detail in section 3.3.4, where status rules are discussed.

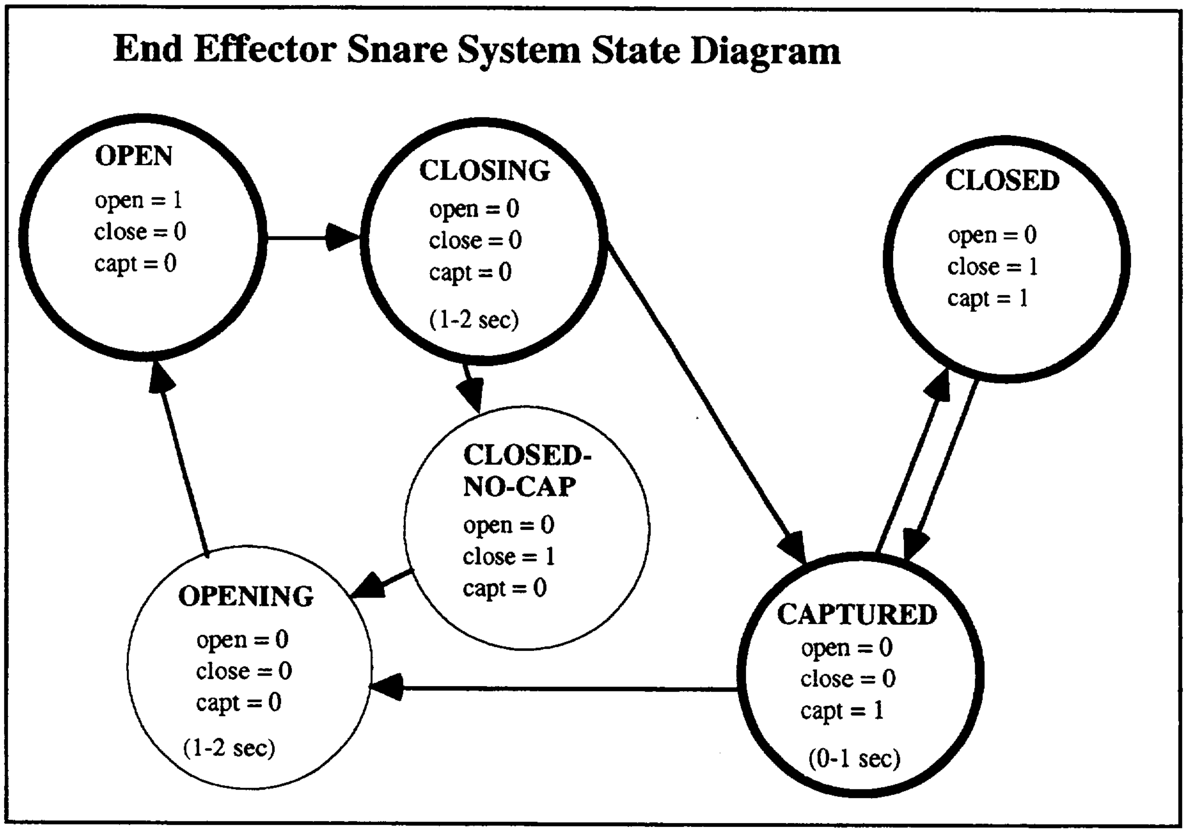


Figure 6. State diagram for the end effector snare system.

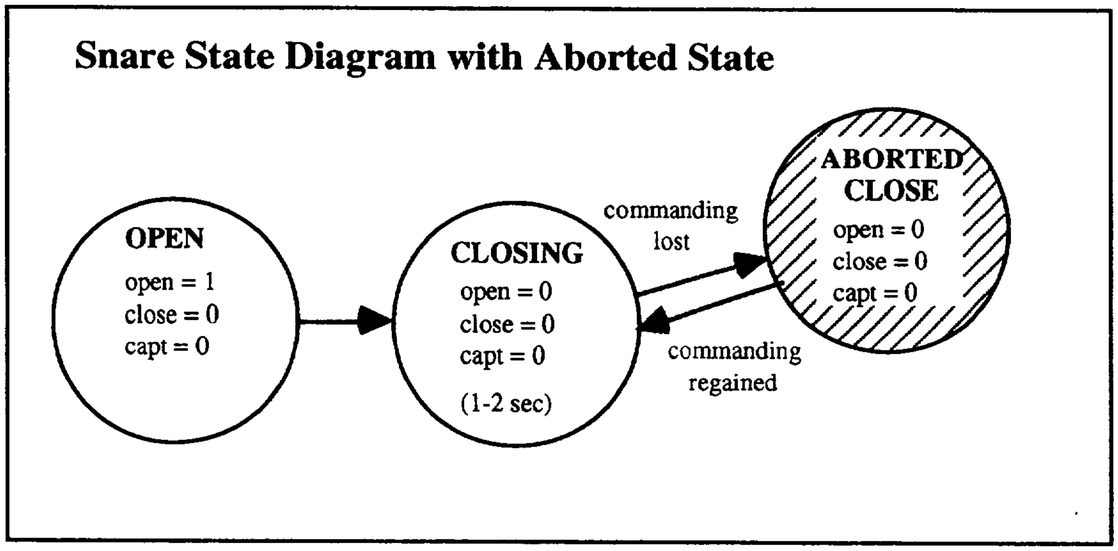


Figure 7. Subsection of the snare state diagram depicting an aborted state.

In summary, state diagrams are essential step to understanding the system configurations that the team will need to model. They should be constructed as soon as the relevant data has been identified. In addition to providing information to the developer, their graphical nature allows them to serve as an effective communication tool between the developer and system expert or end user. State diagrams are a classic computer science tool that should be fully exploited.

2.3.3.2 *Procedural Timelines*

A second tool that can be extremely helpful in understanding the system procedures and transitions is the procedural timeline. Timelines give a chronological step-by-step description of the system transition states. They may, in fact, contain the same information as the state diagram; however, they may be more useful in some cases. They are most useful in modeling procedures with multiple transition phases.

The timeline shown in figure 8 shows the end effector snare capture sequence. This sequence is illustrated within the state diagram in figure 6. The corresponding state nodes are shown in bold circles. The snare system is initially in the static state of "open." The "closing" state lasts 1-2 seconds and the "captured" state lasts about a second. The final static state is "closed." The boundaries between the timeline bars are equivalent to the epsilon edges of the state diagram.

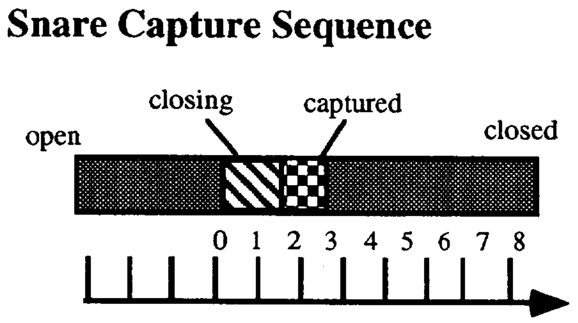


Figure 8. Snare capture sequence timeline.

Figure 9 shows the complete end effector capture sequence of which the snare capture of figure 8 is the first part. Also shown are the two rigidizing steps and the final state of "closed and rigid." The end states of the timeline are static or steady states, while transition states are depicted within specific time intervals.

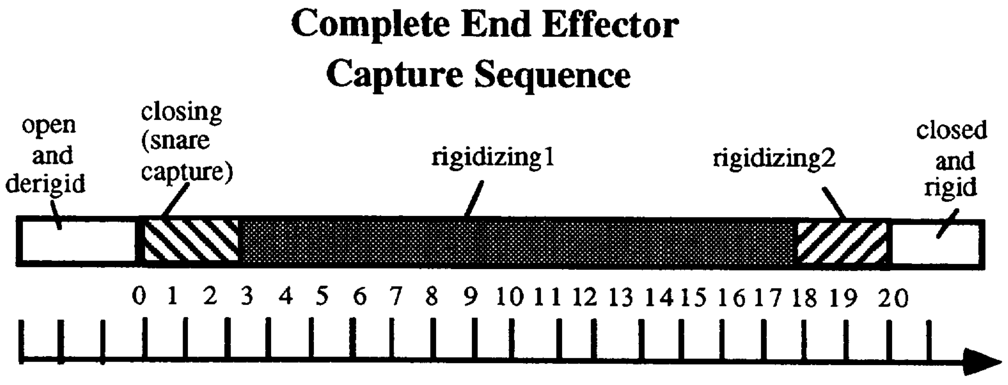


Figure 9. Complete end effector capture sequence timeline.

An extension of the timeline tool might include labeling of data changes at bar transitions. Basically, it is up to the developer to get the most out of a tool like this or the state diagram tool. In DESSY, both were used, as well as other means of representing the system to enable the team to gain a more thorough understanding of the system and its operations. As with state diagrams, this tool should be exploited to the fullest extent to aid the developer in understanding the physical system.

2.3.4 Creating a Storyboard

A storyboard captures the types of information that the intelligent system will display and the types of interaction between the intelligent system and the human user (Schreckenghost, 1990). It is also the forerunner of the user interface. Once data has been identified and operations understood, work on creating the storyboard can begin. This will include sketches of displays and information about how the human would interact with and interpret these displays. A "story" can be constructed illustrating how the human would interact with the intelligent system in a sample scenario. Initially, sample scenarios should address normal operations, but can later be extended to include failures. These scenarios can also be used when developing test cases (see section 6.3).

The storyboard is useful in working out information requirements with the user. This technique can be used to distinguish requirements for functions needed to do the task, from requirements about presentation. For example, two telemetry values may need to be compared over a specified time—a functionality requirement. The means of comparison might be using overlaid plots—a presentation requirement. Both types of requirements should be specified.

Once in place, the intelligent system will change operations. Another use for storyboards is in stepping through typical operational situations to test how well the user can respond to these situations using the new system. This approach helps in assessing operational changes and in preventing adverse changes (e.g., changes that increase workload or cause loss of needed information). Adjustments can made to intelligent system requirements or possibly to operational procedures.

Storyboards are often created using simple drawing and word processing programs, although they might be sketched by hand or developed in the selected prototyping tool. The storyboard will serve as a template for later design and development, but it is unlikely that the final design will closely reflect details found in the initial storyboard. They are, however, very effective for conveying working ideas to other members of the development team. Use this tool to capture early design ideas, but remain open to change as the design evolves.

2.3.5 Failure Space

Understanding the failure space of the hardware system to be monitored may be the most difficult part of the development tasks. Although, as stated in section 1, the authors believe a real-time monitoring system is not usually a failure diagnostics system, there is still status monitoring that must be done and the overall health of the system must be determined. In the DESSY project, the scope of failure detection was limited to status information that could be determined from telemetry data. This would not include, for example, information heard over voice loops. Even with this definition, mapping out the failure space was challenging. It can be difficult to find all relevent data for failures you want to include and justify the elimination of failures that are beyond the scope of the system.

There are, however, some standard types of failure that may be included in real-time monitoring systems. Each of these types is presented in the following subsections.

2.3.5.1 *Transition Timing*

Once state diagrams and procedural timelines are complete, the developer will know the length of time the system normally spends in each state. Thus a feature that should be implemented in the monitoring system is a mechanism for timing the procedural operations. In the DESSY project, we implemented a stopwatch-like timer. It can be started, stopped, paused, reset, and resumed. DESSY timers monitor all significant procedural activities.

Once a procedure begins, the appropriate DESSY timer is started. If the procedure (or procedure segment) completes within the allotted time, a nominal time will be reported by the timer. This nominal time has been preset in the timer or is determined by expert system rules. It is usually a range rather than a single number. If, however, the procedure spends more time in a transition state than is known to be allowed, an anomaly can be flagged. In the MPM/MRL subsystem there were actually several times of interest per procedure. Because two redundant motors drive both the MPMs and MRLs, there could be a nominal time if both motors worked properly, a "single-motor-drive" time if one motor failed, or a "two-phase-motor-drive" time if one motor worked nominally and the other functioned in a degraded state.

In addition to obtaining transition times, the developer must consider how to flag a transition failing to start or end. Both these actions involve using information available in the timer. When appropriate data, often command or motor activity, indicates a transition is taking place, the timer is started. If after some time period, however, the data reflecting positional indicators do not respond, a failure to begin the transition has occurred and can be declared. Because sets of data do not always change simultaneously, it is necessary to include a "subtransition" period from a static state to a transition state, i.e., a second or two to allow the transition to take place. This was not modeled in the state diagrams, but in the real system must be considered.

Finally, just as the monitoring system must watch for the failure of a transition to occur, it must also flag a system being stuck in a transition state. This is not straightforward, because unlike a transition ending marked by changing data, a transition that fails to end has no data indication, but rather a lack of indication. In this case, the timer must flag that the transition did not end. Specifically, a timeout limit is identified. If the timer should reach the timeout limit, a failure has occurred.

Figure 10 summarizes the type of events that should be timed. Each timing indication leads to a different status result (including nominal). Note that each status actually corresponds to a time range, and the labels show the maximum time for that status. Each timing space shown below is covered by a different rule, and all possible time ranges are covered.

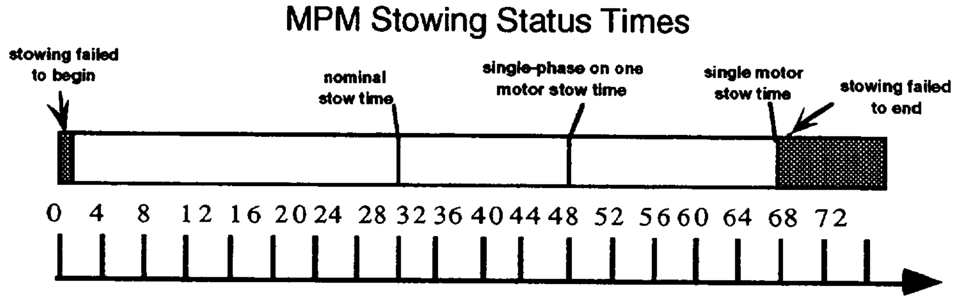


Figure 10. MPM stowing timeline with transition timing.

2.3.5.2 *Data Questionable On/Off—Ramifications to the System*

Another type of status a real-time expert system should detect is a data value known to be in an incorrect state. DESSY flags a data value as "questionable-on" if it is active when it is expected to be inactive (according to other system information). Likewise, a data value that is inactive when it is expected to be active is flagged as "questionable-off." The method of detecting the questionable-on/off data status varies significantly depending on the type of data. Some parameters that are used to identify questionable data are system state, other redundant sensors, and expected transition activities.

Note that the descriptor "questionable" is used to mark suspect data. This is because the data may or may not represent a known failure, and the expert system (and human) may not have the necessary information to conclude a failure. The flagging of data as questionable-on/off in fact is usually the first step for any further DESSY status conclusions. The questionable flag allows DESSY to alert the user that something in not nominal without making any uncertain failure conclusions.

Once the data has been marked as questionable, however, further rules may process this information to make conclusions about the data and its ramifications on the hardware system. For example, it may be determined that the questionable-on value indicates a "stuck on" microswitch. If the appropriate information is available, the status can be upgraded.

A second use for the questionable data status feeds into the diagnostic rules, giving them a less certain or definitive nature. For example, in the MPM system a certain active deploy microswitch would inhibit one of the redundant motors from stowing, causing a single-motor stow to occur, should stowing be commanded. If the microswitch has been given a questionable-on status, a system status of expect-single-motor-stow (as opposed to single-motor-stow) is provided. This strategy allows DESSY to present suspect anomalies to the user without taking the risk of giving the user faulty conclusions. Once additional information is available and the motor inhibit is confirmed by other data, the system message can be upgraded to single-motor-stow.

2.3.5.3 *Failure of Data to Change During an Expected Transition*

A third type of failure the expert system should monitor is failures during transitions. Given an expected operation, the human and expert system should know a priori the set of data that will change. If any subset of this data does not change at the expected time, an anomaly is obvious. The expert system must have status rules capable of identifying appropriate pieces or sets of data that do not change when expected and flag that data at the time of transition. This includes, but is not limited to, failures detected through transition timing.

When flagging suspect data, DESSY will usually mark the data as questionable. However, during a transition the evidence may be strong enough to identify real failures rather than just questionable situations. Depending on the specifics of the scenario, the developer may be able to conclude immediate status information at this time.

An example of "failure of data to change" may occur at the beginning of an MPM stowing operation. At this time two stow opstats indicating motor activity will become active, and the two stow microswitches indicating stow position will go inactive. A stow command indicator is also available. Thus for this particular transition, there are five pieces of telemetry which directly and jointly indicate the operation initiation. If any one (or more than one) of these five pieces of data does not act as expected, an anomaly can be flagged.

To conclude, the system developer must identify the entire set of data that indicates a transition and consider the possibility of each piece (or subset) of data in the set not responding. Each piece and subset identified will likely require a separate status rule.

2.3.5.4 *Multiple Data Values Active, Indicating Conflicting States*

A final type of anomaly that the intelligent system should identify is that of data values indicating conflicting states, i.e., an impossible configuration. These anomalies are flagged when the system is static and no operational activity is expected to occur. They are discovered either by observing that two conflicting data values are simultaneously active, in which case they might both be marked as questionable-on; or for a more sophisticated approach, given that the system is in a known configuration, the suspect data that conflicts with this configuration can be marked as questionable-on and possible further diagnostics performed. This second approach is preferred when possible because it further isolates the suspect telemetry.

An example of this type of failure occurs when the MPM system is in the stowed configuration, and a single deploy microswitch turns active. This conflicting deploy microswitch would be marked as questionable-on. Alternatively, a single stow microswitch becoming inactive would be marked as questionable-off. Another example occurs when the end effector snare is known to be open, and with no procedural operations the close microswitch becomes active. The close microswitch is marked as questionable-on.

The developer and expert must outline all known nominal configurations of the system, and for each configuration, determine data values from other sensors that would indicate conflicting conditions. Then for each piece of data or subset of data, a separate diagnostic rule can be formed.

Although this may not be an exhaustive set of failure types, they are the most common and easily identified. DESSY detects all of these types, and in many cases passes on the information to further failure detection rules. A complete listing of failures in the DESSY end effector example is given in Appendix A.

2.3.6 Relationships with Other Systems

Finding the relationships between the system being modeled and the outside world can be elusive. The expert must be thoroughly involved in this step. How the system affects its neighboring systems, both in nominal and off-nominal operations, as well as how those systems affect it, should both be modeled. System overlap and dependencies may occur during select normal operations or happen only during failures. In either case these scenarios may add complexities to the monitoring the system.

Specific examples of overlap between systems include ambiguities of data that belong to multiple systems. That is, a single piece of telemetry may represent different systems at different times. For example some opstats that indicate MPM/MRL motor activity can also be active when the KU-band antenna is being used and there is no current MPM/MRL activity. This is due to a change in configuration outside the scope of the expert system. Additional data must be used to indicate true MPM/MRL activity so that the KU-band activity does not falsely indicate this. As another example, if a logic switch fails or is turned off, subsets of the MPM/MRL opstats will become active, depending upon the particular switch. Again, it must be assured that this does not incorrectly suggest MPM/MRL activity.

Thus the developer must be aware that there may be overlaps in the use of certain subsets of telemetry and design the system to account for them. If this is not done, the result is most likely to be false conclusions of system activity.

Other issues in dealing with external systems include understanding how failures outside the scope of the monitored system will affect system performance. An obvious example is a power related failure. If a failure occurs in the power source to your system, the system will likely be reconfigured to allow for a redundant power source. The developer must be aware of how this affects the system model. It must then be determined if accounting for the situation is within the defined scope of the project.

Likewise, the effects that failures in the monitored system have on external systems should be determined. Although this is stepping into diagnostics, the effects your system will have on the outside world may be easy to determine in some cases. Again, it must be decided whether accounting for these situations is within the project scope.

In conclusion, it is important to remember that the relationships between the monitored system and external systems do exist and must be identified and understood. The expert will play a key role in identifying this information for the developer. Once these interfaces and interactions have been identified, the team can decide which are within scope of the expert system project.

2.4 Information Requirements

Information requirements should be well understood so that all necessary information is available to the development team throughout the development process. Information sources should be identified and the information sought out before application development begins. The first place from which information can be extracted is the current user displays. The expert/user can identify additional information sources used in his or her job. Finally there are some standard information sources in the aerospace environment that the development team should acquire.

2.4.1 Information from Current Displays

The user's current displays are a crucial source of information because they contain the standard set of data that the user has available. Understanding the displays and how they are used should be one of the first joint tasks of the development team. Specifically, the user must identify which data is used for which subtasks, which is displayed but not used, which is most crucial, and which is most frequently used. Questions like this will help shape the user interface of the expert system display and will help define the initial storyboard.

As the team works to understand how the current displays are used, they must objectively identify both positive and negative aspects of the displays. Factors might include screen layout and groupings of data, text vs. graphical representations, and color. Other questions the team might ask are: How is the user notified when data begins to change; where and how is status information displayed; and how and why are subsets of data grouped? The human factors team member is responsible for assisting the expert/user in deriving this information.

2.4.2 User-Requested Information

It is likely that the user makes use of information not found on existing displays. Any additional information that the intelligent system can provide will further help the user do his or her job and add to the usefulness of the software. Here the difficulty often lies in identifying this information. Again the human factors team member has the responsibility to assist the user. Team members must go through the user's monitoring process to identify the less obvious information on which the user relies. Examples from DESSY include stopwatch timing and handwritten logs. Having this information available through the intelligent system further increases the usefulness and credibility of the software.

2.4.3 Gathering of Information

Before the design and development phase of the project begins, all relevant sources of information should be located. Finding these items ahead of time will make the process more efficient. Below is a list and brief description of items used in DESSY. The expert/user should assist the team in locating this type of information.

- Schematics and Drawings—Provide assistance in understanding the physical system and how failures occur and propagate. Also may provide telemetry information.

- SDRS—Provides displays with listings of available telemetry including MSID number, label, type, and location on display.

- Console Handbooks—Provide detailed descriptions of the system, its procedures, and the monitoring process.

- Current Displays—Provides a graphic of what is currently used during the monitoring process.

- Failure Modes and Effects Analysis (FMEA) Documents—Provides detailed information about failures, their causes, and their effects.

- Malfunction Procedures—Provides information on malfunction procedures for failure diagnostics and recovery.

Once this information is located, it should be made readily available to all team members. Keeping these documents and all other information related to the expert system in an easy-to-access central location will make the entire team's job easier.

Section 3
Building the System

Once you have completed the preliminary activities of understanding real-time monitoring systems and determining the requirements for the system you wish to build, you will be ready to move on to the next phase of the intelligent system project. Section 3 covers both the design and development phases of the application. The authors have chosen to merge the design and development processes into a single step because the high level nature of the G2 (or G2-like) programming tool supports, and in fact necessitates, this concept. Object-oriented, graphical development tools support creating the software design within the tool itself. This leads to development as an extension of existing templates rather than recreating definitions and objects in a separate development step.

In the DESSY project, the authors found that the design and development phases went hand in hand, forming an iterative process. The DESSY object structure was designed by creating a definitions class hierarchy in G2. Once these definitions were created, they were immediately available for use, i.e. development. It did not make sense to separate object design, definition, and implementation. The definitions and their object instantiations did frequently change, thus leading to iterative development.

The key concept one must accept to use this section is that this approach is one of iterative refinements of a single design-development process. This section presents six major steps in the design and development process. Section 3.1, "Organization of the Knowledge Base," prepares you in the setup of the architecture of the knowledge base, including grouping of objects, rules, and displays. Section 3.2, "Object and Structure Design," provides insight into how the physical system should be modeled in object-oriented programming. Section 3.3, "Rules," is essential reading for developers of real-time monitoring rules. Rules have been broken into several key categories, and this section provides an explanation of the use and development of each type. "User Interface Design" is discussed in section 3.4, and section 3.5 covers issues involved in "Setting Up for Real Time." Finally section 3.6, "Setting Up for End Users," provides a heads-up on subtle yet key issues involved in preparing the system for end users.

NOTE: Because section 3 covers the development of the knowledge base, it will be more closely related to the G2 tool than other sections. Many of the examples will be G2 specific. When possible, examples will be generic enough so that developers using other real-time expert system tools will find the information useful. However, the focus is on providing help to developers using G2, and in fact providing many G2-specific tips along the way.

3.1 Organization of the Knowledge Base

A well organized knowledge base will be easier for the development team to understand. The organization will be iterative, and the developer's willingness to change the organization as the knowledge base matures will lead to an application that all members of the team can use. This aspect of creating the intelligent system parallels traditional programming, and the experienced programmer will be familiar with these concepts. This section covers both module and workspace organization. A module in the G2 application corresponds to a file containing a subset of the knowledge base. A workspace is a work window within a module on which objects and rules and all other G2 items are placed. It is important for the knowledge base to be organized at both levels.

3.1.1 Module Organization

Modularization is an important part of knowledge base design, although there is not one right way to modularize. As the system evolves, changes will be made to the module architecture.

G2 modules are files which contain pieces of the knowledge base. As with traditional programming, these files dependend upon one another. For example, the lowest level modules contain definitions. Once these are constructed, object modules can be built which use them. Rules and displays which use those objects can then be built within higher level modules.

If you are just beginning to construct the knowledge base, you will likely start out with all work in a single module. Once the knowledge base grows, you will "modularize" by splitting it into the appropriate files. The authors found this to be a tricky task and recommend that you consult the G2 manual carefully.

Figure 11 shows a simplification of the module hierarchy for DESSY. At the top is *dessy-main*, which contains the highest level information. It directly depends on three modules. *Interface* contains data-connection information, *dessy-top-level* contains rule and object definitions, and *dessy-sims* contain test cases. Note that *dessy-main* is indirectly dependent upon all other modules as well.

The *dessy-top-level* module uses the MPM and MRL definitions supplied by *mpm-mrl-defs*. This module in turn uses class definitions in *dessy-defs*. For example, an MPM object definition is found in *mpm-mrl-defs*. An MPM class is a child or subclass of the class RMS. The definition for RMS is found in the *dessy-defs* module. Therefore the *mpm-mrl-defs* module depends on the *dessy-defs* module.

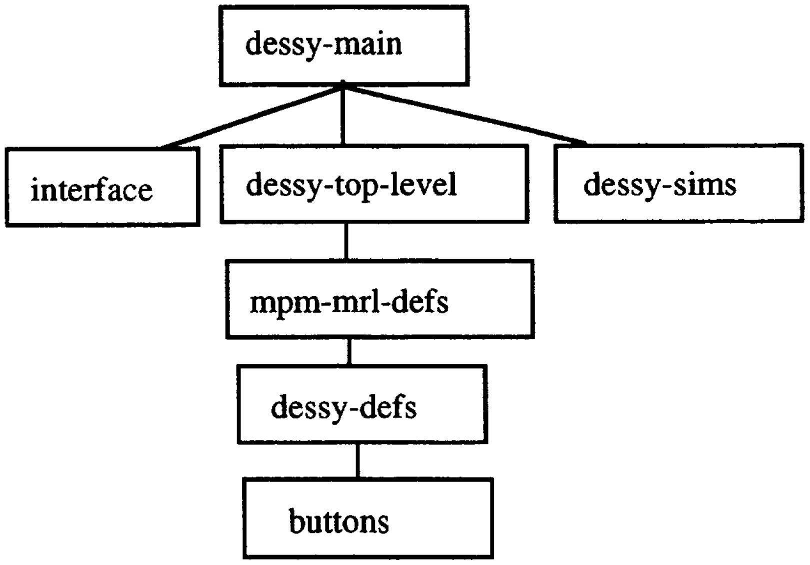


Figure 11. Simplified DESSY module hierarchy.

Why is it so important to modularize? The primary reasons are tractability and reuse. If the knowledge base is divided into smaller manageable parts, it will be much easier to add and document changes. The biggest win in this modular architecture, however, is reuse. Because work has been separated into smaller parts, the parts have good potential to be reused within the knowledge base or in other applications. The *buttons* module was a standard G2-provided module containing button definitions and corresponding rules that was easily merged into DESSY. Once this module is merged into the low-level *dessy-defs*, *buttons* may be used throughout DESSY.

Modularity also provides an option of software integration or segregation. Figure 12 illustrates how both DESSY modules can be loaded and run together or how each module can be used separately. A top level module is created that contains information about which system(s) to load. In the user environment, there were times when only one DESSY module was needed, and times when both were needed. This design satisfies both requirements.

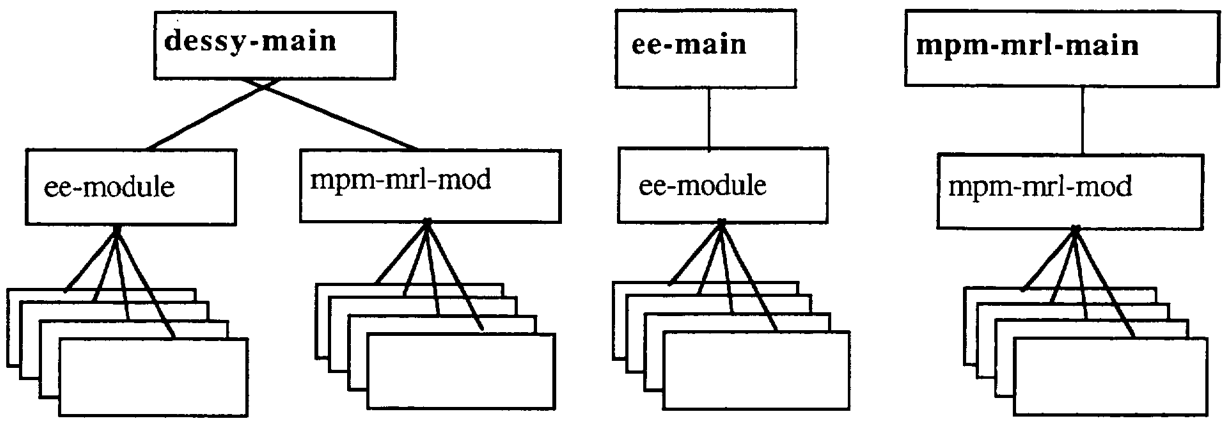


Figure 12. Multiple system configurations for DESSY.

Another module reuse win for DESSY was in the creation of a separate DESSY training application. The DESSY tutorial used existing DESSY modules with an additional module layer built around them to replace the normal interface. Because of the design, the definition and rule modules were easily plugged in to the tutorial. This provided the benefit that when the rule module was edited for DESSY, the tutorial automatically received those changes. Modularization was a big time saver in this aspect of the problem.

The only disadvantage that arose due to modularity in DESSY was that it impacted software loading time. Many small G2 files take longer to load than a single file containing the entire knowledge base. For the mission control environment, these loading minutes can be crucial. G2 provides a capability for saving the knowledge base as a single file. Thus modularity may be kept in the development version of your intelligent system, and the delivered product can be merged into a single file.

3.1.2 Organization within Modules

Workspace organization parallels modular design. The developer should establish a convention for organizing workspaces within the modules. G2 provides the capability for setting up knowledge base organization, but does not enforce or even encourage the developer to proceed in this manner. G2 is global, giving the developer the capability to program with no organizational structure whatsoever. Because of this lenience, we stress the importance of the developer maintaining self discipline and developing the knowledge base in a structured, well organized manner.

Because each workspace is assigned to a G2 module, workspace organization occurs within a module. A parallel organization should be maintained for all modules. First, set up a root workspace with pointers to all other workspaces—a top-level workspace that allows movement to any other workspace, either directly or through other workspaces. Figure 13 illustrates this concept.

To implement this concept in G2, create a workspace and name it something like *module-name*-root. Then create a definition called "workspace (ws) holder" or something similar. The icon should be something simple, like a small box. For each new high-level workspace needed, create a ws-holder object and give it the name of the workspace you want to create. Create a subworkspace for the ws-holder. That will be your new workspace. These subworkspaces can either be work areas or can hold workspace holders to other lower level workspaces.

For purposes of best navigating through the system, assign names to only high-level workspaces. This is due to the way G2 presents workspace names through the Get Workspace* command. Using the G2 "Free Text" works nicely to indicate the name of a workspace. When searching directly for a workspace using the G2 Inspect, you may search for the ws-holder with the appropriate name, and then go to its subworkspace. This method provides for well organized tracking and an easy path to workspaces,

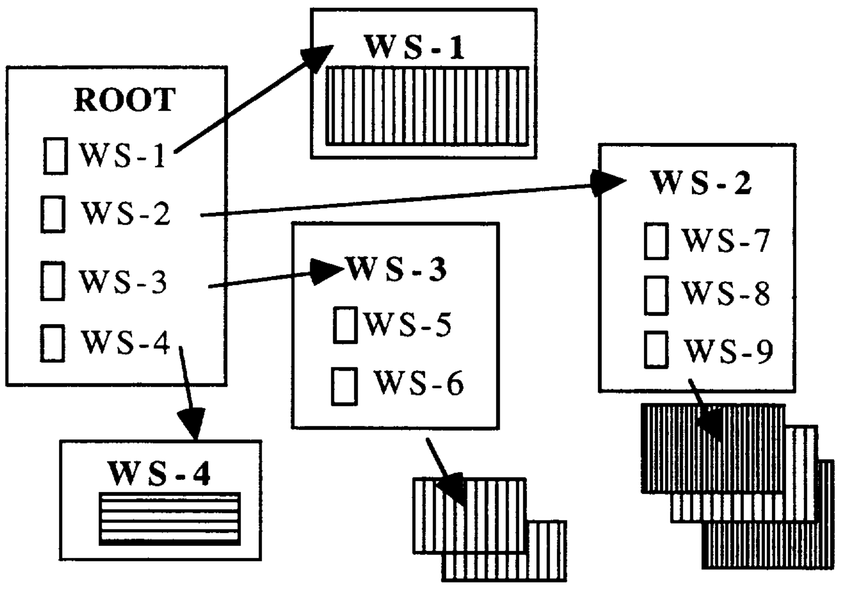


Figure 13. Conceptual illustration of workspace organization.

Now that the structure for workspace organization have been outlined, the details of segmenting work into different categories will be covered. Work breakdown begins with subsystems at the module level. Within modules there is a recommended breakdown as well. Below is a list of workspace categories and subcategories. The top level items might be the named workspaces listed on the highest level root workspace. Although an example and suggestions are provided, it is important to remember that the exact implementation will depend on the structure of your knowledge base.

- Definitions
  - Object Defs
  - Display Defs
  - MSID (data) Defs
- Objects
  - Object-type-1
  - Object-type-2
  - etc.
- User Interface
  - Screen-1
  - Screen-2
  - etc.
- Rules
  - Corrective (always on)
  - Monitoring (deactivatable)
    - State Transition
    - Commanding
    - Status

* G2 commands will be denoted with Helvetica font.

- Relation
  - Definitions
  - Rules
- Initialization
- Data Interface (MSID)
- Simulations (Testing)
- Documentation

The sample list should give you an idea of how to begin to break down your knowledge base; however, it is just one implementation tailored to DESSY. In addition, note that this breakdown does not follow object-oriented programming in the purest sense because the controlling rules, which parallel methods, have been separated from the objects they manipulate. In true object-oriented programming, objects and their methods are more closely tied. The approach taken to separate the rules and objects was primarily for organizational purposes. G2 rules can manipulate multiple objects, leading to complexities in classifying rules based on their associated objects. The authors found this approach to be effective and appropriate.

3.2 Object and Structure Design

Good object and structure design is crucial for a well designed expert system. The sections below discuss the various elements in an object-oriented system. They primarily center on factors in the object structure itself, but also include relations between objects and object groupings. Most of the discussion that follows reflects traditional object-oriented methodology; however, G2-specific examples will be included.

3.2.1 System vs. Subsystem Design

Any hardware system is likely made up of pieces and subpieces and should be modeled accordingly. This is the case for DESSY. For example, although there is a single MPM system, there are actually four individual MPMs making up the system—the shoulder, forward, mid, and aft MPM. DESSY has been implemented such that the MPM state diagram and rules apply to each individual MPM. In addition, an overview system state diagram has been developed which is dependent not on the telemetry data, but on appropriate changes of state in all individual MPM state diagrams. Thus when each of the four individual MPMs moves to a stowed state, the system level MPM reflects the stowed state. This approach allows one to capture two distinct levels of system monitoring—that of the individual pieces of the system and that of the system functioning as a whole. More detailed failure tracking is possible since more information is recorded about the individual MPMs. At the same time, a glance at system state (and status) as a whole is available.

3.2.2 Object Design

The first step in constructing objects is to develop the initial object structure and hierarchy, including the system and subsystem hierarchies. Object attributes should be defined, including state, or position, status, or operational health, and command, if commanding is associated with the operation of the item. In G2, object attributes might be other objects themselves. For instance, each MRL has two motors and four microswitch MSIDs. These six items are modeled in DESSY as both attributes of an MRL and objects themselves. In addition, the motor objects have their own attributes, which include additional MSID objects. This representation is useful because it closely matches the true system structure.

As you construct the object design, be prepared for system/subsystem cases that seem very similar, but are subtly different and therefore often need a different approach. To contrast the MRLs, the MPM system contains four MPMs; however, only the shoulder MPM contains motors. Thus the choice for MPMs was to place two motor objects within the MPM-system object, rather than in the shoulder MPM where the motors are physically located. This allowed all MPM objects to remain symmetric. It did, however, necessitate slightly different reasoning strategies when dealing with MPMs than when dealing with MRLs.

It is important to note that sometimes systems seem very similar, but their subtle differences are manifested when you attempt to model them as software objects. Figure 14 illustrates the DESSY design of the MPM system and MPMs.

A final caution: the developer must understand the subtleties of the systems before deciding on an object design. Be consistent, but don't get bogged down by system similarities that may not be that similar at the detail level. And if the need arises, be willing to change your object design to include newly discovered or understood information.

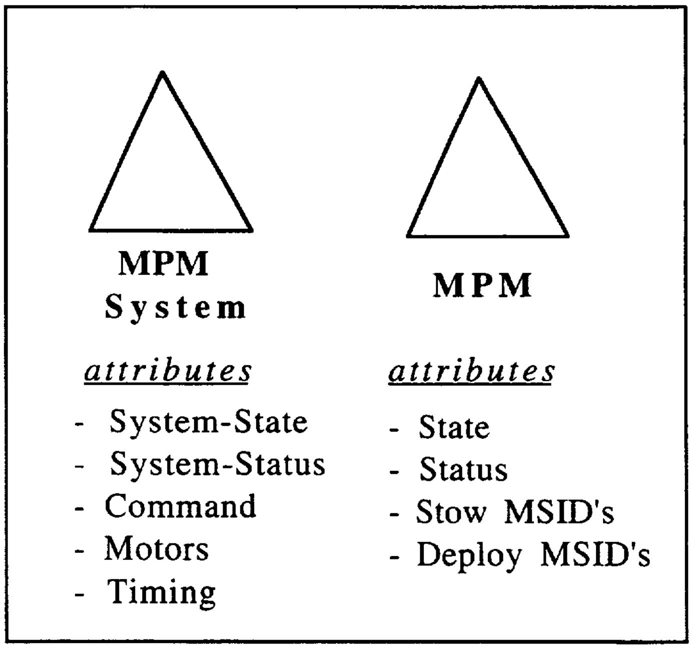


Figure 14. Example of the MPM system and MPM object definitions.

3.2.3 Class Hierarchy

As with conventional object-oriented programming, class hierarchy plays a powerful role in expert system design. G2 is very strong in its ability to reason over class hierarchies, which proved a great advantage throughout the knowledge base development process. A particularly good illustration is DESSY's MSID hierarchy. The highest level MSID definition for DESSY is MSID-def. It contains all the attributes of an MSID. Below it are, for example, end-effector-MSIDs and MPM/MRL-MSIDs. Within each of these groups are microswitch-MSIDs, opstat-MSIDs, and commanding-MSIDs. In addition there are power-MSIDs, enable-MSIDs, and switch-indication MSIDs. Within the MPM/MRL-microswitches there are stow-MSIDs, latch-MSIDs, deploy-MSIDs, and release-MSIDs. The breakdown of various levels could continue.

The point is that with only minor exception, all these MSID definitions are identical. Each lower level has inherited exactly the same attribute slots as its parents and other ancestors. Why then this elaborate breakdown? It reflects a desire to reason about a subclass at any of these levels. In G2 a single rule can either reason over all MSID-defs or reason only over all microswitch-MSIDs or only the stow-microswitch-MSIDs. This is a very powerful feature that allows manipulation of only the desired objects.

Throughout DESSY development, the class hierarchy continued to be refined to allow rules to reason over specific groups of objects. This led to powerful yet concise rules. Unfortunately, if you are not using G2, your tool may not have this strong object structure. If this is the case, you should study the capabilities of the tool you have and attempt to exploit its object reasoning capabilities to the fullest extent.

3.2.4 Inter-Object Structure (Relations)

There are times when you may wish to reason over a group of objects not associated through the class hierarchy. This may be because your tool does not allow the association, or it may simply be because the items were not related enough to be defined within the same hierarchy. G2 provides another powerful way to logically tie two (or more) objects together for reasoning purposes. G2 relations act as pointers between any G2 objects. The relation is associated with any two classes in the class hierarchy, and then instantiated between any two instances in those classes. The relation may be a one way pointer or it may be bidirectional. It may be one-to-one, one-to-many, or many-to-many.

An example of the use of a relation in DESSY is the association of an instance of the timer class with an instance of the MRL or MPM class. Four RMS timer objects were created to correspond to FWD-MRL, MID-MRL, AFT-MRL, and the system-MPM objects. A relation was defined which tied an RMS timer to an RMS object. Once the definition was set up, four rules were written to tie each timer to the appropriate object. Thus a single timer was associated with each MRL and the MPM system. A rule example is shown below.

"Initially conclude that FWD-MRL-TIMER *is-a-timer-of* Forward-MRL."

Creating this link then provided reasoning capabilities such as

"...start the timer that *is-a-timer-of* the MRL..."

where *is-a-timer-of* is the actual relation that references the tie. This capability allowed generic timer rules to be written, rather than creating specific rules for each timer/object pair.

A second example of the use of relations in DESSY is in tying the specific MSIDs to their displays. An MSID display class was defined. Its icon is a box that is light blue when data is inactive and dark blue when data becomes active. A bidirectional, one-to-one relation ties each MSID to its appropriate blue box. Thus it is possible to reference *the-msid-of* any blue box, or *the-msid-display-of* any MSID for which a display is defined. This implementation allowed separation of the user interface and data handling portions of the intelligent system. It made reasoning over the displays (for user interface purposes) very simple and concise.

In summary, relations are yet another extremely valuable tool that G2 provides to keep knowledge bases more concise and therefore more maintainable. Because the concept of relations is very similar to the concept of pointers, it is likely that whatever tool you are using, there will be some level of this capability provided. It is another feature that should be exploited to optimize the expert system development process.

3.2.5 System Variables and Parameters

As you construct objects and associated attributes, you will be defining variables and parameters for the expert system. In G2 the difference between variables and parameters is that variables may or may not have a value at any time, i.e., they may be "NULL." Parameters, on the other hand, always have some value, even if it is the symbol "none." This guide will follow G2's use of these terms. Other systems will likely have similar concepts.

There are three key attributes associated with the objects that model the hardware system. They are state, status, and command. These items should be made parameters so that they are always given a value, rather than being allowed to have a value of NULL. It may be that upon initialization, or at any time for that matter, the value of one of these attributes is unknown. Assigning the value unknown, or something semantically equivalent, to the attribute is an alternative if that is the case. Thus the expert system will directly conclude that the attribute's value is unknown, rather than the item being unknown (NULL) because data and/or other information was not available to make a conclusion. That is, the expert system knows that it doesn't know. This subtle difference in approach provides the developer with more control over expert system conclusions.

When is it advisable to use variables rather than parameters? In DESSY variables are used for the values of MSID data because it should be clear whether or not the data has been set or whether it is not available. In this case a value of NULL is quite informative because it is directly tied to the availability of the telemetry link. In addition, variables are used for items that need to time out. Along with the "NULL" value, G2 variables have validity intervals that allow the last set value to be good for a given time period. Variables are used with a time out or validity interval when the value is to be valid for only a specified number of seconds. For example, MSIDs have an attribute called "flicker." This value is set to true if the data value has flickered, or repeatedly turned on and off for a few seconds. Flicker is valid for about 4 seconds. Thus if the data has been noisy, a flicker value of true would prevent that data from being used in diagnostic rules. After the time period is up, flicker is automatically returned to false because the data is no longer unstable.

Whether object attributes are parameters, variables, or other objects with attributes of their own will depend on your specific design. Most likely the knowledge base will include all of these. You must decide which is most appropriate for each individual circumstance using the guidelines provided.

3.2.6 Grouping of Objects

The final area in the discussion of object design is the groupings of objects within the knowledge base. Because DESSY's display portion has been partitioned from the reasoning portion, most of the real objects are hidden, while their displays are presented to the user. G2 provides an icon slot for each object definition, which means that any object can have its display associated directly with that object, rather than having a separate display object and tying them together with relations. Your implementation will depend on the design goals of the expert system. If there is some possibility that the front end, or display, might eventually be replaced or upgraded with another tool, it is advisable to initially separate objects from display objects. If this is not a design criteria, however, you may wish to take advantage a G2's "icon-attribute" feature. Keep in mind that while separating display objects from the rest of the system will create a more segmented and cleaner system, these additional objects could lead to performance hits. The purpose of your system will be a major factor in this decision.

3.3 Rules

Once preliminary object design is underway, rule construction can begin. This section discusses several issues relevant to creating a maintainable and robust rule set. It starts with a discussion of the organization and types of rules within the knowledge base and then discusses some important factors in writing rules that are robust. Sections are provided on specific categories of rules you will build in the knowledge base. As you read this section, it is advisable to frequently refer to section 4, "Working with Real Time Data." It complements section 3.3 by explaining how to make your monitoring system robust enough to handle real data and its problems.

3.3.1 Organization and Classification

Rules should be well organized for the knowledge base to be maintainable and easily debugged. It is useful to break rules into categories aligning with workspace organization. The subsections below discuss additional issues that must be addressed during initial rule development.

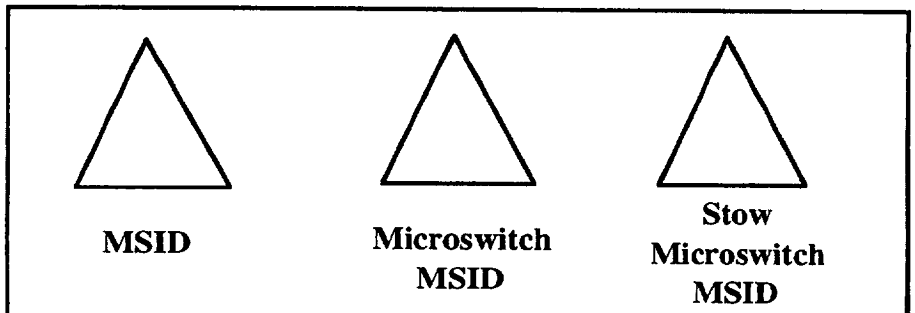
3.3.1.1 *Rules versus Procedures*

A rule-based system not only includes rules, but uses both rules and procedures as needed. G2 provides both rule and procedure writing capabilities, as do many expert system development tools. Although the flexibility of rule sets often makes them the best choice, the sequential nature of procedures makes them more useful at times. Don't hesitate to include procedures when they are appropriate. Although most DESSY logic was written using rules, there were times when procedures were either necessary or appropriate.

The primary use of procedures in DESSY related to timing and the activation and deactivation of sets of rules. Certain sets of rules apply only to particular situations and require continuous poling for a set of events. Because leaving these rules on continuously would bog the system down, we used procedures to turn on, or enable, these rules during the times they were needed, and then turn off, or disable. them once the situation had passed. An example is diagnostic timing rules that flag an MPM stow or deploy timing out. These rules are enabled when the stow/deploy begins. If the operation took longer than the allowed time, these diagnostic rules would fire and indicate the anomaly. Whether the operation is nominal or contains failures, the rules are disabled once the stow/deploy completes. Thus procedures were used in monitoring steps that called for control of diagnostic rules.

3.3.1.2 *Generic Rules*

Generic rules allow fewer rules covering more objects to be written. They reason over classes of objects, rather than individual object instances. G2 is particularly strong in this area due to its powerful object-oriented capabilities. As you develop the expert system, you will likely enhance the object hierarchy to allow for the creation of generic rules that cover precise subclasses of objects. Segments of generic rules, along with the class hierarchy the rules address, is shown in figure15. The Stow-Microswitch-MSID is a subclass of the Microswitch-MSID which is a subclass of MSID. Rules can be written for each level. Developing generic rules will lead to a smaller, more maintainable knowledge base.



For every MSID
If the MSID is...

For every Microswitch-MSID
If the Microswitch-MSID is...

For every Stow-Microswitch-MSID
If the Stow-Microswitch-MSID is...

Figure 15. Generic rules used with the class hierarchy.

3.3.1.3 *Rule Categories*

Rules should also be broken into appropriate categories according to their functionality. The categories DESSY includes are

- State—Rules that use data and existing state information to conclude a new state
- Status—Rules that use data and other scenario information, such as state or command, to conclude system health
- Command—Rules that use commanding and other relevant data, along with scenario information, to conclude a command, including a command of none
- Correction—Rules that watch for a precise data pattern, given a state or status that conflicts with that pattern, and invoke a corrective action to adjust the state or status
- Data Handling—Rules that manipulate raw data
- User Interface—Rules that manipulate a display item on the interface

Categorizing rules leads to more effective management of the knowledge base. It is also necessary to allow disabling of rule sets. Each of these rule categories is discussed in more detail in the following sections.

3.3.2 Making Rules Robust

The primary element that separates real-time data monitoring intelligent systems from off-line intelligent systems is their interaction with continuously changing data. In theory, data changes come in batches and rules are written based on a snapshot of data in the system at a given time. In reality, there are numerous "contaminations" of this idealistic scenario. These contaminations include noisy data and lags in data changes. Their effect is that the idealistic snapshot oriented rules simply don't work or work poorly. To overcome these problems, rules need to be built to be robust enough to function properly even in the presence of noisy or lagging data.

The issue of dealing with bad data has been addressed in section 4, Working with Real Time Data. This section was derived from a paper called "A Monitoring System with Tolerance for Real Time Data Problems." (Land, 1993) Other particularly relevant work is found in Chandrasenkaran and Punch, 1987, and Chandrasenkaran and Punch, 1988. Throughout the remainder of this section, the reader should reference section 4 for techniques in developing rules to withstand the problems of real data.

3.3.3 Initial State and Command Monitoring Rules

Now begins a discussion of building rules in G2. State and command rules will be discussed first, followed by status rules and finally correction rules. The subsections below explain the necessary steps in the rule writing process.

3.3.3.1 *Developing from State Diagrams*

State diagrams allow the development team to obtain a good understanding of the monitored processes and their states. Before you begin writing state and commanding rules, you will want to have completed state diagrams. Section 2.3.3.1 provides an in-depth explanation of the construction of state diagrams. Figure 16 is a state diagram for the MPM system. This example will be used in the following discussion of the construction of state and command rules.

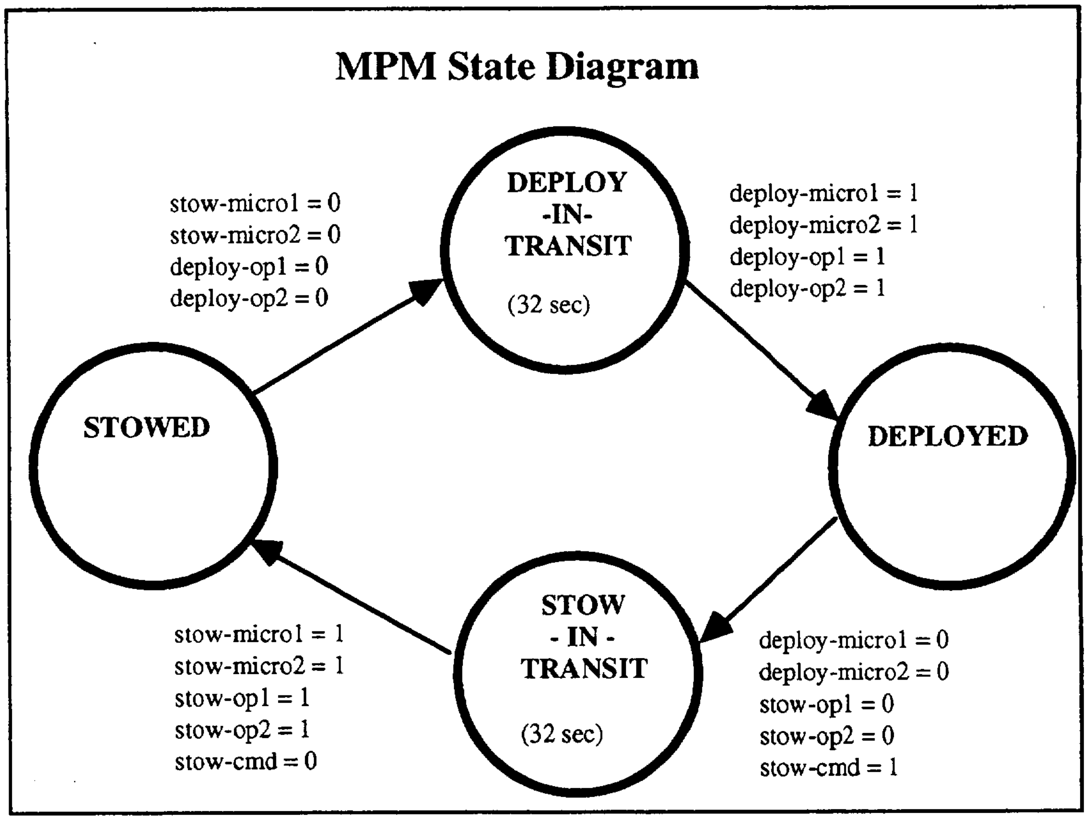


Figure 16. The manipulator positioning mechanism state diagram.

3.3.3.2 *Identification of Sequence Asymmetry*

Before developing the set of rules corresponding to a state diagram, identify procedural symmetry. Notice in the example in figure 16 that both the stowing and deploying operations use microswitch (micro1 and micro2) and opstat (op1 and op2) data. The stowing operation has a command (cmd) indicator, but there is no command indicator for deploying. Small discrepancies such as this are common in procedural symmetry. Because the data is slightly different, otherwise symmetric rules for monitoring these operations will be slightly different as well. Except for this discrepancy, however, the stowing and deploying operations are symmetric.

3.3.3.3 *Initial Rule Development*

Once state diagrams are defined, you are ready to create initial state and command rules. This discussion will continue with the example in figure 16 and construct rules for MPM stowing and deploying. A rule is created for each state transition and each command value. The rules will form a cycle to cover all transitions. The rule construction that follows will be for a nominal case scenario. It is important to realize that these rules are simplified versions of actual DESSY rules. They are a first cut, and working DESSY rules are in many cases much more complex. These complexities will be addressed in later sections. The rules have been consecutively numbered for clarity. Curly braces { } denote comments. Notice that there are both MPM rules and MPM-system rules (see figure 14) and that these two sets of rules interact with one another.

State Change Rules: Stowed --> Deploy-in-Transit

```
Rule #1
For each MPM
If the state of the MPM is Stowed
and stow-microswitch-1 = 0 {becomes inactive}
and stow-microswitch-2 = 0 {becomes inactive}
and the command of MPM-System is deploy {determined from opstats}
Then conclude that the state of the MPM is Deploy-in-Transit

Rule #2
If the state of MPM-System is Stowed
and Shld-MPM is Deploy-in-Transit
and Fwd-MPM is Deploy-in-Transit
and Mid-MPM is Deploy-in-Transit
and Aft-MPM is Deploy-in-Transit
Then conclude that the state of MPM-System is Deploy-in-Transit
and conclude that the status of MPM-timer is Start
```

State Change Rules: Deploy-in-Transit --> Deployed

```
Rule #3
For each MPM
If the state of the MPM is Deploy-in-Transit
and deploy-microswitch-1 = 1 {becomes active}
and deploy-microswitch-2 = 1 {becomes active}
and the command of MPM-System is none {determined from opstats}
Then conclude that the state of the MPM is Deployed

Rule #4
If the state of MPM-System is Deploy-in-Transit
and Shld-MPM is Deployed
and Fwd-MPM is Deployed
and Mid-MPM is Deployed
and Aft-MPM is Deployed
Then conclude that the state of MPM-System is Deployed
and conclude that the command of MPM-System is none
and conclude that the status of MPM-timer is Off
```

State Change Rules: Deployed --> Stow-in-Transit

```
Rule #5
For each MPM
If the state of the MPM is Deployed
and deploy-microswitch-1 = 0 {becomes inactive}
and deploy-microswitch-2 = 0 {becomes inactive}
and the command of MPM-System is stow {determined from opstats}
Then conclude that the state of the MPM is Stow-in-Transit

Rule #6
If the state of MPM-System is Deployed
and Shld-MPM is Stow-in-Transit
and Fwd-MPM is Stow-in-Transit
and Mid-MPM is Stow-in-Transit
and Aft-MPM is Stow-in-Transit
Then conclude that the state of MPM-System is Stow-in-Transit
and conclude that the status of MPM-timer is Start
```

State Change Rules: Stow-in-Transit --> Stowed

```
Rule #7
For each MPM
If the state of the MPM is Stow-in-Transit
and stow-microswitch-1 = 1 {becomes active}
and stow-microswitch-2 = 1 {becomes active}
and the command of MPM-System is none {determined from opstats}
Then conclude that the state of the MPM is Stowed

Rule #8
If the state of MPM-System is Stow-in-Transit
and Shld-MPM is Stowed
and Fwd-MPM is Stowed
and Mid-MPM is Stowed
and Aft-MPM is Stowed
Then conclude that the command of MPM-System is none
and conclude that the state of MPM-System is Stowed
and conclude that the status of MPM-timer is Off
```

Command Rule: Deploy

```
Rule #9
If the command of MPM-System is not deploy
and the state of the MPM-System is not deployed
and deploy-opstat-1 = 0 {becomes active}
and deploy-opstat-2 = 0 {becomes active}
Then conclude that the command of MPM-System is Deploy
```

Command Rule: Stow

```
Rule #10
If the command of MPM-System is not stow
and the state of MPM-System is not stowed
and stow-opstat-1 = 0 {becomes active}
and stow-opstat-2 = 0 {becomes active}
and stow-command-indicator = 1 {becomes active}
Then conclude that the command of MPM-System is Stow
```

Again, the above examples are simplified versions of the rules that are in DESSY. Although they are logically correct, they may not function well when faced with less than perfect data or if failures occur. The following sections elaborate on how the final rules are evolved.

3.3.3.4 *Considering Redundancy in Rule Development*

Once the initial state and command rules have been developed, the task of writing real, more robust rules can be addressed. The above rules work well in an idealistic scenario, but would not hold up under noisy data or under even a minor system anomaly. Section 4.3.2, "Context-Sensitive Bounded Pattern Recognition," is particularly relevant to the following discussion.

First note that each rule that references this type of data has two redundant microswitches and opstats. Because of this redundancy, there exists a flexibility in testing for the data change. An *or* statement, rather than an *and* statement, can be used for each of these data changes. This allows the system to monitor state or command changes even if a single piece of data fails or does not respond. This case may occur due to a single sensor failure, or it may result from bad data. Note that the *or* is not exclusive; thus either one or both microswitches becoming active causes the condition to be true.

Thus, for example, the state change Stowed-->Deploy-in-Transit rule becomes

```
Rule #11
For each MPM
If the State of the MPM is Stowed
and     ( stow-microswitch-1 = 0 {becomes inactive}
        or stow-microswitch-2 = 0 {becomes inactive} )
and the command of MPM-System is deploy {determined from opstats}
Then conclude that the state of the MPM is Deploy-in-Transit
```

The Deploy command rule becomes

```
Rule #12
If the command of MPM-System is not deploy
and the state of MPM-System is not deployed
and     ( deploy-opstat-1 = 0 {becomes active}
        or deploy-opstat-2 = 0 {becomes active} )
Then conclude that the command of MPM-System is Deploy
```

If the two redundant pieces of data were the only data available for the rule conclusion, we would likely require both to respond before a conclusion is made. However, since there is other information supporting each rule (such as state and current command), one of the redundant data pieces is enough to make the conclusion.

One additional piece of information can be included in these rules. Because these rules may fire based on one redundant sensor, the sensor should be checked to make sure it is indeed functional, that is, not previously failed. Suspect sensors will have been marked as questionable-on or questionable-off. Thus the Stow-to-Deploy-in-Transit rule changes as follows.

```
Rule #13
For each MPM
If the state of the MPM is Stowed
and (   ( stow-microswitch-1 = 0 {becomes inactive}
                and the status of stow-microswitch-1 is not questionable-off)
        or (stow-microswitch-2 = 0 {becomes inactive}
                and the status of stow-microswitch-2 is not questionable-off) )
and the command of MPM-System is deploy {determined from opstats}
Then conclude that the state of the MPM is Deploy-in-Transit
```

Writing rules in this manner allows the expert system to continue monitoring in the event that a single failure occurs or a single piece of data lags in its change; however, it also accounts for previous known sensor failures. Because other data indicates that the system is proceeding in its expected path, the expert system follows the new configuration.

3.3.4 Status Monitoring Rules

There are many types of status monitoring rules. The developer should begin status rule development by identifying failures of which detection is straightforward. Primary categories of status rules have been identified in Section 2.3.5, "Failure Space." Examples are provided for each of these areas.

3.3.4.1 *Transition Timing*

A set of rules will cover transition timing for each transition state in the state diagram. These include nominal transition, off-nominal successful transitions, and failed transitions. Examples for the MPM system deploy operation are provided below. Each rule uses a transit-time value which is determined by the timer. The state change rules in the previous section set the timer to *start* in rules indicating the beginning of a transition and to *off* in rules indicating a transition ending. Details of event timing are discussed later. For now assume that transit-time is properly set.

MPM Nominal Motor Deploy

```
Rule #14
Whenever the transit-time of MPM-System receives a value
and when the value of the transit-time of MPM-System > 0 seconds
and the value of the transit-time of MPM-System < 36 seconds
and the system-state of MPM-System is deployed
Then conclude that the deploy-system-status of MPM-System is nominal-deploy
```

MPM Nominal and Two Phase Motor Deploy

```
Rule #15
Whenever the transit-time of MPM-System receives a value
and when the value of the transit-time of MPM-System >= 36 seconds
and the value of the transit-time of MPM-System <= 40 seconds
and the system-state of MPM-System is deployed
Then conclude that the deploy-system-status of MPM-System is nominal-and-two-phase-
motor-deploy
```

MPM Single Motor Deploy

```
Rule #16
Whenever the transit-time of MPM-System receives a value
and when the value of the transit-time of MPM-System > 40 seconds
and the value of the transit-time of MPM-System <= 72 seconds
and the system-state of MPM-System is stowed
Then conclude that the stow-system-status of MPM-System is single-motor-stow
```

MPM Deploy Failed

```
Rule #17
for any mpm M
if the state of M is deploy-in-transit
and the system-state of MPM-System is deploy-in-transit
and the command of MPM-System is deploy {the motors are on}
and (the status of MPM-timer is on and the timer-count of mpm-timer > 72 seconds   {note that
transit-time is not set}
and deploy-microswitch-1 = 0 {is inactive}
and deploy-microswitch-2 = 0 {is inactive}
and the deploy-status of M is not failed-due-to-deploy-indicators-failure
Then conclude that the deploy-status of M is failed-due-to-deploy-indicators-failure
```

Note that these timing rules cover the full range of possible time values. The last rule is slightly different in that the failure is concluded at the individual MPM level rather than at the MPM system level. This is because if the failure is due to a pair of deploy microswitches failing to close, the single MPM with the anomaly can be identified. If all MPMs obtain this failure status, the failure is propagated to the MPM system with another rule.

3.3.4.2 *Data Questionable On/Off—Ramifications to the System*

The next type of status rules are those that identify individual pieces of data as questionable-on and questionable-off. These rules may be as simple as detecting a single piece of conflicting data in a set, or they may be complex and involve a series of events taking place.

The value of these rules is twofold. First the user is made aware of suspect sensors. DESSY flags suspect data by placing a yellow border around the sensor display. Second, a value of questionable-on/off can be used in other rules to predict off-nominal behavior, prevent a faulty sensor from causing a state monitoring rule to incorrectly fire, or provide further diagnostics.

Rule #18
For any stow-msid S
if S = 1 {active}
and the value of S as of 1 second ago = 1
and the command of MPM-System is deploy
and the system-state of MPM-System is deploy-in-transit
and the value of the system-state of MPM-System as of 2 seconds ago is deploy-in-transit
Then conclude that the status of S is questionable-on

Rule #18 flags the failure to lose a stow microswitch when a deploy is initiated. Recall that it takes the loss of only one microswitch to conclude deploy-in-transit for each MPM. Also note that a delay has been added to ensure that the microswitch is stuck on, rather than just slow to change. The microswitch is marked questionable-on.

Rule #19
For any stow-msid S
whenever S receives a value
and when S = 1
and the system-state of MPM-System is deployed
Then conclude that the status of S is *questionable-on*

Rule #19 flags a stow microswitch becoming active when the system is in a static state of deployed and no microswitch activity is expected. The microswitch is marked questionable-on.

Rule #20
For any stow-msid S
if S = 0
and the value of S as of 1 second ago = 0
and the value of S as of 2 seconds ago = 0
and the value of S as of 3 seconds ago = 0
and the system-state of MPM-System is stowed
and the system-state of MPM-System as of 1 second ago is stowed
and the system-state of MPM-System as of 2 seconds ago is stowed
and the system-state of MPM-System as of 3 seconds ago is stowed
Then conclude that the status of S is *questionable-off*

Rule #20 flags a stow microswitch becoming inactive when the system is in a static state of stowed and no microswitch activity is expected. The delay of 3 seconds is necessary to ensure that the system is indeed static and a deploy has not been initiated. Removing the delay would cause the rule to fire, inappropriately, at the beginning of a deploy procedure. The microswitch is marked questionable-off.

Rule #21
For any stow-msid S
whenever S receives a value
and when S = 0
and the system-state of MPM-System-Port is stowed
and the command of MPM-System-Port is none
and the power of MPM-System-Port is unpowered
Then conclude that the status of S is *questionable-off*

Rule #21 flags a stow microswitch becoming inactive when the system is in a static state of stowed and no microswitch activity is expected. It does not require a delay because other information is provided to ensure that a deploy has not been initiated. The microswitch is marked questionable-off.

The four rules above are examples of how data is marked as questionable-on and questionable-off in DESSY. Other expert systems will have rules which differ from those above; however, these types of rules should be written for any data monitoring expert system.

3.3.4.3 *Failure of Data to Change During an Expected Transition*

Throughout the monitoring process, there will be times of expected data change. This often occurs when the start of some event has been signaled by command data. When a command is detected (provided it is not a faulty indication), there is a known set of data that is expected to change. The rule below provides an example of this scenario where commanding initiates a deploy sequence, but the stow indicators are not lost as expected. In reality, because transitions take time to start, DESSY waits for 4 seconds of commanding to give the data time to change before it concludes it has failed to change. Also because the command rule caused the timer to initiate, it is reset to zero if the following rule fires.

```
Rule #22
If the state of any mpm M is stowed
and the deploy-status of M is not deploy-failed-to-initiate
and the command of MPM-System-Port is deploy
and the value of the command of MPM-System as of 1 second ago is deploy
and the value of the command of MPM-System as of 2 seconds ago is deploy
and the value of the command of MPM-System as of 3 seconds ago is deploy
and the value of the command of MPM-System as of 4 seconds ago is deploy
and the stow-microswitch-1 of M = 1
and the stow-microswitch-2 of M = 1
Then conclude that the deploy-status of M is deploy-failed-to-initiate
and conclude that the timer-count of mpm-timer = 0
```

3.3.4.4 *Multiple Data Values Active, Indicating Conflicting States*

Multiple active data that indicate conflicting states is another easily detected anomaly. Although these data will be marked *questionable-on*, and in fact the example shown is a repeat of Rule #19 from section 3.3.4.2, it is important to take this alternative view to ensure all conflicting data cases are identified.

```
Rule #23
For any stow-msid S
whenever S receives a value
and when S = 1
and the system-state of MPM-System is deployed
Then conclude that the status of S is questionable-on
```

In the above example, because the MPM-System is deployed, a stow microswitch becoming active clearly indicates conflicting data. Other data conflicts, such as an active deploy microswitch for a stowed MPM state, should also be identified by expert system rules.

3.3.5 Correction Rules

Correction rules are fundamental to the authors' philosophy of building expert systems. Their very existence is based on the premise that the expert system will eventually make a mistake or omission. This could be the result of faulty logic; however, it is more likely due to noisy data or periods where data drops out. In DESSY, correction rules have been used extensively and have prevented disaster on several occasions. Section 4.3.3, "Graceful Recovery," provides another view of this development technique.

Below are several examples of correction rules, each covering a different aspect of DESSY monitoring. The nature of each rule is described.

```
Rule #24
If the system-state of MPM-System is not deployed
and the system-state of MPM-System is not stow-in-transit
and the system-state of MPM-System is not deploy-in-transit
and the state of shld-mpm is deployed
and the state of fwd-mpm is deployed
and the state of mid-mpm is deployed
and the state of aft-mpm is deployed
Then conclude that the system-state of MPM-System is deployed
```

Rule #24 corrects the situation in which all individual MPMs have a state of deployed; however, the MPM-System does not have a state of deployed. Although this situation is unlikely, it might occur because of a data drop-out or because of a lack of machine performance, i.e., hardware problems.

A side effect of this rule is that it can be used to correctly initialize the expert system upon start-up. When the system is first turned on, the state is unknown. If the data indicates that the MPM is deployed, the individual MPMs will first obtain the state of deployed, and this rule in turn will set the MPM system to deployed. Rules like this should be written for all known stationary positions.

```
Rule #25
If the system-state of MPM-System is not deployed
and the system-state of MPM-System is not deploy-in-transit
and the system-state of MPM-System is not stow-in-transit
and (the count of each deploy-msid D such that (D = 1) > 6)
Then conclude that the system-state of MPM-System is deployed
```

Rule #25 is similar to rule #24, but takes a slightly different approach. It sets the MPM system to deploy, but uses the fact that there are more than six (out of a possible eight) deploy microswitches active. It may also be used to initialize the system. The rule allows for either (1) one of the four MPMs not being deployed, or (2) one or two or the four MPMs having a questionable-off deploy microswitch. Note that the rule depends on the expert's knowledge that in these cases the MPM system would still be considered deployed.

```
Rule #26
If the command of MPM-System is deploy
and (the deploy-sys-a-contact of MPM-Switch is open and the deploy-sys-b-contact of
    MPM-Switch is open )
Then conclude that the command of MPM-System is none
```

Rule #26 removes an MPM deploy command if both deploy-contacts, directly determined by opstat telemetry, are open or inactive. Whether or not the command was expected and/or desired, once the opstat command telemetry becomes inactive, this rule removes the command inference from the expert system.

```
Rule #27
For any MPM M
if the state of M is deploy-in-transit
and the status of M is deploy-failed-to-initiate
Then conclude that the status of M is operational
```

Rule #27 provides an example of resetting a failure conclusion that is no longer valid. In this scenario, the status of the MPM is deploy-failed-to-initiate; however, the state is deploy-in-transit, invalidating the status conclusion. This situation would occur if the deploy had initially failed, but later had successfully started. This rule resets the status to operational. The temporary failure can be recorded.

```
Rule #28
For any MSID M
if the status of M is questionable-on
and the value of M is inactive
Then conclude that the status of M is functional
```

```
Rule #29
For any MSID M
if the status of M is questionable-off
and the value of M is active
Then conclude that the status of M is functional
```

Rules #28 and #29 adjust the status of a questionable MSID, once it returns to normal. If an MSID is questionable-on, but becomes inactive, the questionable-on status is no longer valid. In the same way, if an MSID is questionable-off and becomes active, the status is updated to functional. Other corrective rules may adjust higher level status conclusions based on these MSID status updates. The anomalies are recorded.

Corrective rules are an extremely important part of a real-time expert system because they provide the system a graceful recovery mechanism if a faulty conclusion was made or if the failure observed was intermittent. Unlike monitoring rules, correction rules should always remain active since they are valid under all circumstances, even in the face of noisy data. Corrective rules might be thought of as the axioms of the expert system. For further information on writing rules dealing with noisy data, see Land, 1992, and Land, 1993.

3.4 Event Timing

Every expert system that monitors real-time operations will need a timing mechanism. In the DESSY project, a robust timer was developed that is available through the Control Center Library for Application Reuse and Exchange (CLARE, 1994) and can easily be implemented into other expert systems. This guide provides an overview of timer functionality and addresses writing rules to monitor events and provide timing-related diagnostics.

3.4.1 The DESSY Timer

The DESSY timer has four key controls or commands: it can be started, stopped, reset, and resumed. The functionality of timer is described below. Rules are usually used to send the appropriate command to the timer.

- Start—Sets the timer to 0 and then starts it
- Stop—Stops the timer at current value
- Reset—Sets the timer to 0
- Resume—Starts the timer without resetting it

The Start and Stop functions represent basic timer functionality. The Reset function allows a timer to be zeroed out, if desired, after its use. The Resume function was developed because on occasion a pause occurred in operations, and the paused time was not to be included in the total final time count. For example, MPM operations are timed only while MPM motors are running; i.e., the transition is actively occurring, rather than allow the clock to continue ticking even if the motors are temporarily stopped. One might also choose to time the entire operation, including pauses, with an additional timer.

The timer definition is provided in figure 17. The state of the timer may be on or off, and the status may be nominal or timed-out. The timer-count is any integer greater than or equal to zero. The timeout-limit is any integer greater than or equal to one. A set of timer control rules controls most of the timer mechanism. Only the command is set by external rules, and only the timeout-limit is preset for a given operation.

When the command receives a value of start or resume, the timer control rules change the state to *on*. A state of *on* in turn causes timer-count to begin increments. If the expert system is connected to real-time data, the timer-count is calculated from the incrementing mission elapsed time (MET) MSID. If the expert system is not connected to real-time data, the timer count is calculated from the incrementing system clock in the computer. For accuracy in timing real-time mission operations, it is advisable to compute time from the MET (or other downlisted time), rather than the computer's clock, which is subject to performance problems.

TIMER

Notes	OK
Name	DEMO-TIMER
Command	start
State	on
Status	nominal
Timer-Count	0
Timeout-Limit	32

Figure 17. DESSY timer definition.

As with start and resume, commands of stop or reset cause the timer control rules to change the state, but this time to *off*. Timer-count will no longer be incremented and, given that the command was stop, monitoring rules can then access timer-count based on the occurrence of the timer state change. Whenever timer-count exceeds the preset timeout-limit, the timer status is automatically changed to timed-out. This information can then be propagated to diagnostic rules. The full timer implementation, including detailed descriptions and control rules, can be found in the Control Center Library for Application Reuse and Exchange (CLARE, 1994).

3.4.2 Timing in DESSY

Below are example rules of the DESSY timer used in MPM operations. They include extensions of Rules #2 and #4 found in section 3.3.3.3. Keep in mind that your system and specific implementation may vary significantly.

Rule #30 sets the timer command to *start* based on the MPM-System being in-transit. The start-time is also recorded from OI-MET-TIME. Rule #31 sends a *stop* command to the timer based on the end of the transition. The stop-time is recorded from OI-MET-TIME and the transit-time is recorded from the timer-count. Of course, transit-time could also be derived from stop-time through start-time; these values should be equal.

```
Rule #30
If the state of MPM-System is Stowed
and Shld-MPM is Deploy-in-Transit
and Fwd-MPM is Deploy-in-Transit
and Mid-MPM is Deploy-in-Transit
and Aft-MPM is Deploy-in-Transit
Then conclude that the state of MPM-System is Deploy-in-Transit
and conclude that the command of MPM-Timer is Start
and conclude that the start-time of MPM-System = OI-MET-TIME

Rule #31
If the state of MPM-System is Deploy-in-Transit
and Shld-MPM is Deployed
and Fwd-MPM is Deployed
and Mid-MPM is Deployed
and Aft-MPM is Deployed
```

```
Then conclude that the state of MPM-System is Deployed
and conclude that the command of MPM-System is none
and conclude that the stop-time of MPM-System = OI-MET-TIME
and conclude that the transit-time of MPM-System = the timer-count
    of MPM-Timer
and conclude that the command of MPM-Timer is Stop
```

3.4.3 Timing Issues

Even with the timing mechanism provided, there are still several issues to be resolved for the expert system to successfully time events. The logic of timing events for your system should be carefully analyzed. Several questions which arose in DESSY event timing are submitted below, along with DESSY's solution. Keep in mind that your solutions might be quite different.

- Given a changing data set, do you start/stop the timer at the first piece of data changing, or do you watch for 2,3,... pieces?

  DESSY usually watches for more than a single piece of data before concluding anything.

- Which type of data should the timer command be based on: opstat (motor) data, command data, or microswitch data?

  DESSY uses opstat data and command data (when available) to determine command. Then the command inference and microswitch data together determine a state change.

- Should the period after the command is given and before position indicators are lost also be timed?

  DESSY uses only one timer for each event; however, there are diagnostic rules that detect if microswitches fail to be lost after a command has been given for x seconds. An additional timer for this purpose would be useful.

- If there is a pause in operations, do you time that? do you include it?

  DESSY pauses the timer when there is a pause in operations. If it is desired to time pauses also, an additional timer could be added.

- If operations stop, and are then reversed, what should the timer indicate?

  DESSY had a surprise the first time this behavior was observed! This may or may not be possible in your system. Here we record the "forward" time, and then restart the timer. Other non-timer rules are needed to detect this behavior. This timer behavior should really depend on the system and what the expert would like to see.

- Are the failure detection (time-out) timers the same objects as the operations timers?

  DESSY had only one timer per operation. Sets of diagnostic rules were used, with a rule corresponding to each status-related time indication. However, an alternative approach would be to create a timer with several time-out limits, each relating to a different timing anomaly category.

In conclusion, keep in mind the many factors that must be considered when developing timing rules. Writing event timing rules requires close work with the expert. You will want to ask which data is used by the expert when manually timing events in the current system. Once this is done, you will have to decide which rules (commanding, state transition) contain timer command control, and how detailed the diagnostics should be for event timing. The authors encourage you to investigate CLARE to see the latest information about the DESSY timer.

3.5 Quick Test Buttons and Displays

Throughout rule development, testing should be done. G2 provides several standard button types which make simple testing easy. In addition, simulations can be written using G2 procedures. These test procedures should include both nominal operations and single data point failure operations. Below are some example test buttons and displays that G2 provides. We focus on how they were used in the developer's interface rather than the end user's interface. Other software development environments will likely have similar items.

- Action Buttons—Useful for the initiation of some action. Sets of action buttons can change the value of any variable or parameter. They were found to be the fastest method of implementing very simple tests. Action buttons can also be used to launch procedures. The were used extensively in DESSY for starting the simulation procedures.

- Radio Buttons—Useful for toggling variables among specified sets of values. Although not as quick as action buttons, radio buttons provide a very effective way for rapid testing. They are mutually exclusive and provide for setting a value upon start-up.

- Check Boxes—Useful for setting true/false values. They are similar to radio buttons, but a single check box acts in an on/off manner to set a value.

- Slide Rules—Useful for selecting a value from a range for a variable or parameter. Slide rules also behave as display devices in G2, changing value when their associated variable or parameter changes value through another source.

- Procedures—Particularly useful for creating simulation scenarios. Procedures were used throughout DESSY development to create test scenarios. They may contain *wait* statements which provide necessary delays for a scenario, allowing data values to be directly set from the test procedures.

- Readout Tables—Useful for displaying a value. They are a very quick mechanism for displaying any G2 variable or parameter value and provide immediate verification that the values are responding as expected. Although the look of readout tables can not be modified, they are quite adequate for supplying the information the developer needs to see.

- User-Defined Menu Choices—Very useful for associating a set of menu choices with any display item. A primary use of User-Defined Menu Choices in DESSY was to associate menu choices of "active" and "inactive" with each MSID display. Thus, the developer could easily set a data value for testing purposes at any time during development. In addition, a user-defined menu choice might call a procedure that changes a set of data values or starts a simulation. Although the G2 menu choices were slightly difficult to use and could be improved, they were valuable because they made data values accessible. Note that G2 does provide the capability to remove this feature from the end user, while preserving it for a developer's further testing.

3.6 Setting Up for Real Time

Setting up the expert system to monitor real-time data is a crucial part of expert system development. Up to this point the system has been self-contained, and testing procedures have been handcrafted to produce desired results. Real-time data (live or recorded) will provide the system with a new plateau of testing possibilities, and likely a new level of challenges to go with it.

The first step in setting up for real-time data is to identify the data source(s). DESSY originally used the Real Time Data Systems (RTDS) data source; however, this data source has recently been replaced by the Information Sharing Protocol (ISP) data source. The DESSY team's experiences with setting up for the RTDS data are discussed with a focus on the expert system's behavior towards real time data. See CLARE for more information about DESSY's data interface.

3.6.1 Setting Up the Knowledge Base

The G2 tool provides an interface object and interface code to tie it to real-time data. The DESSY team was fortunate to obtain both RTDS and ISP data interface objects from their respective organizations rather than having to develop one. Creating the real-time interface can be a difficult task. When possible, developers should obtain a predeveloped interface and avoid writing (and likely duplicating) this code. A data interface should be a standard element in any real-time expert system library. For purposes of standardization and time savings, it should be developed only once, by a team that thoroughly understands the data source.

In addition to setting up the data interface, the individual sensors (i.e., MSIDs) in your knowledge base must be configured to use real-time data. Each instance of a sensor object must have its table filled in appropriately. This includes the MSID identification number, the telemetry server's name, and other information about that sensor. Again, the DESSY team obtained the sensor object template from the authors of RTDS. Figure 18 shows a sensor (MSID) object. See CLARE to find information on the latest MSID definitions for the currently used real-time data interface.

SHLD-MPM-STOW-MICROSWITCH-1

Options	do forward chain, breadth first backward chain
Notes	OK
Name	SHLD-MPM-STOW-MICROSWITCH-1,
Data type	quantity
Initial value	none
Last recorded value	0
History keeping spec	keep history w/ max age of data pts = 5 sec
Validity interval	indefinite
Data server	telemetry-server
Default update interval	13 weeks
Gsi interface name	rtds-data-server
Gsi variable status	0
Category	msid
Param name	v54x0820e
Param type	cal
Sample number	1
Cvt type	none
Status	functional
Flicker	false
Microswitch value	open

Figure 18. An MSID sensor object.

The final component in setting up the expert system to use real-time data is to obtain the data source itself. This involves identifying the data source, verifying that the data stream is available to your machine, and understanding how to connect to and control the real-time test data. The three sources of real-time data the DESSY project used were live mission data, live Shuttle Mission Simulator (SMS) data, and recorded data from either of these sources. Although mission and simulation data are valuable because of their surprise element, prerecorded playback data is the most useful because many failures can be recorded on a single tape (file) and these failures can be played back at will as the system is corrected and refined. Section 6, "Testing the System," will provide an explanation on testing with each of these data sources.

Once the knowledge base and data source are ready, the real-time data testing can began. Setting up the real-time data source is not a trivial task, and you should get help from the data host organization or from other available help organizations. Details on setting up the data source are beyond the scope of this guide. The advice to developers is to get help from someone who is experienced in this area. Reuse of other developers' codes and knowledge will save your development team valuable resources.

3.6.2 Using Real-Time Data

Once the real-time data source and sensor objects have been set up and the data connection has been established, real-time testing can begin. During this stage of project development, many unknowns will likely surface and the iterative nature of application development will become apparent. Rules which performed ideally given ideal data scenarios may now malfunction due to the inpreciseness of real data. Be prepared for design changes during this step. It is likely you will have to retrace many of the steps in section 3, and you may even need to revise your understanding of the system and consult section 2 again. However, following the development steps outlined in these sections should minimize knowledge base changes.

As you begin using real-time data, keep in mind that Section 6, "Testing the System," provides insight into the testing process. As with any type of testing, nominal cases should be checked first to verify the system's basic functionality, and then more complex cases, i.e., failure cases, can be addressed. It may turn out that even in nominal cases the behavior of the data is not ideal. You will need to make sure your system is robust enough to handle the normal data discrepancies that occur in the operations it monitors.

3.7 Setting Up for End Users

Setting up for end users involves providing the finishing touches that will allow someone unfamiliar with the expert system to readily use it. This includes instructions on starting up and shutting down the system. If you are using Motif, you will likely want to allow users to launch the system from a Menu rather than having them use a command line prompt. You will need to make sure the users' path has been correctly set for both the G2 (or other) applications and for your expert system and make sure the display variables are correctly set (assuming you are working in a Unix environment). This may involve changing the environment on the users' workstation. Since the workstation will likely be running other applications as well, coordination with other workstation users is necessary.

To help users feel comfortable with the system, a basic instruction sheet should be provided with an overview of software use. For DESSY color cue cards were created for the MPM/MRL and end effector systems. A black and white copy of these cue cards is provided in Appendix B of this guide. This type of overview sheet is common for software applications, and has been useful for DESSY users.

In addition to the cue card, a DESSY tutorial was written in G2 "on top of" the DESSY system. The tutorial provides detailed explanations of each item on the DESSY screen. The tutorial is discussed in section 7. It is an important part of preparing the end users to work with the expert system.

In conclusion, the most important element in setting up for end users is to ensure that it is as simple as possible for users to understand and use your system. Many (possibly most) users do not have significant software experience and may in fact be uncomfortable with software. You must encourage them to use the system by making it not only a beneficial, but also a simple task.

Section 4
Working with Real-Time Data

A major challenge for intelligent monitoring systems is to handle real-time data robustly. Section 4 addresses problems in and solutions to handling data that is often unpredictable and unreliable. The principal real-time telemetry data problems that are encountered in the space operations domain include (1) missing data, (2) erratic or noisy data and (3) data lags and irregularities during state transitions, commanding, and other operational events. A combination of methods evolved throughout DESSY development that address these areas include rule disabling, use of context and expectations to make assessments that tolerate some bad data, and graceful recovery through system correction when reliable data returns.

When working with real-time data, one must expect the unexpected in data problems. In spite of filtering and other techniques to keep bad data out of the system, some will always get through. When it does, the only choice is to design for recovery when things return to normal. Real-time data monitoring systems that reside in high-risk environments, such as NASA's Mission Control Center, must be built to be robust enough to handle bad and missing data. Although the specifics of these techniques will differ from one project to another, the guidelines presented here provide a good set of ground rules from which a robust system can be built for real-time monitoring.

Section 4.1 covers "Related Work," which gives other approaches to addressing the real-time data challenges. References are provided. Section 4.2, "Types of Data Problems," explains the cause and effects of each of the data problems. Section 4.3, "Data Handling Methods," elaborates on the techniques used in DESSY to build a robust system that could withstand these data problems. Examples of rules and techniques used in DESSY are provided throughout.

4.1 Related Work

Conventional approaches such as data filtering or data validation may be used to deal with some unreliable data, but they are often insufficient to eliminate all bad data, and can even introduce data problems. Even if it were possible to filter out the bad data, the use of unfiltered data is desirable because it gives the human monitor a better understanding of the telemetry. Human experts routinely tolerate significant amounts of bad data in monitoring and diagnosis, whether or not it has been preprocessed in these standard ways.

Statistical approaches to data problems in diagnostic systems (Cooper, 1990, Paasch and Agogino, 1992, and Wellman, 1990) use sampling and statistical testing techniques or probability estimates. Sampling and testing approaches, which are forms of preprocessing, may require sample sizes or sensor redundancies that are not available. Additionally, such approaches are somewhat unreliable and cannot eliminate all data problems. Probability estimation approaches suffer from tractability problems in cases where the possible failure space is high, and certainty estimates may be difficult to obtain from the domain expert. Statistical approaches have proven to be useful in cases that can accommodate their limitations; however, they remain incomplete because they do not provide for recovery from wrong conclusions.

Another approach to problem data is to concentrate on diagnosing sensor failures (Scarl, 1987). Once a sensor failure is diagnosed, the faulty data can be eliminated until the sensor is fixed. This approach requires sensor redundancy or testing to identify the failures, and shifts the focus from operations monitoring to sensor diagnosis. DESSY does include a limited amount of faulty sensor identification; however, the majority of the data problems in telemetry monitoring systems are not due to sensor failures, but are temporary data noise. Unlike sensor failures, these data problems are both transient and expected (Shontz, et. al, 1992). While sensor diagnosis may be prudent and useful, it does not address the telemetry data problems.

A knowledge-based approach to dealing with bad data is to rely on diagnostic expectations and the preponderance of the data (Chandrasekaran and Punch, 1987). DESSY uses such an approach, basing conclusions on operational context and expected data patterns. Such an approach provides graceful degradation in the face of bad data (Hayes-Roth, et al, 1983), permitting reasonable interim assessments while eliminating most false conclusions. However, since bad data will inevitably enter the system, a recovery approach is needed. Recovery should be more graceful than a system restart. A system should be able to recover by incrementally reinterpreting data as it is gathered (DeCoste, 1991). DESSY uses a technique its developers have deemed graceful recovery. New data is reevaluated as it enters the system, and adjustments are made to expert system conclusions based on that data.

The data handling techniques implemented in DESSY have repeatedly proven effective in maintaining robust expert system operation while tolerating bad data. Disabling rules when data is unreliable mirrors the human process of ignoring bad data. The pattern recognition approach, similar to that presented by Chandrasekaran and Punch (1987), allows tolerance in conclusions by considering context and expectations and by marking data as questionable if discrepancies occur. The DESSY approach differs from Chandrasekaran; however, in that if the expert system makes a mistake based on bad data, a graceful recovery mechanism is used. Although each of the techniques presented has merit by itself, it is their combination that most reflects the human monitoring process leading to a system with a human-like level of tolerance to data problems.

4.2 Types of Data Problems

The DESSY approach to dealing with unreliable telemetry data is distinctive because of its objective of achieving human-like tolerance to bad data. Data problems such as loss of data, erratic or noisy data, and data lags and irregularities are tolerated by monitoring data quality, using diagnostic expectations, and correcting wrong conclusions when good data returns. The following sections explain each type of data problem. Section 4.3 provides their DESSY solutions.

4.2.1 Loss of Data

The most common and well understood data problem is a loss of data. Data loss usually takes the form of a complete loss of signal (LOS) and may be either unexpected or expected, i.e., when the Shuttle enters the Zone of Exclusion (ZOE) where there is no telemetry downlink. Unexpected data loss may occur as a complete or partial loss of data, usually due to ground processing or computer hardware problems.

Although LOS seems like a simple phenomenon to account for, hardware implementation of telemetry processors can complicate the situation. Depending upon how a particular telemetry processor handles periods of LOS, the monitoring system may receive an inactive state for all data values, no data values at all, or a static frame of the last data values. In the case of the DESSY data source, the Real-Time Data Systems (RTDS), the last 4-5 seconds of data is repeatedly replayed until the signal returns. Yet another possibility is that nonsense data is received during periods of LOS. The first step in dealing with loss of data due to signal loss is to find out the form of data that will be received during this time.

The DESSY data source provided a data quality measurement from the telemetry processor from which LOS could be concluded. OI-Quality indicates the telemetry processor's assessment of data quality for each data frame. However, OI-Quality does not always reliably reflect true data quality. There is often a lag between the data quality drop and OI-Quality's reflection of this drop. Even when the lag does not appear, the low quality indication often occurs in the same data frame that includes bad or missing data, making it impossible to filter the data or to alert the system of its presence. Inevitably, bad data due to an LOS periodically enters the system. Methods such as rule disabling and graceful recovery address this problem.

4.2.2 Erratic Data

Erratic data is unstable and does not meet expected behavior for a given operational context. Erratic data may occur at any time, but is most likely to be seen immediately before or after LOS or at the time of state transitions. Erratic data is characterized by frequent flipping of bits (in the binary case) in a particular data set. Bit flipping occurs when a data value toggles or flips between values of 0 and 1. This signature may also result from intermittent sensor failures, and that possibility should not be ruled out. For large sets of data, however, it is more likely that any bit flipping is due to bad telemetry data rather than bad sensors.

For DESSY, periods surrounding an expected LOS are a common time for erratic data to appear. As the Space Shuttle moves in or out of a satellite's range, the telemetry link has a period of degradation, during which the OI-Quality has not yet dropped. The result is often a significant amount of bad data entering DESSY. Because there has been no previous low quality indication, DESSY has no indication of the degraded data. The second situation in which erratic data often occurs is near the beginning and end of a state transition. This type of erratic data is discussed in the next section. Context-sensitive bounded pattern recognition and graceful recovery can be used to deal with both types of problems.

4.2.3 Data Lags and Irregularities During Operational Events

The final type of data problem results from unexpected data activity when data is expected to change because of an operational event such as a state transition or commanding. Data that is expected to change may flicker or lag before reaching a new stable state. Given a set of data that is expected to change at transition time T, subsets may flicker or lag, causing the data transition to occur over some delta time t. Typically delta t is 1 to 3 seconds. Figure 19 depicts five examples of data transitions from low to high values, including a normal data transition and four anomalous cases. The delta time for this data set, 2 seconds, is the time it took for every piece of data in the set to change to its new expected value.

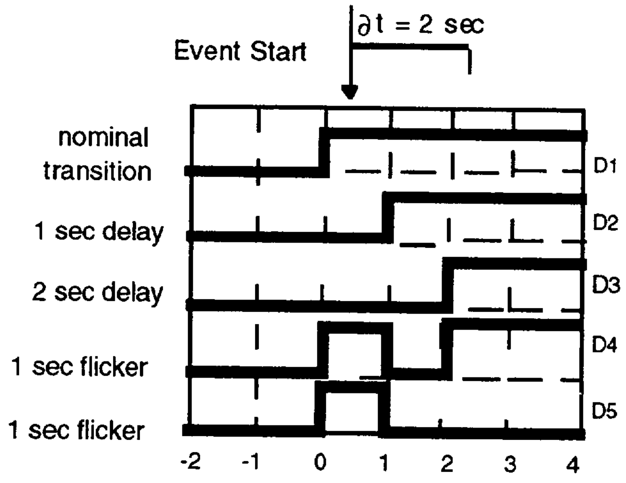


Figure 19. Examples of data lags and irregularities.

Data lags and irregularities during operational events may be caused by telemetry noise or by the physical properties of the hardware being monitored. In either case, there is no smooth transition. Failing to include this behavior in the system's monitoring rules will often lead to erroneous conclusions.

4.3 Data Handling Methods

Now that the types of data problems experienced by real-time monitoring systems have been explained, the focus will turn to solving those problems. Table 3 lists the data problems with their applicable solutions from the DESSY approach. Each method works independently to prevent or correct erroneous expert system conclusions resulting from bad or missing data, but it is their combination that ensures a robust program capable of lengthy periods of uninterrupted use in operations. The techniques are discussed in the following sections.

Table 3. Data Problems and Solutions

Data Problem	Solution Methods
loss of data	rule disabling, graceful recovery
erratic data	pattern recognition, graceful recovery
data lags and irregularities during operational events	pattern recognition, graceful recovery

4.3.1 Rule Disabling

The most straightforward method of dealing with data of uncertain quality is to ignore it by not responding to changes in that data. In any system there will be times when data quality is so low that the data should not be used at all. The intelligent system should have the capability to ignore data when it is unreliable, by a method such as disabling rule sets. Modularity within the software (see section 3.1, "Organization of the Knowledge Base") separates rules for data monitoring, state transition detection, failure diagnosis, expert system control, and user interface. This partitioning allows the enabling or disabling of appropriate groups of rules when necessary. Certain diagnostic and state transition rule sets are disabled in DESSY when the value of OI-Quality is any number other than 100. When quality returns, the rules are again enabled.

Although this tactic seems simple enough, the quality indicator and its corresponding data are in the same time frame, making immediate disabling impossible. Also, as the Shuttle enters or leaves the ZOE, the telemetry link deteriorates, making OI-Quality itself unreliable at that time. However, even though erroneous data may have already entered the system, it is still desirable to disable rules to minimize further faulty conclusions.

Disabling rules is the simplest of the techniques used in dealing with real-time data problems. Unfortunately, it effectively eliminates the usefulness of the expert system during the time the rules are disabled, retaining only its function as a raw data monitoring source. For this reason, the technique is only used when absolutely necessary—when it is certain that quality is low and data is unreliable.

To supplement automatic rule disabling, the DESSY user may also turn off the expert system portion of DESSY at any time, leaving DESSY to act as only a data monitor. Actually, this turning off merely grays out the parts of the DESSY display that present expert system conclusions, allowing DESSY to continue to work in the background. Section 5.6.3, "Control of the Expert System," elaborates on this concept. Even if DESSY has made incorrect conclusions and the user has grayed out the expert system part of the DESSY screen, the built in self-corrective rules should eventually lead to recovery. The intent is that even if DESSY has been turned off due to erroneous expert system conclusions, it will recover by itself, and the user will once again be able to use the expert system part of DESSY. Nonetheless, this feature gives the user the opportunity to override the expert system at the level of the user interface (Malin and Schreckenghost, 1991).

4.3.2 Context-Sensitive Bounded Pattern Recognition

The second technique implemented to ensure DESSY's robustness deals with characteristics of the sets of data that DESSY uses to detect events and identify failures. Because of problems with data lags and irregularities, the expert system often has insufficient or even erroneous evidence from the data set upon which it can determine the occurrence of an event. Also, because of the nature of space operations, there is often an insufficient amount of sensor data available, making event determination even more difficult.

For example, if the event is an MPM stow, there are only two sensors to indicate the stowed state once the MPM has reached its new stowed position. This data set is insufficient to determine the stowed state. DESSY requires that all rules except recovery rules tolerate a single data failure in any set of data considered. If one of these two pieces of data was active and the other inactive due to a data problem, the state would be inconclusive. The data must therefore be supplemented with additional information.

When appropriate, context such as the system state or the detection of a prior event is included. When context is considered in a rule, a context variable (CV) is used. Context variables are simply known pieces of relevant information. They may be a combination of data and/or information and may be either G2 variables or parameters. Use of CVs supplements the data set, and some rules use only CVs. Table 4 gives an example of a DESSY rule using two pieces of data and two CVs. CV use also simplified the verification process.

Table 4. Rule With Two Pieces of Data and Two Context Variables

DESSY Rule: state transition	Data/CV set
If the state of any MPM is *stow-in-transit*	CV
and (the sys1-stow-microswitch = *1*	data
or the sys2-stow-microswitch =*1*)	data
and the command of MPM-SYSTEM is *stow*	CV
then conclude that the state of the MPM is *stowed*	conclusion

Following are the general guidelines developed in determining necessary data sets and context variables for DESSY rules. The lower limit or minimum requirements needed to make reliable conclusions is addressed, followed by a brief discussion on the upper limit or maximum data set recommended. Establishment of a lower limit is necessary because enough data must be observed to correctly monitor an event, given that some predetermined amount or percentage of the data set being considered is bad. An upper limit is established because of the impact the data set size has on computer hardware performance. An example from DESSY is provided.

4.3.2.1 *The Lower Limit*

Given an operational event, there exists a set of data S that the intelligent system directly monitors to determine when the event occurs. In addition, there exists a second, usually larger, set S', a superset of S, that makes up the context in which the event will occur. Figure 20 depicts this concept. The set S' indicates state, status, and any other operational context of the system being monitored and includes both data and CVs. Humans may verbally testify to monitoring S when watching for an event to occur, but in reality it is the entire context set, S', that the human observes. DESSY attempts this behavior.

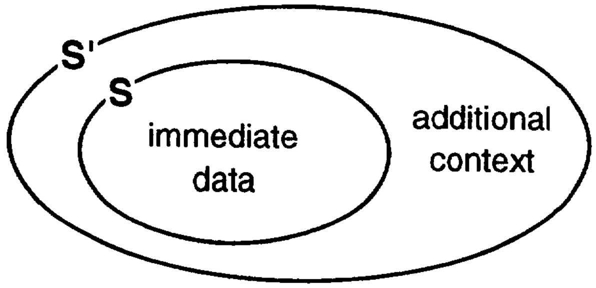


Figure 20. Sets of immediate data and additional context.

The CVs from S' in the example of Table 4, which concludes an MPM stow operation, are the current MPM state and MPM command. Thus we have the following sets.

S_{stowed} = {sys1-stow-microswitch, sys2-stow-microswitch}

S'_{stowed} = {sys1-stow-microswitch, sys2-stow-microswitch, current state, command}

The resulting concept is shown in figure 21.

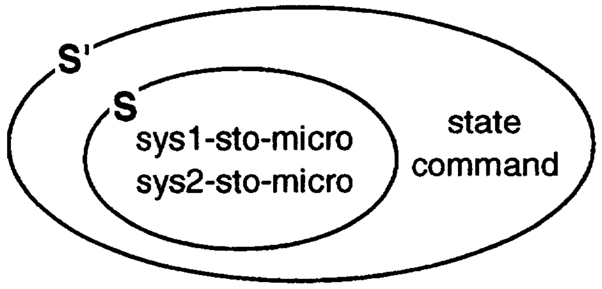


Figure 21. Example of S and S' for immediate data and additional context.

If the set S were the only data considered, a constraint that both data elements be active would need to be imposed to eliminate ambiguity and ensure a stowed state had been reached; however, this does not meet DESSY's requirement to continue to monitor given a single failure. Because context is considered in the extended set S', the constraints to require only one of the two microswitches to be active in determining the stowed state can be relaxed. Because the set size of two for S is insufficient for determining the event given DESSY's requirements, S' offers a more plausible solution.

Even in the case where three pieces of telemetry are available, the data is probably insufficient. Although in actual operations, a double hardware failure must occur to lose two of the three sensors, in a situation with noisy data, it is possible that two of these three may erroneously become turned on. In this case, a larger data set including CVs is desirable. CVs such as state and command in the example above can be used to obtain the minimum information set by imposing physical constraints or providing the current system configuration. CVs limit the scope of a rule, and thus limit the chance that it will fire incorrectly given it has received bad data.

4.3.2.2 *The Upper Limit*

In real-time operations, every piece of data observed has an associated performance cost. Thus there are limits on the amount of data DESSY is allowed to inspect in most rules. The best way to implement this is through context variables which hold summaries of data values and are usually already stored in DESSY for other purposes. Use of CVs provides constraints limiting the size of inspected data sets. DESSY uses the upper bound per rule of four pieces of telemetry data or two to three pieces of data given a CV. The real-time expert system should usually not be overloaded with a larger set, although there will be exceptions in safety critical or unreliable areas.

4.3.3 Graceful Recovery

Although some expert systems attempt graceful degradation in the face of trouble, DESSY has extended this concept to one of graceful recovery. If the system makes faulty conclusions because of bad data, a set of corrective rules act to restore the expert system once good data returns. This includes correction of individual data, state, and status. The system does not have to be restarted by the user because the corrections automatically restore offending parts of the knowledge base.

Corrective rules are similar to other system monitoring rules, except that they do not allow for any inconsistencies in the data sets they observe when making conclusions; i.e., every piece of data in the set must be exactly correct. Additionally, corrective rules are written only for the cases that it can be determined with certainty that the system is in a particular configuration. These rules, therefore, can be thought of as the axioms of the expert system. They can be as simple as determining that a single piece of data is reliable again because it returned to a legitimate value after it had previously been deemed unreliable. A more sophisticated example is the reevaluation of system state when all microswitch data for a new state becomes active.

Examples of corrective rules are shown below. The first example shows the reevaluation of the status of a single microswitch. If the microswitch status is *questionable-on* because the microswitch is inappropriately active, and then the microswitch value returns to inactive (0), the status is set to *functional*. The second example reevaluates the state of MPMs. If the state is not stowed, but both stow microswitch indicators become active (1), then the MPM state is set to stowed. Section 3.3.5, "Correction Rules," gives detailed instructions on the construction of corrective rules and provides further examples.

Corrective Rule: Microswitch Status

```
If the status of sys1-stow-microswitch is questionable-on
and the sys1-stow-microswitch = 0
then conclude that the status of sys1-stow-microswitch is functional
```

Corrective Rule: MPM State

```
If the state of any MPM is not stowed
and the sys1-stow-microswitch = 1
and the sys2-stow-microswitch = 1
then conclude that the state of the MPM is stowed
```

These correction features have not only been useful, but also have good side effects. Complementing the correction rules, initialization rules allow DESSY to be started during any stable MPM/MRL configuration and to be initialized to the proper states and statuses. The MPM state rule shown would initialize MPM state to stowed if DESSY were started with the MPM in the stowed position. If DESSY is started during transition periods, once the transition is complete, DESSY can initialize itself. This has been an important and even necessary DESSY feature.

Section 5
Evolving the System through the User Interface

The user interface is the window into the monitoring application and is therefore a crucial part of the overall system design. It is separate from, yet intricately linked to, the expert system, and should be recognized as one of the many components that make up the complete system. Like the expert system part of the application, the user interface is developed iteratively. This section presents some guidelines for creating this vital application component, starting with "Understanding the User Interface" and "Preliminary User Interface Design" (sections 5.1 and 5.2). It then discusses "Evolution of the User Interface" (section 5.3), and provides an application display example in section 5.4, "The DESSY User Interface." Section 5.5 discusses "User Input and Control," or how the user interacts with the intelligent system during operations. Section 5.6, "Expert System Management.," covers management of data problems and problems with the expert system itself. Finally, section 5.7, "Developer's Interface," addresses those displays which assist the developer throughout the life of the system. (After all, as a developer of these systems, you are a software user too.)

5.1 Understanding the User Interface

Because the user interface is such an important aspect of the monitoring application, its design should be carefully thought out before development begins. The user interface is more than just a screen. Early user interface design will help drive expert system requirements. The following sections explain the relationship of the user interface to the expert system and define the parts of the interface that establish this relationship.

5.1.1 The Window into the Expert System

Graphics provide an excellent communication medium, and the user and developer may find this an effective means for gaining a mutual understanding of both the operations being monitored and the expert system. The DESSY project started with a storyboard which included the preliminary user interface and information on the functionality of each interface item. This storyboard served as a basis for initial interface development and expert system design. Storyboards are discussed in section 2.3.4.

The design of the user interface, like the design of the knowledge base, is iterative and interactive. Significant changes in the user interface may mean new or revised expert system rules. At each step in the process, allowing the user to interact with the interface, and thus the intelligence behind it, will expedite the development process by exposing problems or confirming good ideas. Because the user interface acts as the window into the system, it is useful not only for identifying user interface issues, but for disclosing problems hidden in the expert system as well.

In addition to the user interface window, which is designed for the end user, the developer's interface can function in a parallel manner for the developer. A common mistake in user interface design is assuming that the user needs access to all displays the developer can access. The developer should construct an interface, not necessarily available to the user, to allow interaction with the expert system from a developer's perspective. This concept is discussed in section 5.6.

5.1.2 Layers of the Interface

The user interface of any software application can be broken down into several layers corresponding to steps in its design. The highest level layer is the information content of the displays. This includes data, state, and status information to be displayed, as well as any other information relative to the monitored system. This level of display definition is purely abstract and does not include graphics or behavior definitions.

Once the appropriate information to be included in the interface has been identified, the display elements' appearance and behavior can be established. Icon descriptions such as graphics, color, and size can be defined for each item of information, and the behavior regulating the icons can be determined. Behavior includes actions such as color changes and icon changes (e.g., the DESSY end effector snare icon depicts the snares in open and closed positions). Together the appearance and behavior make up the complete display definition for each informational item, the middle interface layer.

Finally, the layer of overall screen design can be established by creating a display layout that makes effective use of individual display items. Almost always, screen "real estate" is an issue; i.e., there will be only a limited amount of screen space available for the interface. This will likely be a major factor in the interface design, and it may drive a redesign of individual display elements. Pop-ups, temporary display windows related to particular screen elements, are a useful mechanism to include relevant, but less important information in the user interface. Pop-ups lead to navigational issues such as how and where to pop up, minimizing the distraction from more important screen information.

The three user interface layers described above will be implemented in steps, but as with other expert system development will be iterative, with each step influencing the outcome of the other two. Distinguishing among these activities, however, will help lay the foundation for a useful and reliable expert system user interface.

5.1.3 Elements of the Interface

In addition to identifying the layers that are involved in user interface design, it is helpful to understand the types of elements present. The four categories of interface displays used in DESSY are text, icons, graphs and plots, and navigational items. Each is discussed briefly below.

Text displays contain ASCII characters and are found primarily as messages or description boxes. Messages are useful for logging events, particularly if they require a time stamp or should be recorded in an event log. Messages are also useful in alerting or warning the user of the occurrence of an event. Description boxes are used to display the current value in a set of changing information. This includes parameters such as state (snares are open, closed, opening, closing...), status (snares are nominal, failed-to-closed...), and time. Both types of text displays can be color-coded and recorded in a file.

Icons are a second type of display item. Icons may be graphical or symbolic. Graphical icons may contain complex drawings and include motion and color schemes. They have the advantage of more accurately reflecting a graphical view of some subsystem part. Symbolic icons are simple geometric shapes, usually having a color to indicate state or status. They are useful because they simplify screen design, and may be easier for humans to monitor. It may not be possible to say precisely whether an icon is graphical or symbolic; rather it may fall somewhere in the middle of this spectrum. The question of which type of icon is better remains unresolved. It will likely depend on the situation, and a mix of icon types may be the most appropriate.

Graphs and plots are common displays that show trending. Although they are very useful, they require significant screen space and may be most appropriate as pop-ups. DESSY does not currently contain graphs or plots; however, this is due more to limitations of the G2 environment than to their usefulness. There are, however, several applications for these items within the operations community. See the CLARE library to locate these applications.

The final type of display item is the navigational item. Although these displays do not show data, they are an important part of the user interface. The most obvious navigational item is the button. Buttons can bring up or hide additional displays, or give the user control over some expert system feature. In addition to buttons, pull-down menus are very effective in allowing the user to navigate though the expert system. In G2, any display item can have a menu associated with it. Examples of menu use in DESSY include going to pop-up displays and resetting the status of an MSID. Both buttons and menus were found to be very effective in DESSY, not only in the user interface, but in the developer interface as well.

5.2 Preliminary User Interface Design

Before developing the user interface, one must understand the existing displays of the users. This section provides help in understanding these displays and in tying the displays you design to them. Also covered is help on getting started in gathering the information elements.

5.2.1 Getting Started

For the expert system display to be accepted by end users and have a good chance for success, it should directly relate to existing displays. For this to happen, the developer must study the existing displays and their use and determine positive characteristics. The connection of the expert system display to existing displays might be as simple as having the data arranged the same way or on the same area of the screen. Although the design should be driven by existing display(s), one should be wary of jeopardizing an improved display design. Some design compromises may be beneficial in the short run, but the end goal of improved displays should remain in sight. As the system evolves, additional improvements can occur.

Once you have an understanding of the existing displays, you are ready to begin display development. Because of the nature of operational prototyping, it is important to begin development soon in the design process, providing the flexibility to change as the design matures and more information becomes available. As you construct the initial display, remember to capture all information in current display(s) that the system will monitor. Because the expert system will perform the human task of monitoring, it is important to provide the flight controller with all data with which the system makes conclusions. This will allow the controller to verify the expert system's decisions, and the confidence level in the system can evolve.

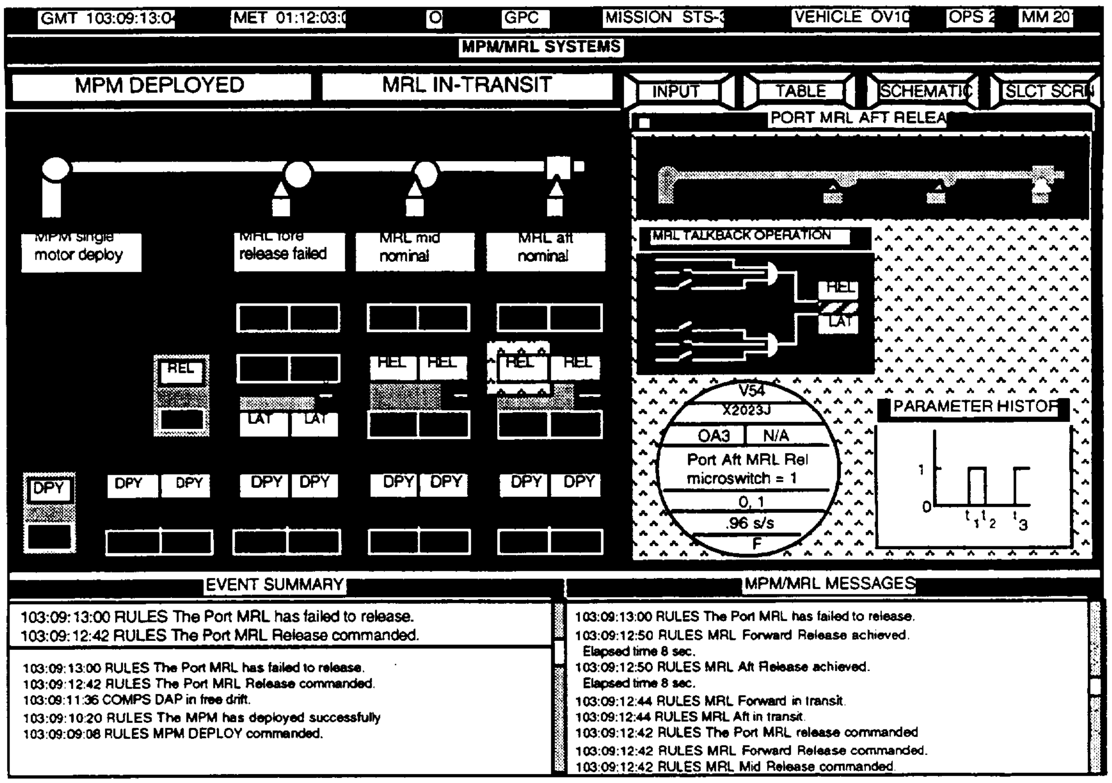
5.2.2 Gathering Information Elements

There are several standard items which seem appropriate for real-time monitoring systems. Items from the DESSY display which might also be suitable for other expert systems are presented. Figure 22 shows the original (but not final) MPM/MRL DESSY display. Compare this information to that of figure 23, the existing RMS console display. This DESSY display contained most of the MPM/MRL data that existed in the console display, along with its expert-system-supplied information. The original display concepts served well in providing the ability to understand the operations; however, changes were made as the system evolved.

Although the original DESSY display design was only preliminary, it helped significantly in identifying necessary display elements and display layout. Basic information identified for the expert system display is listed below. Also given are the types of display element used to represent information.

- Raw telemetry data
  - simple displays show active/inactive state
- Telemetry "comps"
  - simple displays represent two or more pieces of data
  - data items have been combined by *and* or *or* to represent a single value
- System state and status
  - text boxes give concise message
  - icons depict real objects when feasible
  - pop-ups lead to more detailed information
  - color coding indicates problems
- Timing
  - timers are located in appropriate screen area
- Event recording
  - message log stores and records events
- Generic mission information
  - header or panel display includes GMT/MET

For each information type, the display elements must be identified, defined, and assembled into the initial display design. As the preliminary design is created, you will be constructing a storyboard that will help lay the foundation of both the display and intelligence of the expert system. Section 2.3.4 discusses creating a storyboard, the key element in tying the support operations to the user interface display. Once the initial storyboard is in place, you can begin its evolution into the application software.



```
103:09:13:00 RULES The Port MRL has failed to release.
103:09:12:42 RULES The Port MRL Release commanded.

103:09:13:00 RULES The Port MRL has failed to release.
103:09:12:42 RULES The Port MRL Release commanded.
103:09:11:36 COMPS DAP in free drift.
103:09:10:20 RULES The MPM has deployed successfully
103:09:09:08 RULES MPM DEPLOY commanded.
```

```
103:09:13:00 RULES The Port MRL has failed to release.
103:09:12:50 RULES MRL Forward Release achieved.
 Elapsed time 8 sec.
103:09:12:50 RULES MRL Aft Release achieved.
 Elapsed time 8 sec.
103:09:12:44 RULES MRL Forward in transit.
103:09:12:44 RULES MRL Aft in transit.
103:09:12:42 RULES The Port MRL release commanded
103:09:12:42 RULES MRL Forward Release commanded.
103:09:12:42 RULES MRL Mid Release commanded.
```

Figure 22. The original DESSY display design for the MPM/MRL system.

5.3 Evolution of the User Interface

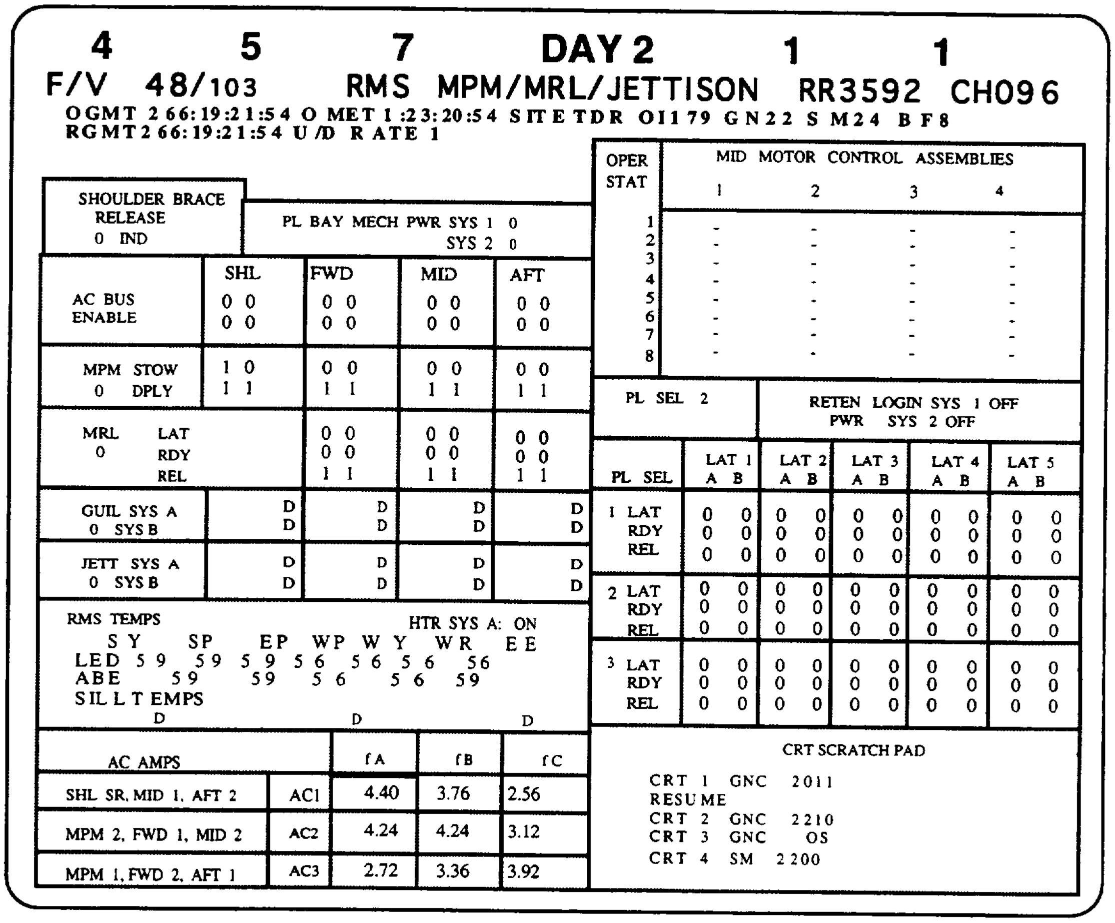
Throughout expert system development, the user interface will evolve to meet design requirements. To assist you in this development, the following sections are provided with suggestions and guidelines.

5.3.1 Designing for Change

Developers can promote user interface evolution by designing for change in two major areas. First, a separation of the user interface from the expert system will simplify design changes. Second, a well-structured display class hierarchy, like the expert system's class hierarchy, will lead to flexibility and encourage reuse of displays. In addition to taking these initial design steps, the developer should remain open to potential change. The purpose of early design and prototyping is to better understand the system and learn what changes need to be made.

In DESSY separate display classes were developed with sets of rules to control their object instances. Changes in these classes and rules did not effect the expert system logic. This design provides the option of using an alternative interface environment if the need arises. Even if G2 (or some other development tool) has been chosen to construct the user interface, it is possible that another interface tool may eventually take its place because of performance or flexibility, or even for political reasons. In any case, separation of the interface from the expert system should make the software more readable and maintainable.

Display class hierarchy is as important in the structure of displays as is the class hierarchy of the expert system. A strong display class structure will allow reasoning over appropriate sets of display items and will promote reuse. The hierarchy will evolve as the user interface evolves and will contribute to the development process. For a detailed discussion of class hierarchy of the expert system and its benefits, see section 3.2.3.



```
 4        5        7       DAY 2        1        1
F/V  48/103     RMS  MPM/MRL/JETTISON   RR3592  CH096
 OGMT 2 66:19:21:54 O MET 1 :23:20:54 SITE TDR OI179 GN22 S M24 B F8
 RGMT2 66:19:21:54 U /D R ATE 1

SHOULDER BRACE                                 OPER  MID MOTOR CONTROL ASSEMBLIES
 RELEASE       PL BAY MECH PWR SYS 1  0        STAT   1      2      3      4
 0  IND                        SYS 2  0           1   -      -      -      -
                                                  2   -      -      -      -
             SHL    FWD    MID    AFT             3   -      -      -      -
AC BUS       0 0    0 0    0 0    0 0             4   -      -      -      -
ENABLE       0 0    0 0    0 0    0 0             5   -      -      -      -
                                                  6   -      -      -      -
MPM  STOW    1 0    0 0    0 0    0 0             7   -      -      -      -
 0   DPLY    1 1    1 1    1 1    1 1             8   -      -      -      -

MRL    LAT          0 0    0 0    0 0          PL SEL 2      RETEN LOGIN SYS 1 OFF
 0     RDY          0 0    0 0    0 0                         PWR   SYS 2 OFF
       REL          1 1    1 1    1 1
                                                        LAT 1  LAT 2  LAT 3  LAT 4  LAT 5
GUIL SYS A        D      D      D      D       PL SEL   A  B   A  B   A  B   A  B   A  B
 0   SYS B        D      D      D      D       1 LAT    0  0   0  0   0  0   0  0   0  0
                                                 RDY    0  0   0  0   0  0   0  0   0  0
JETT SYS A        D      D      D      D         REL    0  0   0  0   0  0   0  0   0  0
 0   SYS B        D      D      D      D       2 LAT    0  0   0  0   0  0   0  0   0  0
                                                 RDY    0  0   0  0   0  0   0  0   0  0
RMS TEMPS                  HTR SYS A: ON         REL    0  0   0  0   0  0   0  0   0  0
   S Y    SP    EP  WP  W Y  WR   E E          3 LAT    0  0   0  0   0  0   0  0   0  0
LED 5 9   59  5 9  5 6  5 6  5 6   56            RDY    0  0   0  0   0  0   0  0   0  0
ABE     5 9    59   5 6    5 6  59               REL    0  0   0  0   0  0   0  0   0  0
SIL L T EMPS
    D             D             D                        CRT SCRATCH PAD
                                                  CRT 1 GNC  2011
AC AMPS                  fA     fB     fC         RESUME
SHL SR, MID 1, AFT 2  AC1  4.40   3.76   2.56     CRT 2 GNC  2210
MPM 2, FWD 1, MID 2   AC2  4.24   4.24   3.12     CRT 3 GNC   OS
MPM 1, FWD 2, AFT 1   AC3  2.72   3.36   3.92     CRT 4 SM   2200
```

Figure 23. Existing RMS console displays.

5.3.2 Guiding the Evolution

The process of operational prototyping is evolving a software design to its final state. For this evolution to be successful, close cooperation must exist between the developer and user. The developer will have the responsibility for soliciting feedback from the user, and the primary mechanism for obtaining this feedback is through user interaction with the expert system through the user interface. Cooperative evaluation of the interface will lead to changes in both the expert system logic and the user interface itself. Because users do not always understand the importance of their role in providing this feedback, the developer must take the initiative to encourage user interaction.

In addition to working with a key user contact, it is important for the developer to expose the other users to the application. Although it is appropriate for a single user to represent the user base, additional users interacting with the software will lead to a more complete design.

As the software is evaluated and improved, the sophistication of both the expert system and interface will gradually increase. Thus the level of complexity of the scenarios presented should increase as well. Initially, user interaction will consist of "playing with" screen displays. Soon, however, simple nominal case scenarios should be presented so user(s) can evaluate the system as a whole. These scenarios gradually become more complex and include failures. Finally, canned demos are replaced by real data playbacks. This concept of iterative operational prototyping with increasingly complex scenarios leads to a well-designed system in which all interested users have a chance to understand the system and make contributions.

5.3.3 Using a Library

Throughout this guide we have mentioned CLARE,* the Control Center Library for Application Reuse & Exchange. Using a library can greatly enhance development and improve the final application. Specific benefits of using a library include potential code reuse, better requirements understanding, and availability and guidance of standards.

The most obvious benefit derived from a library is time saved by reuse of code, provided that the code is well-tested and does indeed meet the specific requirements. A goal of CLARE is to include sample applications, or pointers to applications, for each library item. By providing accessible examples, libraries can expedite software development and eliminate redundant efforts.

A less obvious but equally important benefit of libraries is their contribution to requirements definition. Having access to an example application similar to an idea you have in mind, even if it does not exactly meet your needs, can help you better understand your own requirements and solidify them. It may also be that a modified version of the existing application will meet your specific needs. Thus additional time savings can be achieved by having access to working examples that help you understand the problem you are solving.

A third library benefit is the information that supports the development of application standards over a wide user community. This is beneficial for maintenance purposes as well as cross-discipline understanding of applications of other groups. For example, if all flight controllers choose a single plotting program, or choose navigation buttons that look the same, they will be a step ahead in understanding each other's applications. And even if flight controllers are not interested their neighbors' applications, management and software developers likely will be.

Use of a well maintained library could be a great benefit to the NASA mission operations community. The authors of CLARE encourage its use and urge developers to make their own contributions.

5.4 The DESSY User Interface

Now that the key concepts in user interface development have been introduced, the DESSY example will be presented. In figures 22 and 23, the original DESSY screen design was shown along with the existing RMS console display. The following sections provide more recent DESSY screens along with the concepts that were incorporated into their displays. More information is also available through CLARE.

5.4.1 DESSY Interface Requirements

DESSY design began with a storyboard. The original screen design is shown in figure 22. However, design consists of much more than a plan for the screen. In the information-gathering step of interface design, key concepts were identified for inclusion in DESSY. Excerpts from an information requirements document for DESSY are shown in figure 24. Most details have been removed. This document corresponds to the information layer of the interface design.

The second layer of the interface as defined in section 5.1.2 is made up of display elements. DESSY uses text displays, icons, and navigational items and incorporates a Hypercard-like approach in that it makes significant use of pop-ups to get additional information on display objects. The screen is designed to include a hierarchical representation of expert system information by presenting high level information on the main screen and more detailed information on subscreens. The following section discusses the MPM/MRL and EE DESSY displays, including display elements as well as screen layout.

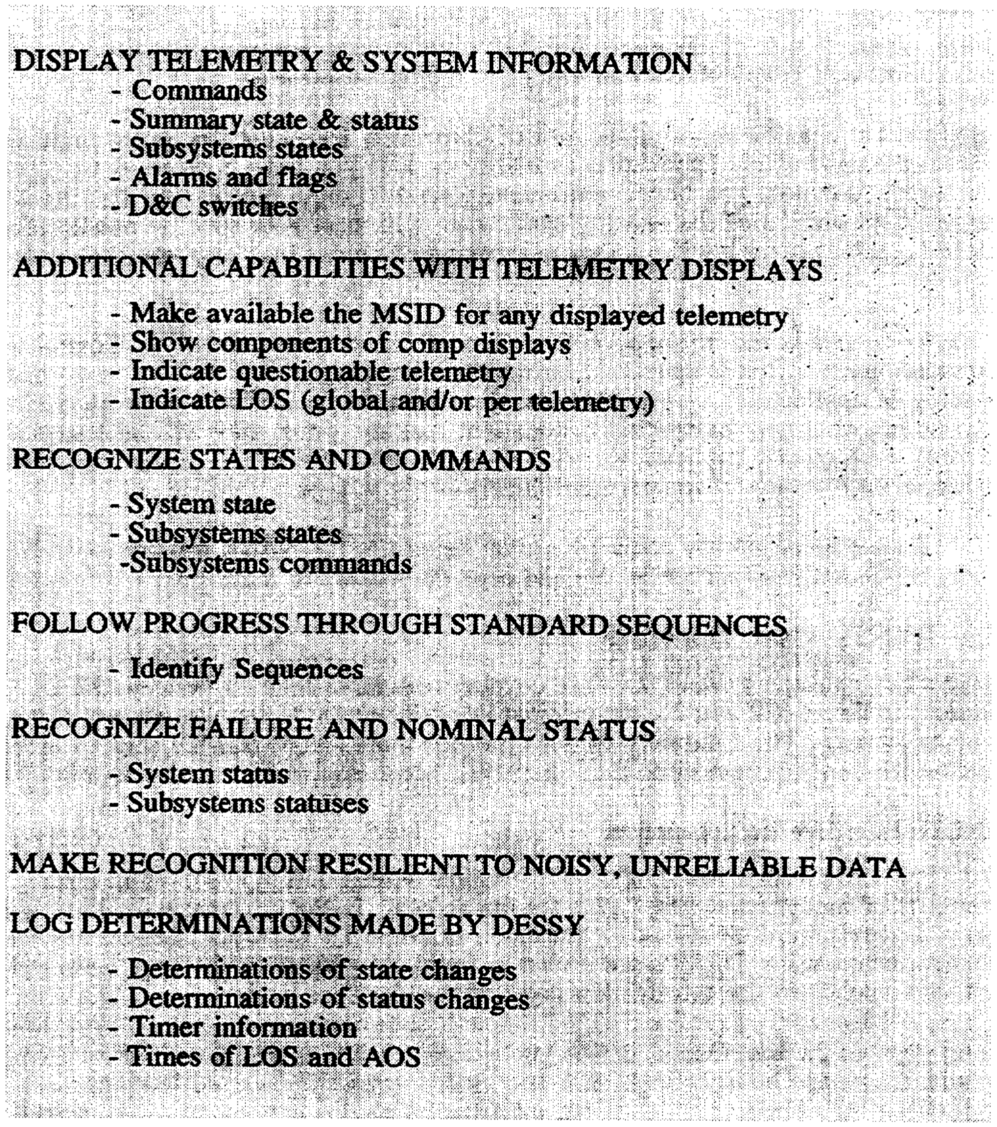
5.4.2 DESSY Display Examples

DESSY currently monitors two RMS subsystems: the manipulator positioning mechanisms and the manipulator retention latches (MPM/MRL) and the end effector (EE). Each subsystem has its own display. An overview discussion of these two screens and their displays is provided. A detailed explanation of these screens is available in the form of cue cards, which are shown in Appendix B.

*URL: http://tommy.jsc.nasa.gov/~clare

5.4.2.1 *Telemetry Data*

The first items of interest on the DESSY displays are the telemetry data boxes. On the MPM/MRL display these are represented as a set of small boxes in the center of the display. Because each piece of data is closely tied to a particular MPM or MRL, the layout of the display is such that the data boxes are placed underneath the MPM or MRL to which they are related. In addition, each column pair represents the system-1 and system-2 subparts of the MPM and MRL.



DISPLAY TELEMETRY & SYSTEM INFORMATION
- Commands
- Summary state & status
- Subsystems states
- Alarms and flags
- D&C switches

ADDITIONAL CAPABILITIES WITH TELEMETRY DISPLAYS
- Make available the MSID for any displayed telemetry
- Show components of comp displays
- Indicate questionable telemetry
- Indicate LOS (global and/or per telemetry)

RECOGNIZE STATES AND COMMANDS
- System state
- Subsystems states
-Subsystems commands

FOLLOW PROGRESS THROUGH STANDARD SEQUENCES
- Identify Sequences

RECOGNIZE FAILURE AND NOMINAL STATUS
- System status
- Subsystems statuses

MAKE RECOGNITION RESILIENT TO NOISY, UNRELIABLE DATA

LOG DETERMINATIONS MADE BY DESSY
- Determinations of state changes
- Determinations of status changes
- Timer information
- Times of LOS and AOS

Figure 24. Information requirements document.

The data boxes for the end effector are grouped and labeled according to functionality. This is because the operations of the two end effector parts, snare and rigidizer, are more highly integrated than MPMs and MRLs. End effector activity almost always includes data changes in both systems, while MPM and MRL activities are independent. Thus while the MPM/MRL screen divides data based on the physical systems, the EE screen corresponds more closely to operational activity. The MPM/MRL and EE screens are shown in figure 25.

5.4.2.2 *Switches and Talkbacks*

Both the MPM/MRL and EE displays contain switch and talkback icons. These were created in attempt to provide flight controllers with some indication of what the onboard crew saw. DESSY switches are in some cases driven directly by telemetry and at other times by expert system inference, depending upon the availability of the data. For example, the MPM Stow switch can be set to the stow position by a single Stow Command MSID; however, there is no Deploy Command MSID, and therefore other data sets the switch position during a deploy. In either case the switch represents the position of the actual hardware switch on board the Shuttle. Talkbacks are often merely single data boxes; however, they may be more complex. The MPM and MRL talkbacks are each comps made up of the logical "and" of data values for the system-1 of their respective systems. Because the onboard talkbacks behave in this manner, DESSY's talkbacks were constructed to use telemetry to do so as well.

Using icons such as switches and talkbacks does seem to add depth to the user interface, providing a more appealing screen. There is, however, a pitfall to using such realistic graphics. Switch failures, faulty data, or incorrect expert system inferencing (perhaps based on faulty data) can all lead to an expert system switch or talkback state that does not reflect the true state of the onboard hardware. It is important to make sure the users understand that switches and talkbacks, like other display objects, can only reflect the data they receive. They are data and/or expert system displays only and cannot be relied on as a certain reflection of the onboard hardware configuration.

5.4.2.3 *Graphical Icons*

Both DESSY displays use graphical icons to correspond to the systems they monitor. Square boxes represent MPMs and triangles represent MRLs, while more detailed snare and rigidizer pictorials are used for the end effector. The MPM boxes and MRL triangles lead to further MPM/MRL pop-ups with more graphical icons. Their advantage is that they are simple, yet still convey necessary information with a more complex graphic available if desired. They also have three color states—red, yellow, and green—corresponding to failed, degraded, and nominal status.

The snare and rigidizer icons contain a more detailed icon description, eliminating the need for pop-ups. These icons change their shape to reflect the current state of their corresponding system parts. For example, the snare icon depicts the snares as being open, in transit, or closed. For both systems, providing these pictorial icons along with text supports a quick state and status view of the entire system.

5.4.2.4 *Text Displays*

Text displays are also a major part of the DESSY screens, reflecting the icons discussed in the previous section. Because they provide summary level information, they are immediately visible at a glance. The MPM/MRL screen contains text displays for the state and status of each of the two systems. State and status text displays are also available for individual MPMs and MRLs through their pop-ups.

The end effector screen contains text displays for the states of the snare and rigidizer, and consolidated system text displays for the state and status of the end effector system. It was the integrated nature of the snare and rigidizer subparts that led to this display design.

In addition to the text boxes, a message list is available for both displays. Any event can be recorded to this log. Examples include state changes, power start-ups, and failures. These events are all time-stamped and may be stored in an electronic file.

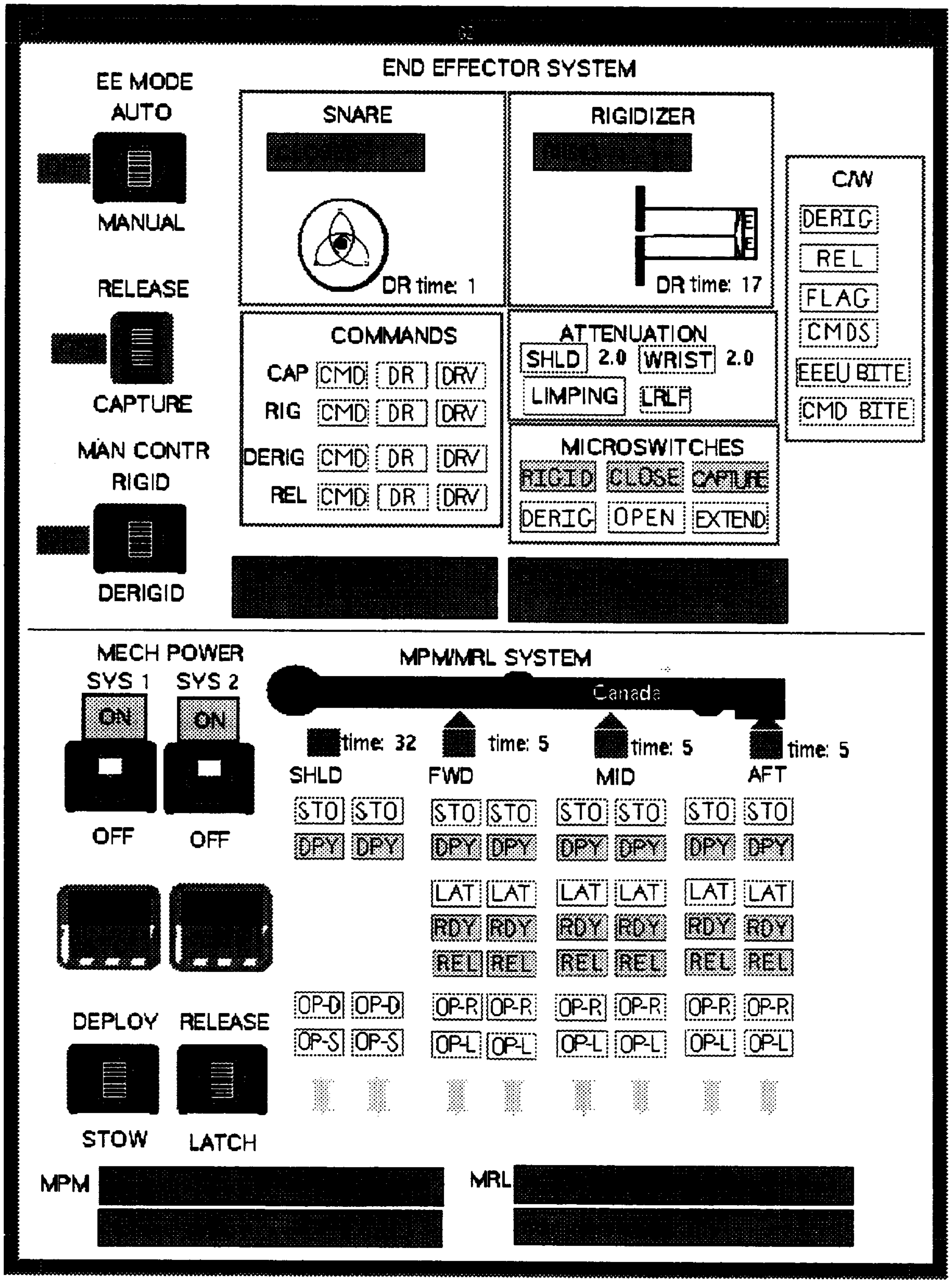


Figure 25. End effector and MPM/MRL screens.

5.4.2.5 *Navigational Items*

DESSY makes extensive use of navigational items. Both displays contain buttons and menus to reach pop-ups or control the expert system. The same types of buttons and menus are available in both. See the cue cards in Appendix B for a more detailed explanation of the navigational items.

5.4.3 DESSY Screen Arrangement

The screen layout of the intelligent system is another important aspect of the user interface. DESSY screens, shown in figure 25, contain a status overview, switch and talkback displays, system icons, and groups of data displays. An attempt was made to make the two screens as symmetric as possible; however, differences in systems led to differences in their user interfaces. The grouping of similar displays and the arrangement of those groupings is the feature of the DESSY screens on which this section focuses.

The DESSY status appears near the top or bottom of the screen to support a "status at a glance." Subpart icons, e.g., snare and rigidizer icons of the end effector system, are grouped with their state and timers, supporting the next level of information detail. Switches and talkbacks, which reflect those found on board, appear together. Finally, various types of data displays are grouped according to function.

In the end effector system, command, microswitch, and alarm data all form data groups. They are outlined and labeled on the EE screen. The commands group forms a grid whose rows represent which command is being given (CAP, RIG,...) and whose columns represent command type (CMD, DR, DRV). Microswitch and opstat data for the MPM/MRL system are grouped similarly, but have been aligned under the corresponding MPM or MRL icon, and are also in columns corresponding to their system A or system B power source. The rows of data in this grid correspond to the particular data type (STO micro, DPY micro...). Study the figure to understand these concepts. During operations, a "flow" emerges, within and among the groups, of changing data patterns. This flow must be easy for the human to follow.

Screen layout is perhaps the trickiest part of user interface design. The storyboard stage of software design and requirements definition first addresses this challenging issue. Use the DESSY examples for inspiration, but do not allow them to limit your own ideas. Keep in mind that appropriate groupings of data and consideration of flow will allow construction of a screen that best supports monitoring operations.

5.5 User Input and Control

Users of intelligent monitoring systems will have some interaction with the software, affecting its output. This will likely be limited for real-time operations, but may include a significant amount of input if the system performs diagnostics. The following sections address three types of input a user might provide to an intelligent system.

5.5.1 Real Inputs

A type of input avoided by DESSY is "real input," or information that affects the logic of the expert system. This might include setting a variable, forcing a rule to fire, or changing a data value. The input might be used to reset or change part of the intelligent system, or it might be supplemental information to be used for further state or status determination, or for diagnostics. DESSY avoids this type of information because it is currently strictly a data monitoring system. The credibility of all DESSY conclusions can be directly tied to telemetry.

Intelligent monitoring systems that can be altered by end users are more complex, but are not unreasonable. Issues of concern include affects on real- time monitoring, tracking of user input, and "undo's" or correction of user mistakes. This feature would be most useful in diagnostics where the real-time nature of monitoring systems is not in jeopardy. Some future version of DESSY might include such a feature, as long as performance is not impaired.

5.5.2 Resets

Resets involve changing some expert system part to a default value, removing the current value from the system. This is similar to, but not exactly the same as, inputting a real value into the system. Resets do not cause chaining and do not affect expert system logic. They simply replace the current visible value with the default value. They are used to remove incorrect or outdated information from the user interface.

Examples of resets used in DESSY include (1) setting a timer to zero, (2) setting the status of a data value to functional, and (3) setting the status of a subsystem to operational. Each of these actions removes the display value and resets the value in the knowledge base object, but has no other affect on the expert system.

System resets might be used if the expert system has made faulty conclusions or if the user no longer wishes to see a failure annunciation. Resets should always be recorded, along with any explanation necessary to justify the user's action.

5.5.3 Acknowledgments and Logs

Acknowledgments and logs have no affect on the expert system, but may affect the expert system display. An acknowledgment is used to verify that the user has seen an expert system conclusion. It should change the related user interface display, may include a reset, and should always be logged. Logs are used to record any information the user deems necessary. In addition to common events being logged automatically, it is a good idea to provide a log for any additional information the user wishes to record.

5.6 Expert System Management

In addition to understanding how to use the expert system to monitor operations, the end user should learn something of how the expert system works and how to manage it. Areas where expert system management is necessary include control of the data interface, control of interfaces to other software applications, and control of the expert system itself. These will be discussed in the sections below. In addition, the user should be provided with a simple mechanism for launching and shutting down the expert system.

5.6.1 Control of the Data Interface

Although the end user should not be required to understand details of the data interface, he or she should know the basics of how the expert system gets its data so that if problems with the data source arise, the user may have options for recovery. G2 programs have an interface object which acts as the connection port between your knowledge base and the G2 Standard Interface (GSI) program. The GSI program is itself the interface between G2 and the data source. As a developer, you will want to identify all normal data connectivity problems and provide indicators should these problems be detected. Although the user need not understand the details, providing access to the connection object status and presenting the user with a simple diagnosis of the situation will shorten the time to recovery and perhaps even eliminate the developer's or data manager's involvement. Sometimes recovery may be as simple as a restart or switching the data source workstation. At other times it may be more severe. In any case, providing the user with the knowledge and power to solve simple problems will benefit everyone.

5.6.2 Control of the External Software Interface(s)

Control of external software interfaces follows the same guidelines as control of the data source interface. Provide users with easy access to simple status information, and give them a chance to diagnose the problem themselves if they so choose.

5.6.3 Control of the Expert System

The mechanism for controlling the expert system within DESSY was designed to allow users to continue to use the data monitoring parts of the DESSY display, even if the expert system was determined to be unreliable and thus unusable. This might conceivably occur because of bad data, insufficient computer performance, or faulty logic. In any case, the DESSY developers wanted DESSY to be useful as a data monitoring display even if the expert system could not currently be used.

DESSY users are provided with a control button that has the effect of turning off the part of the expert system display that reflects the monitoring and diagnostic activities. The specific action is to grey-out the part of the display related to the expert system, while leaving visible the portions of the screen that simply display telemetry data. Thus any distracting faulty conclusions are dimmed while data continues to be prevalent.

This function has been deemed control of the expert system; however, it is actually a control of the display. For the user ,however, the net effect is for the expert system part of DESSY to be turned off. Although within the DESSY code there is a capability for disabling rules, and in fact appropriate rules are disabled in cases of loss of signal (LOS), the specific action referred to above is actually user interface control.

5.7 The Developer Interface

A final area of importance in developing the expert system and its interface is constructing a useful developer interface. Many of the aspects of the developer interface were covered in the design and organization subsections of section 3, "Building the System." Here are provided some additional suggestions specifically related to the developer's interaction with the software, for developers are software users too, and users may at times play the developer's role.

5.7.1 Knowledge Base Organization

Good knowledge base organization, discussed in section 3.1, is a crucial aspect of the developer interface. The developer must be able to navigate through the knowledge base quickly and must be able to easily locate items. G2 is very lenient in allowing developers to construct KBs with little or no organization. Thus developers must enforce this discipline upon themselves and organize the knowledge base so that it is useful to them and others who might develop or maintain it.

5.7.2 Access Modes

A strength of the G2 environment is that it allows G2 Modes to be defined that set specified restrictions on knowledge base items. These restrictions include actions such as going to pop-ups, activating pull-down menus, or preventing an item from being deleted or moved. The G2 manual should be consulted for a complete list of knowledge base restrictions.

In DESSY four modes are especially useful: (1) the administrator mode, provided as the G2 default mode, (2) an end-user or operator mode, (3) a testing mode, and (4) a demo mode.

The administrator mode has no restrictions. It gives the developer full access to knowledge base items.

The end-user or operator mode has significant restrictions to restrain the user from "messing up" the knowledge base or "getting into trouble." It restricts the user in such activities as moving display items, changing rules, and deleting objects. In addition, the end-user mode provides appropriate pop-up menus for items while hiding unneeded G2-provided menu choices. (See the G2 manual for help in understanding these concepts.) In short, the end-user mode provides the polishing to make the application as user friendly as possible.

The testing mode is useful to developers because appropriate testing activities such as "change value" or "start test procedure X" can be added to item menus to expedite testing. Other unneeded menu choices may be hidden, and restrictions such as those given to end users may be used to prevent the developer/tester from accidentally harming the knowledge base during this phase. The developer would define testing restrictions as needed to expedite the testing process.

Finally, a demo mode was found to be quite useful, giving a knowledge base demonstrator the appropriate menu choices, while hiding unnecessary G2 menu choices. In addition, many of the usual end-user restrictions may be applied to prevent accidents during a demonstration. The developer should define this mode to allow demonstrations to go as smoothly as possible.

5.7.3 Developer Displays

The final area of the developer interface is developer displays, items usually hidden from the end user that the developer uses in understanding or testing the system. Displays to help the developer understand the system include state diagrams and rule documentation. These items are not used as part of the knowledge base logic, but are there simply to help the developer create the system and to help future developers extend or maintain it. They also serve as a valuable aid to the developer in communicating with the end users.

There are a number of displays which aid in debugging and testing. G2 provides many simple buttons and displays; these were discussed in section 3.5. In addition, the developer may want to construct panels of frequently used items for quick access to those items. Examples from DESSY include a button panel containing the various button types used in DESSY development and a demonstration panel containing a set of buttons that launch select demo procedures. These items were found to be helpful throughout DESSY development.

Once your knowledge base development is complete, you may decide to delete most developer displays if performance or knowledge base size is an issue. If you do not have these restraints, however, leaving these developer aids in the knowledge base will ease system maintenance and help future developers understand the work you have done.

Section 6
Testing the System

After each cycle in the design and development of the expert system, a testing phase occurs. Although testing is generally thought of as the final stage of the application, and indeed certification or quality assurance testing should conclude software development, testing must occur regularly throughout operational prototyping.

Each member of the development team plays a role in ensuring the software is tested to the users' and organizations' requirements; this section attempts to define the unique roles. The developer must create a test plan which includes testing the software at all appropriate levels. In creating this plan, the developer must be aware of the various types of testing that should take place. The user(s) must be prepared to provide test case scenarios. The scenarios should vary in complexity and detail, and evolve to include more and more realistic situations.

The sections that follow cover various aspects of testing real-time monitoring systems. Section 6.1, "Verification and Validation," discusses rule consistency issues, while section 6.2, "Creating Test Objects," reviews test object mechanisms discussed in section 5.6, "Developer's Interface." "Writing Test Cases" is discussed in section 6.3 and "Testing with Real Data" in Section 6.4. Section 6.5, "Real Time Performance," brings up some important performance-related issues and section 6.6, "Testing for Continuous Improvement," relates testing to the overall operational prototyping process. Section 6.7, "Certification Testing," explains the final testing process used to gain certification of the intelligent system.

6.1 Verification and Validation

Verification and Validation (V&V) are common buzz words in software testing. Although DESSY's testing included V&V, most work done in this phase focused on test case scenarios. V&V are briefly covered as related to the DESSY project. See (Chandrasenkaran and Punch, 1987) and (Chandrasenkaran and Punch, 1988)

.6.1.1 Definitions

Verification involves ensuring that the code you write is logically correct and "bug-free." Validation, on the other hand, involves ensuring that the problem you have addressed truly meets the requirements of the actual problem. Because there are numerous papers availablein the topics, the details of these definitions are not covered. Instead, the focus is on how each concept was applied in DESSY.

6.1.2 Verification of Rules

Sophisticated software tools are commercially available to verify rule sets. However, because DESSY rules were separated into relatively small sets, manual verification sufficed. For each set, the telemetry relevant to that set was identified, along with all possible patterns of that telemetry. Because the telemetry was isolated into small groups, the problem was tractable. Valid configurations, both nominal and off-nominal, and invalid or impossible configurations were identified. Of course any data configuration could occur due to noisy data, but no diagnostic or monitoring rules need be written for meaningless configurations. However, care must be taken to ensure that meaningless data configurations resulting from subsets of noisy data do not cause diagnostic rules to fire incorrectly. This is achieved by ensuring that the scope of the diagnostics rules is narrow enough that these rules are not triggered by noisy data, while being broad enough to catch a variety of failures. See section 3.3.4, "Status Monitoring Rules."

DESSY also has some rule chaining; however, even chained-to rules are ultimately derived from telemetry in real-time monitoring systems. This concept was handled by the small rule sets using small data sets (perhaps two or three pieces of data) acting as a single parameter in the rule antecedents and conclusions. (See section 4.3.2, "Context-Sensitive Bounded Pattern Recognition"). A single parameter with several states, for example an MPM command parameter with states of Deploy, Stow and none, could be verified and then used as a single element in rule verification. State diagrams and tables listing related data were useful in identifying valid data sets for various system configurations. This approach worked adequately for DESSY because the levels of chaining are shallow, i.e., usually two or three rules deep. A more complex expert system would likely require a more rigorous verification approach.

6.1.3 Validation of the Knowledge Base

Only the user expert can reliably validate a knowledge base because this task requires a deep understanding of system operations. The approach used in DESSY was to use real test scenarios, either mission or simulation data, to validate expert system behavior. Many times in this process it was discovered that the developers, and even the expert, had made incorrect assumptions about data behavior, and DESSY rules had to be changed. Even the end user may find it difficult to correctly recall exact data behavior of all operations and failures. Having access to the expert was particularly important in this phase of software testing.

Testing with real data and having the user expert closely involved both in development and testing were the validation approaches applied to DESSY. It is important to have access to reliable test examples for all operational scenarios modeled, and multiple similar test cases which might include noisy data elements are even better. Although no formal validation mechanisms were applied, this approach was found to be reliable and led to a correct and robust system.

6.2 Creating Test Objects

To thoroughly test the knowledge base, it is useful for the developer to create a suite of test objects that are easily accessible throughout development, to both the development team and the end users. These include test buttons, readout tables, and operational scenarios procedures. Section 3.5, "Quick Test Buttons and Displays," provides detailed discussions of many test objects one can create in G2. Section 5.6, "Developers' Interface," illustrates other ways G2 can be used to enhance the testing process. Scenario procedures are discussed in the following section.

Test objects can be created and placed with their relevant rule sets and/or objects, or they may be located on separate testing panels. The former approach can make locating the objects easier, while the latter allows simpler scenario construction and provides the capability for these objects to be stripped out of the final delivered system. Your knowledge base may include both.

Providing a separate test panel for end users was found useful in DESSY verification. Users can configure the panel with select data patterns by setting MSIDs, and observe the results on the DESSY screen. This is more favorable than directly setting the displays because users can create a data configuration scenario on the test panel and then observe that scenario on the DESSY screen. Users also want easy access to MSID number during testing, which is available on the test panel. The test panel is an important DESSY display because it allows users to build confidence in the system by directly controlling its inputs.

6.3 Writing Test Cases

Once you have satisfactorily tested individual knowledge base components with quick test buttons and displays, you can began developing operational scenarios. Although there are several methods of doing this in G2 (e.g., using the G2 built-in simulator or reading a test data file through G2's GFI interface), scenario testing in DESSY was done using a very simple G2 mechanism, the G2 procedure.

6.3.1 Simulation Procedures

The G2 procedural language is very similar to Pascal and can be used to manipulate almost any knowledge base item. Its key benefits in scenario development are its ability to execute statements both sequentially and in parallel, its ability to launch other procedures as either dependent children processes (waits on return) or independent processes (executes in parallel), and its wait statement. These features allow the developer to construct very flexible test scenarios. In addition, G2 procedures have standard procedural features such as local and passed parameters, if and case statements, and error handling capabilities.

6.3.2 Scenario and Configuration Development

In developing test procedures, the programmer should begin with simple nominal case scenarios. A scenario can be developed for each known system operation, and scenarios should cover all nominal configurations defined in the state diagrams. These nominal case scenarios can gradually be extended to include the effects of noisy data or single sensor failures. It is important to verify that the expert system reacts appropriately to nominal scenarios before more complex failure scenarios are explored. Verifying nominal procedure activity is key in the validation part of testing.

It will be helpful to write procedures as generically as possible, passing in appropriate parameters, and breaking procedures into smaller building blocks to expedite the overall testing process. For example, in the DESSY MPM/MRL system, the RMS Power-Up and Power-Down subsequences were written as stand-alone procedures. They were then called out by higher level procedures as part of a simple scenario. If desired, more complex scenarios, possibly including both MPM and MRL operations, could also be constructed, primarily for demo purposes.

Once a complete suite of nominal procedures has been constructed, the failure spaces can be explored. Standard types of failures, discussed in section 2.3.5, should be included in the test scenarios. Making these procedures flexible with appropriate input parameters will expedite testing. In DESSY, some failure scenarios even included a randomness factor. For example, the test scenario might fail a randomly selected microswitch, and then proceed with operations. Adding randomness to test scenarios increases their usefulness in verification by checking system behavior and in validation by proposing new scenarios.

In addition to creating test scenarios, it was useful to create procedures to set up particular data configurations. Examples include an MPM-stowed procedure which simultaneously sets all stow microswitches to active and deploy microswitches to inactive, or an MRL-latched procedure which sets latch micros to active and release micros to inactive. Note that these are not scenarios—they are static system configurations. They provide yet another way for the developer and user to interact with the expert system.

6.4 Testing with Real Data

Although "canned demos" or scenario procedures are an excellent means for testing a knowledge base, there is no substitute for testing with real telemetry data, running from a real data source. This data may be live or prerecorded; it may be from a mission or from a simulation from the Shuttle Mission Simulator (SMS). Any of these sources will work; however, access to all of them is ideal.

6.4.1 Real Data and System Validation

Real data will provide the ultimate validation test for your knowledge base. Because even experts can be mistaken about the details of operations, data taken directly from those operations provide the most reliable source for system validation. In fact, even though SMS simulation data should be thoroughly used and is the only practical way to test a wide range of real failure scenarios, only mission data provides unquestionably accurate test cases.

The best way to test and validate a knowledge base for NASA's mission control environment is to obtain a test suite of scenarios generated by the SMS. This is not necessarily a straightforward task, and help from the flight controller will be needed. Once the test suite of all known failure classes is constructed, the data should be fed into the expert system and records should be kept of system performance. Because it is unlikely that the expert system will pass all tests the first time around, the developer and user team should be prepared to resolve any problems, whether they be verification or validation problems, and replay the test suite. Only when the expert system can give satisfactory performance during the entire suite should it be considered validated and ready to use.

6.4.2 Completeness Checking

An inevitable limitation of expert systems is incompleteness, their inability to detect all failures, even all known failures. Rather, expert systems can only detect failures which have been specifically entered into the system knowledge base. It is important for the expert system development team to see that the user group understands this common-sense limitation, and does not expect super expert capability to detect any failure whatsoever.

Real time data testing can help increase the completeness of the expert system by exposing failures that the expert may have forgotten or not known about. In DESSY, several new failure scenarios were discovered during the testing phase, leading to improvements to the software. Although the developers were able to verify DESSY behavior for many known configurations, it is not possible to track all possible sequences of data. Thus it is unlikely that a knowledge base can ever be thought of as complete. In any case, testing the system with real data will increase expert system completeness and decrease the chances that the system will miss a known failure.

6.5 Real-Time Performance

Another issue that arises during the real-time testing phase is that of expert system performance. Because performance can vary greatly depending on the computer platform and its load, it is important to test your software under the same loading conditions that it will operate under during real operations. This testing setup should represent the exact configuration, including computer and software load, monitor, and network load.

In an earlier phase of the DESSY project, development and testing was done on a SPARC 2 platform with only one copy of G2 running. At that time the mission control environment consisted of a DEC with a Masscomp display. There was also the possibility of multiple G2s being used simultaneously in the control center. DESSY was deployed into this environment without any performance testing. It was then discovered that the Masscomp display could not support the expert system's graphics. Needless to say, this set the application back and greatly lowered user acceptance.

A related performance issue for DESSY is software loading time. Because DESSY is made up of many software modules, it takes several minutes to load. In the mission control environment when operations occur rapidly, this is often unacceptable. The solution was to merge the final DESSY into a single file to reduce load time. G2 also has a "strip text" option, which can decrease the size of the knowledge base and reduce load time.

Testing in the actual operational environment is extremely important. This testing should include that hardware and time constraints that exist in real operations. The DESSY developers learned this lesson and hope future software developers can learn from these experiences!

6.6 Testing for Continuous Improvement

Developers should remember that testing is only one phase of the entire development process. As you begin to test, establish expectations of making improvements; do not have an accept or reject attitude. Because of the nature of operational prototyping, you will have constructed a system that is evolving. You should develop appropriate evaluation criteria for where you are in the process and plan your test cases and expectations accordingly. Following the levels of testing defined above, quick test buttons and displays, simple scenario procedures, complex scenario procedures, and finally real data will allow you to evolve the system into a robust tool that can be deployed into the user's environment.

6.7 Certification Testing

Certification testing is the final phase of testing in which a software application is approved for use on a flight controller's console for the purpose of making flight-related decisions. It occurs after all other testing has been performed, as shown in figure 26, which depicts the life cycle of an intelligent system. It includes both a number of integrated simulations and "flight following." In both forms of certification testing, the flight controllers perform their jobs without knowing what problems to expect. This provides a realistic test of how well the intelligent system supports the flight controllers in performing their job.

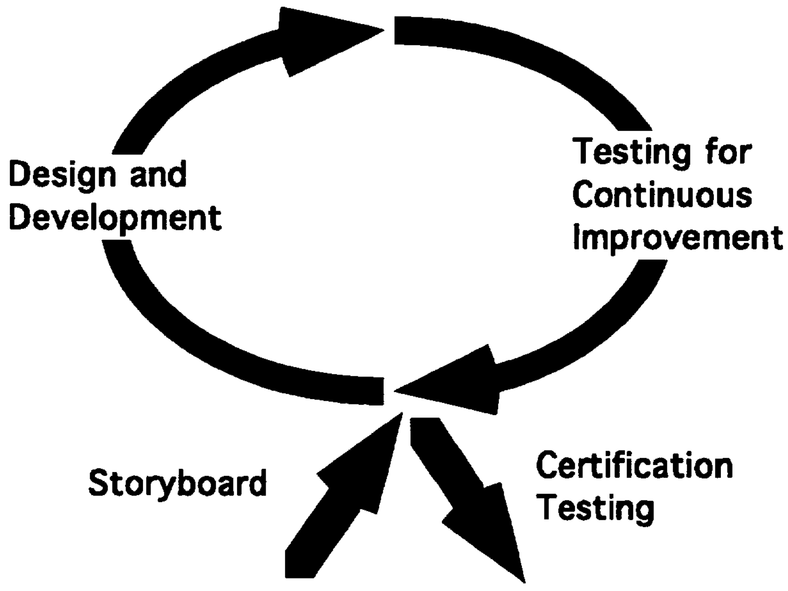


Figure 26. Intelligent system life cycle.

Integrated simulations, the first type of certification testing, help test the software by confronting it with a large number of failure situations (more than would be expected from a comparable amount of actual mission data). Simulations are considered integrated when they involve more than just one flight control discipline. Not only does an integrated simulation allow a fuller portrayal of a complex problem, it also tests performance in the presence of all other software in the control center. It allows detection of performance problems and difficulties in receiving telemetry streams when the computers are loaded.

Flight following is a testing situation in which live mission data is directed to the software application, but flight-related decisions are not based on the results from that application. Flight following typically presents fewer problems from the monitored system than simulations, but tends to present the software with noisier data. Consequently, flight following tests the software more rigorously in terms of receiving telemetry and dealing with noisy data, whereas simulations test software more rigorously in terms of detecting monitored process problems.

Certification is a stamp of approval given to a software application when it has been shown trustworthy enough to be relied upon for flight-related decisions. This trustworthiness is rarely stated in purely quantifiable terms, probably because there are a number of ways the software can perform differently from its user's expectations. One critical error might be enough to deny certification, whereas a number of noncritical surprises might be tolerated.

DESSY certification is expected to occur in phases. Initial certification will be for presentation of telemetry. Later certification will be for expert system performance. In other words, DESSY will be considered useful for supporting flight-related decisions if it presents telemetry reliably. Later, after considerably greater experience with DESSY's fault recognition logic (and when that logic has proven itself accurate and reliable) the expert system portion of DESSY will be certified. There is also indication that a related application called the Remote Manipulator System Checkout software, or RMSCO, will also undergo a phased certification. (See CLARE for information on this application.) This will allow RMSCO to provide early, basic support for monitoring RMS checkout procedures, followed by later, enhanced support. It illustrates the certification process as a stamp of trustworthiness for a specific type of support.

Section 7
Documentation and User Training

As in any software project, documentation and user training are both necessary steps toward a complete deliverable product. This section discusses specific knowledge base aspects that should be documented and provides several effective training mechanisms. Sections 7.1, "Documentation within the Knowledge Base," discusses several knowledge base structures that can be used to support documentation. Then "Documentation of Requirements," which aids in the evolutionary prototyping process, and "Documentation of Testing," necessary to ensure that the software correctly supports the task, are covered in sections 7.2 and 7.3.

The next sections focus on user training; however, setting up the suggested training tools is another excellent way to further document your knowledge base. Section 7.4 discusses cue cards and their benefits. Section 7.5, "The Tutorial: Interactive Exploration of the Expert System," provides yet another example of an effective documentation/training tool. Scenario demonstrations are discussed in Section 7.6 This construct provides documentation of test scenarios as well providing a template for effective demonstrations. Finally, Section 7.7 discusses the steps in "The Developer-User Handover," including delivery of documentation and training materials, as well as the software itself.

7.1 Documentation Within the Knowledge Base

A variety of tools can be used to effectively document your knowledge base. Discussed are several types of documentation found in DESSY, including organization strategies which provide easy-to-follow maps into the knowledge base. Other documentation forms such as state diagrams and references to supporting literature are also useful. They can be left within the application itself to support further development and maintenance, or can be stored in a separate knowledge base module that is merged into the system when needed. Modularizing the documentation segment of the knowledge base may be advisable if performance or knowledge base size is an issue.

7.1.1 Knowledge Base Organization

One of the most effective means of documenting a knowledge base is to properly organize it so that an outline of the software is provided within the hierarchy. The G2 tool supports this type of organization by allowing users to create hierarchies of both workspaces and modules. Sections 3.1.1 and 3.1.2, discuss these concepts.

Modules are G2 files which allow the user to logically separate software where needed. They contain dependencies upon one another, forming a hierarchy. To create a workspace hierarchy within a module, a workspace holder object is defined which serves as the basis of the internal hierarchy structure. These workspace holders are then labeled appropriately, providing an outline of the software within a module from the highest to lowest levels.

How do modules and workspace holders contribute to documentation? Because these structures force the user to think of software in a structured, hierarchical manner, an outline will be formed of the software as the system evolves. This outline can serve as the backbone of code documentation. For example, at the highest level, the module hierarchy provides insight into software. Figure 27 shows the end effector module hierarchy. Even from this abstract view, it is easily seen that this knowledge base contains rules associated with the end effector, uses a timer, and contains predefined simulations.

Within each module workspaces also form hierarchical outlines of the software. In DESSY each module has a root workspace from which all others are created. Examples of workspace hierarchies from two of the modules are provided in figure 28. The EE-Defs module, to which EE-DEFS-ROOT belongs, contains a variety of end effector definitions as well as some objects and specific end effector timer information. EE-RULES-ROOT, from the EE-Rules module, contains all end effector rules. Having this structure easily accessible not only documents the types of rules provided, but gives the peruser quick access to any rule category.

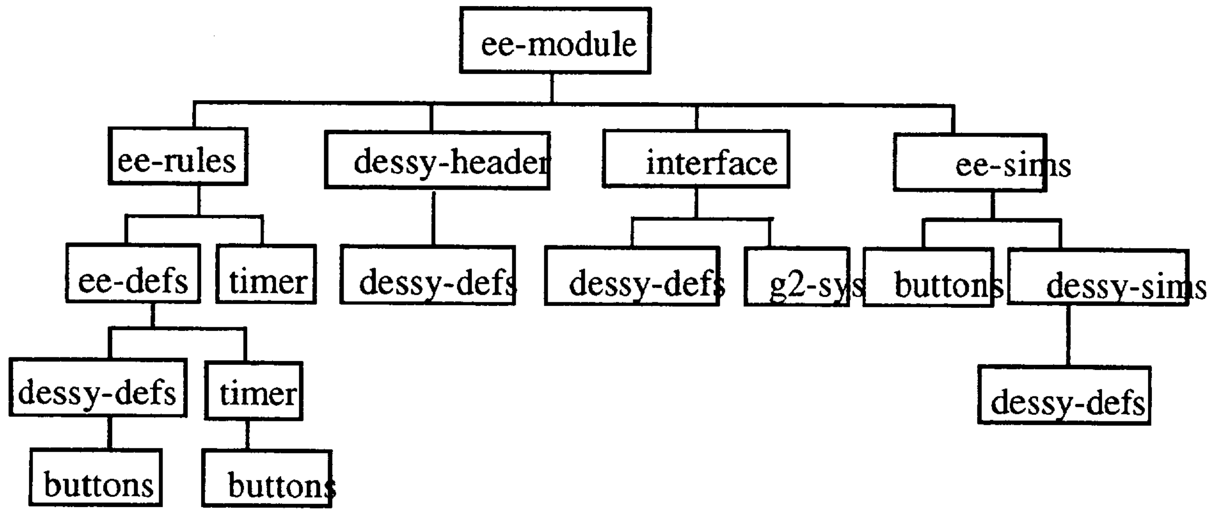


Figure 27. End effector module hierarchy.

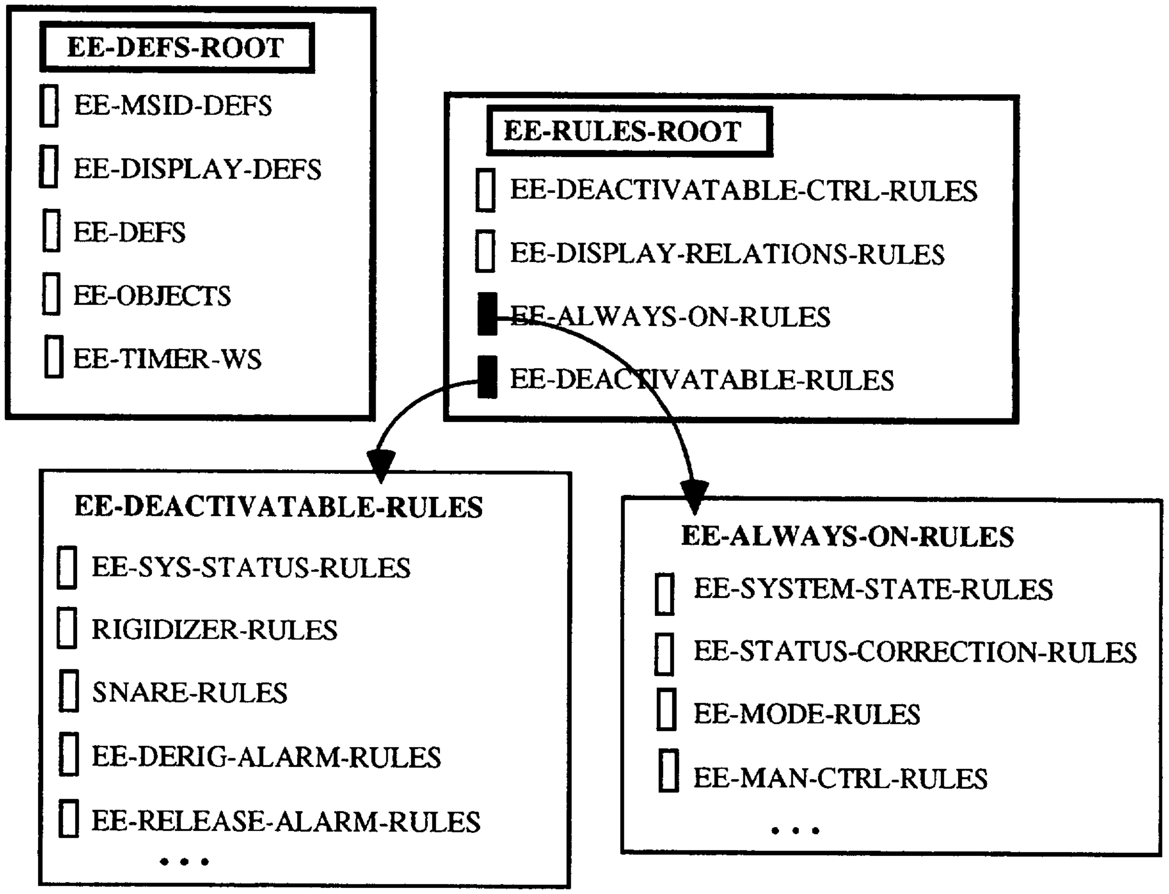


Figure 28. End effector sample workspace hierarchies.

The EE-RULES-ROOT tells the knowledge base inspector that some rules are deactivatable and others always remain turned on. The subcategories of rules can then be followed as well. Note that those shown in figure 27 represent only a partial listing of all end effector rules categories. The MPM/MRL module has a similar module and workspace hierarchy, so anyone familiar with the end effector module should be able to peruse this module as well, and vice versa.

The workspace and module hierarchies are just the beginning of knowledge base documentation, but they represent an important first step that should lead to other good documentation habits. Documentation should be easily located and should be readable. A strong knowledge base organization will support both these factors.

7.1.2 State Diagrams

Another form of documentation useful in DESSY is the state diagram. Originally used as a tool to understand states or physical orientations of a hardware system, state diagrams can also serve to document this understanding and the rules which result. Figure 16 showed the state diagram of the MPM. Because MPM state change rules were constructed directly from this diagram, it remains an ideal graphical overview of the state change rules. It captures the significant knowledge embedded in the rules and is easy for an expert system novice to understand.

State diagrams may be static or animated to reflect the current state of a running system. They may be hierarchical in nature and reflect substates of a given state. Although these extensions were not fully explored in DESSY, they are simple to construct and might be appropriate for your knowledge base documentation. If performance is an issue and you feel state diagrams would impair your running knowledge base, the state diagrams could be stored in a separate module that is eliminated from the module hierarchy during run time.

7.1.3 Flight Rules and Other Support Documentation

Support documentation, probably taken directly from literature and flight manuals, should be exploited throughout knowledge base construction. Although it is not feasible or even desirable to have all support documentation on line, that is, recorded within your software, you will probably want to include references to relevant documentation wherever possible.

Flight rules and schematics are specific examples of sources for documentation. In DESSY some expert system rules reflect a specific flight rule, information which would be useful to the flight controller users. Tables from schematics that contain data set configurations and their implications are also useful. These tables can be included and enhanced by highlighting current scenarios within the table. DESSY contains several of these animated tables.

Documentation with credible sources will give credibility to your knowledge base. Some documentation will serve only as a developer's guide to your system knowledge, while other documentation may be accessible by the user when the system is running. Keep in mind that providing references for rules and definitions will not only help other developers, but will help you as you return to parts of the knowledge base you haven't recently visited. In any case, consistency, thoroughness, and preciseness will lead to a usable and maintainable system.

7.1.4 Documentation Overhead

The only disadvantages to documentation in a real-time system are knowledge base size and potential performance impacts. If performance or size is an issue, you may need to limit the level of documentation included in the run time system. This can be done in two ways. First, it may be possible to record much of the documentation in a separate module that is not loaded when the run time system is deployed. This solves performance issues; however, it might prevent you from placing documentation near rules and objects where this information would be most useful. Second, you may want to include only reference pointers instead of the reference documentation itself. Although this limits the information available to the user or developer, it may be all that is needed for parts of the expert system.

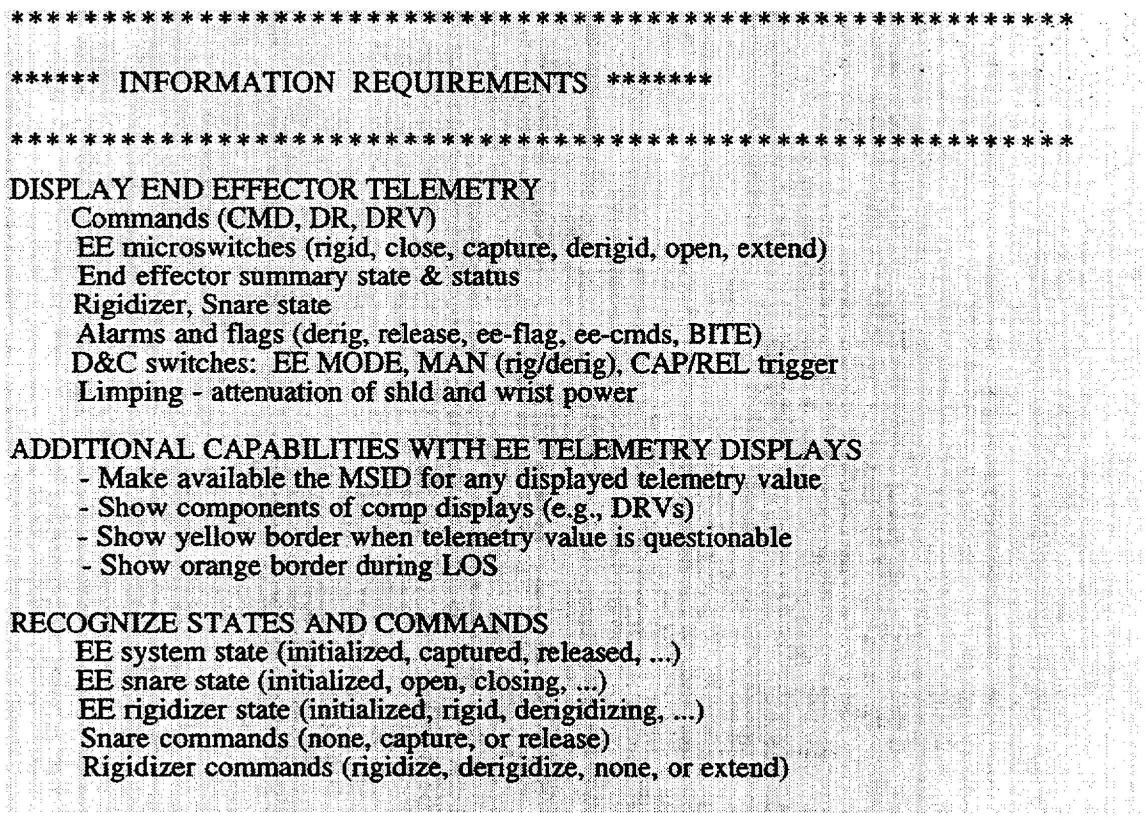
In DESSY, on-line documentation was limited to prevent the knowledge base from becoming too large and so that performance would not be affected. However, select reference information that flight controllers said would be useful in performing their run time tasks was included. Examples include telemetry bubble pop-ups corresponding to displayed MSIDs and logic tables taken from subsystem schematics. In addition, intelligence was added to logic tables to provide the user with even further assistance.

The specific type and amount of documentation for your system will depend on a number of factors, including performance, availability, and user need. Documenting throughout the prototyping process, however, can improve understanding of the application, and therefore produce a superior knowledge base product.

7.2 Documentation of Requirements

Documentation of requirements can help guide developers through prototyping, as well as provide a tool for tracking progress. The DESSY requirements document was a living document throughout DESSY development. It was not a formal list of application "shalls." This document contains plans for future development as well as accomplishments.

Specific information in the requirements document includes, for example, identification of displays and their functions; identification of system states, commands, and failures determined by the expert system; special features such as resilience to noisy data; logging capabilities; and test case scenarios. A shortened version of the End Effector DESSY requirements document is provided in Figure 29.



```
**********************************************************

****** INFORMATION REQUIREMENTS *******

**********************************************************

DISPLAY END EFFECTOR TELEMETRY
    Commands (CMD, DR, DRV)
    EE microswitches (rigid, close, capture, derigid, open, extend)
    End effector summary state & status
    Rigidizer, Snare state
    Alarms and flags (derig, release, ee-flag, ee-cmds, BITE)
    D&C switches: EE MODE, MAN (rig/derig), CAP/REL trigger
    Limping - attenuation of shld and wrist power

ADDITIONAL CAPABILITIES WITH EE TELEMETRY DISPLAYS
    - Make available the MSID for any displayed telemetry value
    - Show components of comp displays (e.g., DRVs)
    - Show yellow border when telemetry value is questionable
    - Show orange border during LOS

RECOGNIZE STATES AND COMMANDS
    EE system state (initialized, captured, released, ...)
    EE snare state (initialized, open, closing, ...)
    EE rigidizer state (initialized, rigid, derigidizing, ...)
    Snare commands (none, capture, or release)
    Rigidizer commands (rigidize, derigidize, none, or extend)
```

Figure 29. The End Effector DESSY Requirements Document, shortened.

FOLLOW PROGRESS THROUGH STANDARD SEQUENCES
RECOGNIZE FAILURE AND NOMINAL STATUS

EE system status (operational, timeout, capture-failed-to-initiate, ...)
Snare status (operational, nominal, snare-opening-timeout, ...)
Rigidizer status (operational, nominal, rigidizing-timeout, ...)
Microswitch conflicts which trip ee-flags on Shuttle are recognized

MAKE RECOGNITION RESILIENT TO NOISY, UNRELIABLE DATA
- Recognize states and commands within situation context.
- Corrective rules - when off-nominal condition in telemetry goes away, so does recognition of off-nominal status
    - microswitch status
    - state recognition
    - system status
- Some changes in state, status, and commanding recognized only when telemetry has continually indicated change for 2-3 seconds
- Suppress state and status determinations during LOS

LOG DETERMINATIONS MADE BY DESSY (along with GMT or MET of entry)
- State, Status changes
- Timer information
- Times of LOS and AOS

****

************ TEST CASES ********************

****

NOMINAL SEQUENCES
- auto release
- auto capture
- manual release
- manual capture
- ee checkout (capture and release)

EE FLAG FAILURES (inconsistent talkbacks)
- rigid-derigid
- close-open

. . .

EE CMDS FAILURES
- auto derig w/o DRV
- auto rig w/o DRV

. . .

AUTO TIMEOUT FAILURES
- auto release with derig timeout
- auto release with snare timeout

. . .

MANUAL TIMEOUT FAILURES
- manual release with derig timeout
- manual release with snare timeout

. . .

SEQUENCE VIOLATIONS
- auto capture without limp
- auto release with CLOSE ms failed on

. . .

FAILED TO INITIATE SEQUENCES
- open fails to initiate
- close fails to initiate

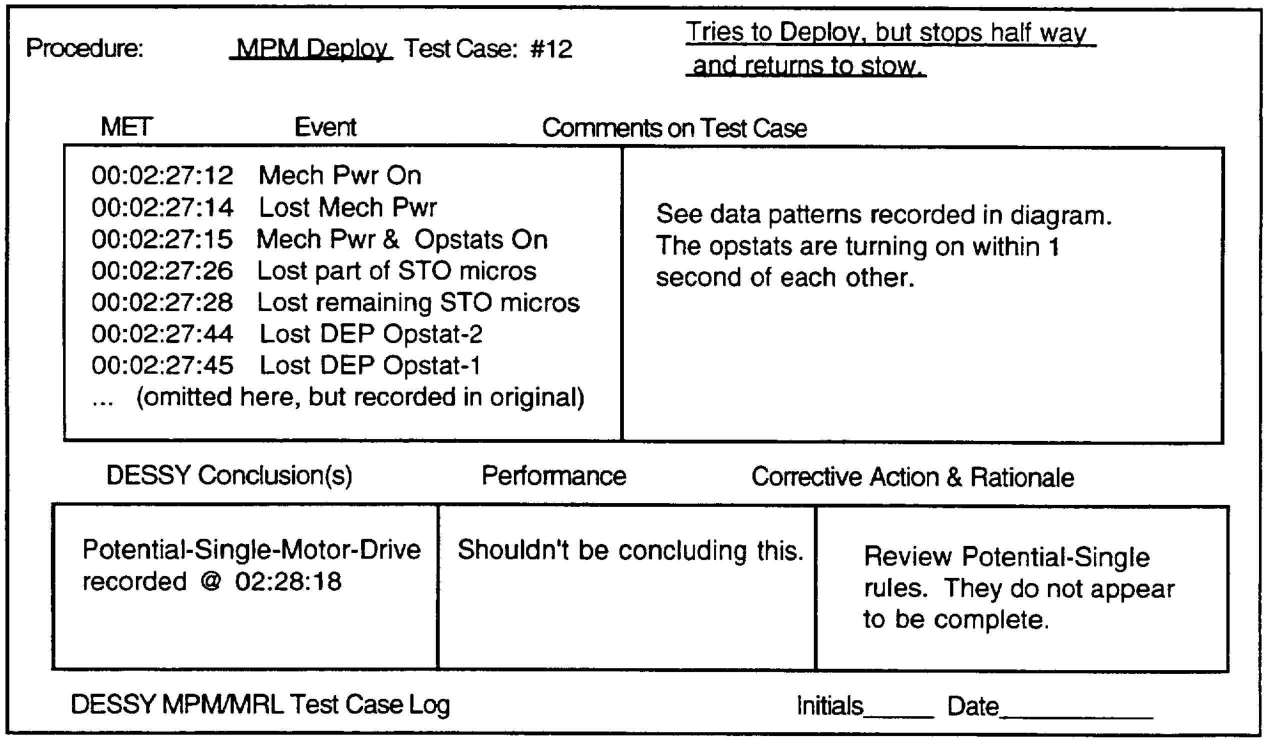
Figure 29. (concluded)

7.3 Documentation of Testing

Section 6 discussed the necessary steps and motivation for testing expert system software. There are two independent phases in system testing. The first is to find logic errors in rules so that the system may be evolved and changed to correctly monitor operations. The second phase begins only when the first is complete. It is to verify that there are no errors in the rules and that no additional changes need to be made. Both phases need to be thoroughly documented.

When testing is done to evolve and correct the system, test runs are repeatedly made and the results recorded. DESSY has a test file book in which each test run was recorded, along with the expert system's behavior and any corrections that were made to the system for that run. A subsequent run for that case would again record behavior and document corrections if any were needed. Figure 30 shows an example from the Test Case Log used in DESSY, and figure 31 shows a subsequent run after modifications were made. Also provided is a set of simplified screen snapshots on which particular data patterns can be easily recorded. An example is shown in figure 32. Note that the example shown provides the MET, indicating that the data came from real telemetry, rather than a "canned" test procedure written by developers. Tests made with canned data would be recorded similarly.

After sufficient testing has been done and appropriate changes to the system made, a final set of runs for all test cases is made to verify that no further changes are needed. These should be recorded in the Test Case Log as well. Once all testing is complete, the Test Case Log may be thought of as deliverable documentation. If later it is determined that the expert system needs modification, the Log may be used again and the process repeated.



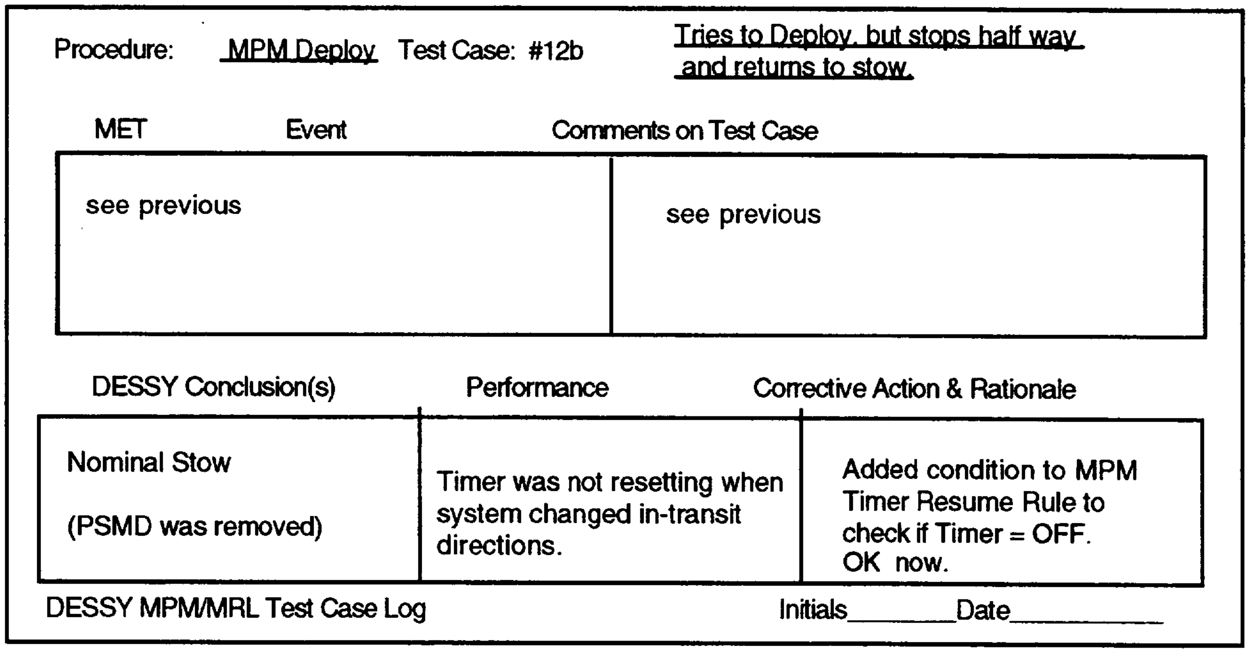
Procedure: MPM Deploy Test Case: #12 Tries to Deploy, but stops half way and returns to stow.

MET	Event	Comments on Test Case
00:02:27:12	Mech Pwr On	See data patterns recorded in diagram. The opstats are turning on within 1 second of each other.
00:02:27:14	Lost Mech Pwr	
00:02:27:15	Mech Pwr & Opstats On	
00:02:27:26	Lost part of STO micros	
00:02:27:28	Lost remaining STO micros	
00:02:27:44	Lost DEP Opstat-2	
00:02:27:45	Lost DEP Opstat-1	
...	(omitted here, but recorded in original)	

DESSY Conclusion(s)	Performance	Corrective Action & Rationale
Potential-Single-Motor-Drive recorded @ 02:28:18	Shouldn't be concluding this.	Review Potential-Single rules. They do not appear to be complete.

DESSY MPM/MRL Test Case Log Initials_____ Date__________

Figure 30. DESSY Test Case Log–first run.



Procedure: MPM Deploy Test Case: #12b Tries to Deploy, but stops half way and returns to stow.

MET	Event	Comments on Test Case
see previous		see previous

DESSY Conclusion(s)	Performance	Corrective Action & Rationale
Nominal Stow (PSMD was removed)	Timer was not resetting when system changed in-transit directions.	Added condition to MPM Timer Resume Rule to check if Timer = OFF. OK now.

DESSY MPM/MRL Test Case Log Initials________Date___________

Figure 31. DESSY Test Case Log—second run.

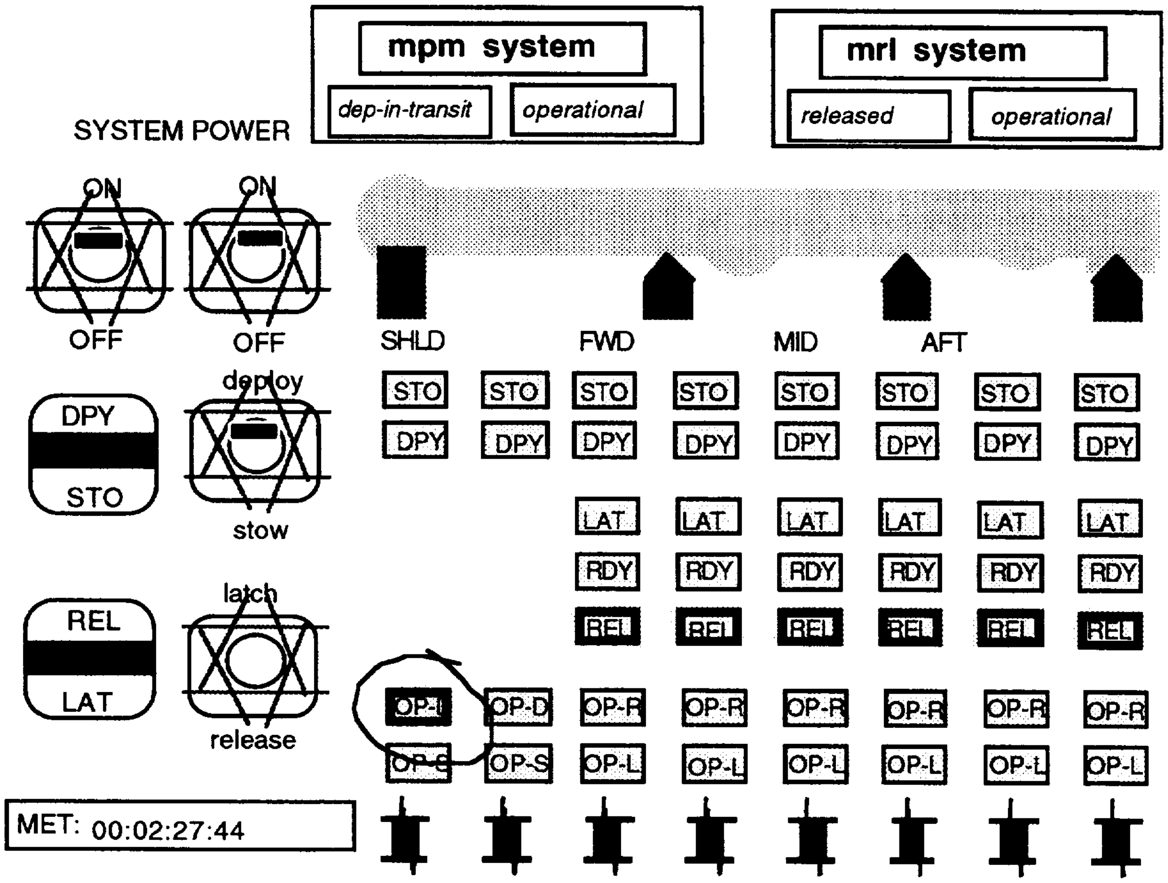


Figure 32. Screen snapshot for recording data patterns. Anomalous data is circled.

7.4 The Cue Card: Map to the User Interface

The Cue Card is an excellent means of knowledge base documentation, as well as a tool that can be greatly beneficial to end users both in learning the system and as a system reference. The cue cards created for the DESSY MPM/MRL and end effector modules are available as Appendix B. The cue cards, which consist of two color pages each, have specific formats and contain select types of information. Because DESSY does not include a users guide, the cue card is especially important as a reference to flight controllers.

The first page of each DESSY cue card provides a thorough description of all screen displays. A snapshot of the screen is shown, and each item is labeled and includes a brief statement about the item's function. Listings of all possible expert system conclusions for the system state and status are provided. Color interpretations can also be found on this page.

Page two of the cue card focuses on navigation through displays and contains information on buttons, tables, and pop-ups. An example is shown for every item type that has a pop-up, such as the telemetry bubbles associated with MSIDs. Directions for viewing these pop-ups are given. Any special windows associated with particular buttons are shown and described. Any tables that provide the user with supplemental information are also provided.

Cue cards should be included with any software products delivered to the mission operations environment. They are not difficult to construct and provide excellent screen documentation. They are an effective communication tool, serving as a descriptive overview to anyone not familiar with the intelligent system. In addition, their conciseness and convenience make them the ideal reference card for use during mission operations.

7.5 The Tutorial: Interactive Exploration of the Intelligent System

A second type of documentation which is also quite effective in user training is the interactive tutorial. The tutorial, if built properly, will allow a novice user to peruse the expert system interface and gain an understanding of each display piece without any outside help. The DESSY tutorial was designed as a page turner, which means that the user browses through a series of pages, each referencing a particular expert system aspect. The screen is assembled piece by piece as the user progresses through the tutorial. Once the user has completed this section, a demo section is available. Here the user runs test case scenarios, gaining a complete view of the expert system use in mission operations.

7.5.1 The Tutorial as a Training Tool

Like the cue card, the tutorial focuses on the expert system through the user interface, rather than covering specific rules and design strategies. Most end users do not need or want access to this more advanced information. Interested users would be welcome to explore these aspects of the expert system along side the developer. The pages of the tutorial contain simple explanations about screen items, how they are used, and how they will change when data is present. User input buttons are provided to cause tutorial displays to change based on the user's simulation of data. Thus the users can interact with the displays as well as read about their functionality.

The DESSY tutorial covers both the MPM/MRL and end effector subsystems and contains both mission header and summary information. As the tutorial user traverses the pages of the DESSY tutorial, the MPM/MRL or end effector screen is built one section at a time. At the completion of the tutorial section, the entire screen is built. This gives the user a chance to focus on each screen item, but concludes with a view of the screen as a whole. In addition, sections may be repeated and pages may be traversed forwards or backwards, providing a means to review any subsection at any time.

Once the user has completed this part of the tutorial, he or she can move to the demonstrations section. This part of the tutorial is intended for users who understand the basic display items and are ready to see how operational scenarios cause those items to behave together. There are MPM/MRL and end effector demos to run, and the user can choose from nominal case demos or a variety of failure case demos. When appropriate, there are even some random elements present to keep these demonstrations interesting.

The tutorial was found to be an excellent training mechanism for new users. It is simple to use and requires no outside expert intervention or guidance. The DESSY developers feel that a tool such as this provides an ideal training mechanism for new software users.

7.5.2 The Tutorial as Documentation

As well as being an effective training tool, the tutorial was useful in documenting both displays and test scenarios. Because there is no DESSY users guide, the tutorial was important for initial user reference, as well as for documenting display behaviors and storing test procedures. Because it can be easily viewed by a variety of end users, it served as a means for user verification of test procedures and expert system conclusions. As a documentation forum, the tutorial and cue card together are a more than adequate replacement for a users guide and are in many ways superior.

7.5.3 Tutorial Design

One of the primary advantages of the DESSY tutorial was that it was simple to create. There were two main areas where work had to be done. First, the pages of the page-turner had to be constructed and appropriate text written. This included making sure the users could only turn to pages which were appropriate from their current position within the tutorial. Second, because the screen was to be constructed piece by piece and users were to interact with each piece, an altered copy of the expert system screen had to be stored in which each appropriate screen section could be viewed separately. User input buttons were created for each section to give the tutorial browser control of these screen displays. The remaining parts of the expert system, however, remained unchanged, and the tutorial was simply added as a layer on top of the intelligent system.

7.6 Demonstrations

Builders of software applications are frequently asked to provide demonstrations along with their delivered products. This may be thought of as part of the documentation process and might even provide introductory training to users and their supervisors. In the DESSY project, numerous demonstrations have been given and a recommended demo approach has evolved. The following sections contain the DESSY developers' view of effective demonstration techniques and discuss how to organize software to provide effective demonstrations.

7.6.1 Effective Demonstrations

The two basic sources of scenarios for demonstrations are prerecorded data that can be played back during the demo and user-written demonstration procedures, called "canned" demos. DESSY demos typically used user-written procedures because this provided maximum flexibility during the demo. For example, complicated scenarios with rapidly changing data can be broken into sequential steps, each step played at a time. Once the scenario is understood, the entire demo sequence is played. This gives the audience a better understanding of complex scenarios. In addition, the selection of demo procedures is limited only by what the developers have constructed, thus providing a wide range of demo alternatives.

Using playback data for demonstrations has advantages as well, primarily because this is more realistic than canned demo procedures. Viewers may see the updating of the telemetry clocks corresponding to data changes, which gives the demo increased credibility. The disadvantage of these demos is that manipulation of the playback software may be cumbersome. Choices of demonstration scenarios will also be limited by available playback files.

Whichever demonstration method is used, demos should begin with a brief description of the software functionality and its intended user base. A tour of the screen should then be provided with a quick description of each screen element, and any particularly interesting pop-ups should be pointed out. It is usually best to present a simple nominal case scenario first to give users an easy case to follow while you point out expert system features. This can be followed by failure scenarios that illustrate more advanced expert system capabilities. Finally, demonstrations are concluded with a few words about future project work. Because every demo is a chance to "sell" the software, it is important to provide the most complete view of the application, and look for opportunities for others to contribute to or expand on your work.

7.6.2 Organization of the Software to Support Demonstrations

Easy access to the canned demonstration procedures made giving DESSY demos go much more smoothly. DESSY contains a header with buttons providing various expert system functions. One of the buttons, labeled "Demos," leads to workspaces with a variety of demonstration choices. Included are the standard nominal and failure-related scenarios that are typically run, and access to other procedures that might become appropriate during the demonstration. These scenarios are in fact the same test cases used for knowledge base testing and in the tutorial. They are contained within a simulation module which may be optionally loaded along with the rest of the KB. Thus, through careful organization of the knowledge base, the use of test scenarios for a variety of purposes can be optimized.

7.7 The Developer-User Handover

The final stage of expert system development is the handover of the application to the user community. Although this may mark the end of the project life as you have known it, it is only the beginning of the next project phase—that of software maintenance. Continued open communication between the original developers and the software maintainers is important for the continued success of the intelligent system. Although the software may no longer primarily be your responsibility, keep in mind that its success still depends on your willingness to follow the application through its remaining life cycle and help maintain it when necessary.

The DESSY project is currently in the handover phase. The DESSY developers have met with the individuals responsible for maintaining the software in the control center environment. Although a completed product was delivered, there is still additional work to be done. For example, the control center data source is changing and the message logger currently used by the flight controllers is different from that used by DESSY. DESSY will need to be altered to accommodate both these changes, and the DESSY team will be involved. The bottom line is that the software developer's ties to the project are never entirely eliminated. Being willing to maintain those ties at least in a consulting capacity will give the application the greatest chance of success in its new environment.

The traditional waterfall model of software development perpetuates the perception that software can be static. A common misconception is that if the requirements can be adequately identified and met, there should be no further need to change the software. However, it is unrealistic to presume that anything can remain static. Job responsibilities change over time. Other software supporting the user's job performance will change. New hardware and software technology present new opportunities to support human performance even better. Further experience with a software application can reveal oversights in its design. Any of these conditions can result in a need for the application to change. If an application cannot be modified gracefully to fit the new requirements, it will need to be replaced by a new one. Continued developer ties with the user communitymake it more likely that the delivered software application can be modified gracefully to continue good support to the users.

In Section 6.7, the idea of phased certification was introduced, based on experiences with both the DESSY and RMSCO software. In these cases, there will likely be a portion of the software which has been delivered and a portion which is still under active development. While there is active development associated with a project, the lines of communication remain open so that requirements changes to the delivered portion can be detected as the user discovers them. This continued communication will help the developer to keep the application current so that it continues to support the user's needs.

Because DESSY is in the handover phase, we are likely to learn more about this topic after this document is published. Consequently, the user may wish to check the World Wide Web (WWW) page which contains an updated version of this document. This page is viewable by anyone with an Internet connection and an html viewer such as Mosaic or Netscape. The address (Uniform Resource Locator, or URL) is

http://tommy.jsc.nasa.gov/~clare/Developers_Guide/

References

Bachant, Judith and Soloway, E., "The Engineering of SCON," *Communications of the ACM*, 32:3, March 1989, pp. 311-317.

Chandrasekaran, B. and Punch, W.F. III, "Data Validation During Diagnosis: A Step Beyond Traditional Sensor Validation," Proc. AAAI-87, Seattle, WA.

Chandrasekaran, B. and Punch, W.F. III, "Hierarchical Classification: Its Usefulness for Diagnosis and Sensor Validation," *IEEE Journal on Selected Areas in Communications*, June 1988, pp. 884-891.

CLARE, Control Center Library of Applications for Reuse and Exchange, 1994.

Collins, David, "Payload Deployment and Retrieval System Overview Workbook," PDRS OV 2102, Mission Operations Directorate, Johnson Space Center, Houston, Texas, February, 1988. (JSC internal document)

Cooper, G.F, "The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks," *Artificial Intelligence* 42, 1990.

DeCoste, D, Dynamic Across-Time Measurement Interpretation, *Artificial Intelligence* 51, 1991.

Dvorak, Daniel, "Expert Systems for Monitoring and Control," AI 87-55, The University of Texas at Austin, Austin, Texas, May, 1987.

Gabrielian, A. and Franklin, M.K., "State-Based Specification of Complex Real-Time Systems," Proceedings, Real-Time Systems Symposium, Huntsville, AL, December 6-8, 1988, pp. 2-11.

Hayes-Roth, F.,Waterman, and Lenat, *Building Expert Systems*, New York:: Addison-Wesley, 1983.

Hopcroft, John E., et al., *Introduction to Automata Theory, Languages, and Computation,* New York: Addison-Wesley, 1979.

Jordan, W., Keller, K., et al, "Software Storming: Combining Rapid Prototyping and Knowledge Engineering," *Computer*, May 1989.

Land, Sherry, Malin, J., and Culp, D., "A Monitoring System with Tolerance for Real-Time Data Problems," Conference on Artificial Intelligence Applications, 1993.

Land, Sherry, Malin, J., and Culp, D., "DESSY: Making a Real-Time Expert System Robust and Useful," SOAR 1992 [Proceedings of Sixth Annual Space Operations, Applications, and Research Symposium].

Malin, Jane, Schreckenghost, D., et al., "Making Intelligent Systems Team Players: Case Studies and Design Issues," NASA Technical Memorandum 104738, September 1991.

Malin, Jane, Schreckenghost, Debra, and Thronesbery, Carroll, "Design for Interaction Between Humans and Intelligent Systems During Real-Time Fault Management," Proceedings of Fifth Annual Space Operations, Applications, and Research Symposium, Johnson Space Center, Houston, Texas, July 9-11, 1991.

Marcot, Bruce, "Testing Your Knowledge Base," *AI Expert*, August 1987, pp. 42-47.

Moore, Robert and Kramer, Mark, "Expert Systems in On-Line Process Control," Lisp Machines, Inc., Los Angeles, California., [198_].

Paasch, R.K. and Agogino, A.M., Automated Diagnosis for the Time of Flight Scintillation Array: Development of a Structural and Behavioral Reasoning System, Proc. 8th CAIA, 1992.

Padalkar, Samir, et. al., "Real-Time Fault Diagnostics," *IEEE Expert*, June 1991, pp. 75-85.

SDRS DTE Display Description Reports: Displays 3582M, 1024K,.1030E, 3592I, April 8, 1992. [Johnson Space Center internal report.]

Scarl, E.A., et al.., "Diagnosis and Sensor Validation Through Knowledge of Structure and Function," *IEEE Trans. Syst. Man Cybern.* 17 (3), 1987.

Schreckenghost, Debra , Human-Computer Interface Design Concepts (DESSY blue book), W124, Mitre, Corp., Houston, Texas, July 2, 1990.

Shontz, W.E., et al., Flight Deck Engine Advisor Final Report, NASA Contractor Report 189562, Langley Research Center, Hampton, VA., February 1992.

Simpson, William and Sheppard, J., "System Complexity and Integrated Diagnostics," *IEEE Design & Test of Computers*, September 1991, pp. 16-30.

Wellman, M.P., "Fundamental Concept of Qualitative Probabilistic Networks," *Artificial Intelligence* 44, 1990.

Appendix A

DESSY End Effector Failures

Appendix A

DESSY End Effector Failures

NOMINAL SEQUENCES
auto release
auto capture
manual release
manual capture
ee checkout (capture and release)

EE FLAG FAILURES (inconsistent talkbacks)
rigid-derigid
close-open
captured-open
extend-not-derigid
extend-rigid

EE CMDS FAILURES
release with commands alarm
capture with commands alarm
(this is the least mature area of ee fault recognition)

UNCOMMANDED RELEASE/CAPTURE
uncommanded derigid
uncommanded release
manual capture sequence with uncommanded release

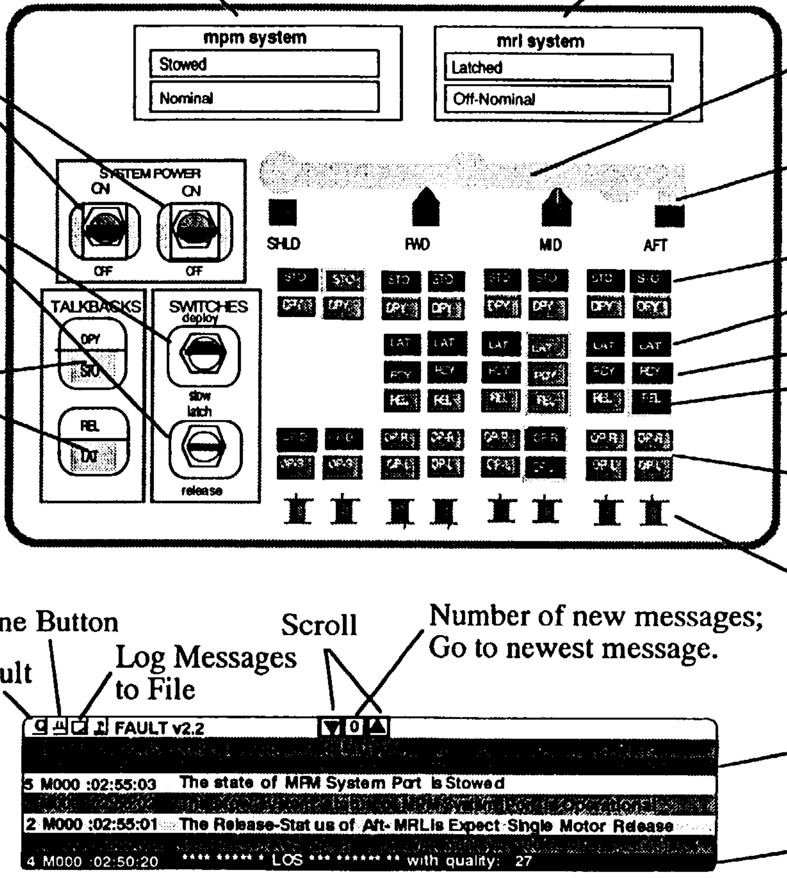
AUTO TIMEOUT FAILURES
auto release with derig timeout
auto release with snare timeout
auto release with extend timeout
auto capture with snare timeout
auto capture with rig1 timeout
auto capture with rig2 timeout
auto release commanding with no talkback effects
auto capture commanding with no talkback effects

MANUAL TIMEOUT FAILURES
manual release with derig timeout
manual release with snare timeout
manual release with extend timeout
manual capture with snare timeout
manual capture with rig1 timeout
manual capture with rig2 timeout

SEQUENCE VIOLATIONS
auto capture without limp
auto release with CLOSE ms failed on
capture with auto mode switch failed off
capture with manual mode switch failed off
release with auto mode switch failed off
release with manual mode switch failed off

Appendix B

DESSY Cue Cards



5 M000 :02:55:03 The state of MPM System Port is Stowed
2 M000 :02:55:01 The Release-Status of Aft-MRL is Expect-Single Motor Release
4 M000 :02:50:20 **** ***** * LOS *** ****** ** with quality: 27

MPM Text Display gives MPM state and status information.

MRL Text Display gives MRL state and status information.

Mech Power Command Switches. Direct telemetry. Mech Power is off. Yellow warning when command is present and power is off.

MPM and MRL Command Switches. Derived from Opstat telemetry. MPM Deploy command. No MRL command.

Talkbacks display the expected on-board talkback state using system 1 telemetry. Talkbacks reflect stowed and latched.

Arm Icon has selectable MPM(square) icons and MRL (triangle) icons.

Arm is white (shown) when RMS is RFL, grey when RMS is not RFL.

Degraded MRL

MPM's Stowed
MRL's Latched
RFL's
Yellow border indicates Failure

MPM/MRL telemetry

Opstats indicate deploy command

Opstat telemetry

Power icons (Mech Pwr+Bus Enables) indicate whether or not a motor is powered. Power icons are inactive.

XFault Message Codes

White - State
Green - Nominal Status
Yellow - Caution Status
Red - Warning Status
Blue - Command & LOS/AOS
Black - Acknowledged

Tone Button

Quit XFault

Log Messages to File

Scroll

Number of new messages; Go to newest message.

Logic Switch is off. Derived from data pattern shown.

Message acknowledged by selecting it.

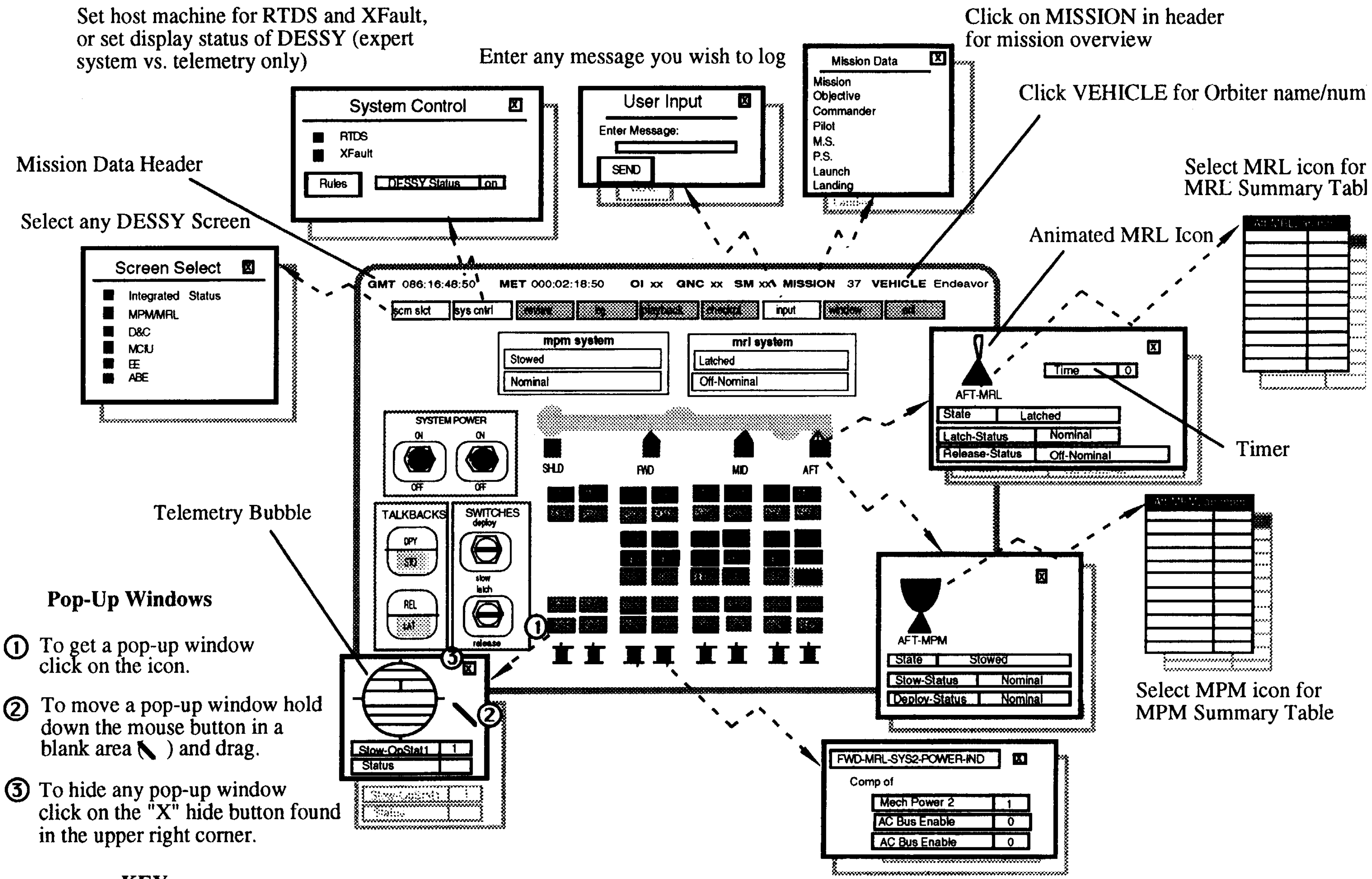
<u>Manipulator Positioning Mechanism (MPM) Status</u>

operational	off-nominal
nominal-stow	nominal-deploy
single-motor-stow	single-motor-deploy
expect-single-motor-stow	expect-single-motor-deploy
nominal-and-two-phase-motor-stow	nominal-and-two-phase-motor-deploy
potential-single-motor-stow	potential-single-motor-deploy
stow-failed	deploy-failed
stow-failed-to-initiate	deploy-failed-to-initiate
stow-command-failure	deploy-command-failed
failed-due-to-stow-indicators-failure	failed-due-to-deploy-indicators-failure

<u>Manipulator Retention Latch (MRL) Status</u>

operational	off-nominal
nominal-latch	nominal-release
single-motor-latch	single-motor-release
expect-single-motor-latch	expect-single-motor-release
nominal-and-two-phase-motor-latch	nominal-and-two-phase-motor-release
potential-single-motor-latch	potential-single-motor-release
latch-failed	release-failed
latch-failed-to-initiate	release-failed-to-initiate
latch-command-failure	release-command-failed
failed-due-to-latch-indicators-failure	failed-due-to-release-indicators-failure

revised 5/7/92



Pop-Up Windows

1. To get a pop-up window click on the icon.
2. To move a pop-up window hold down the mouse button in a blank area ( ) and drag.
3. To hide any pop-up window click on the "X" hide button found in the upper right corner.

KEY

▭	Pop-up Window
▨	MPM/MRL Window

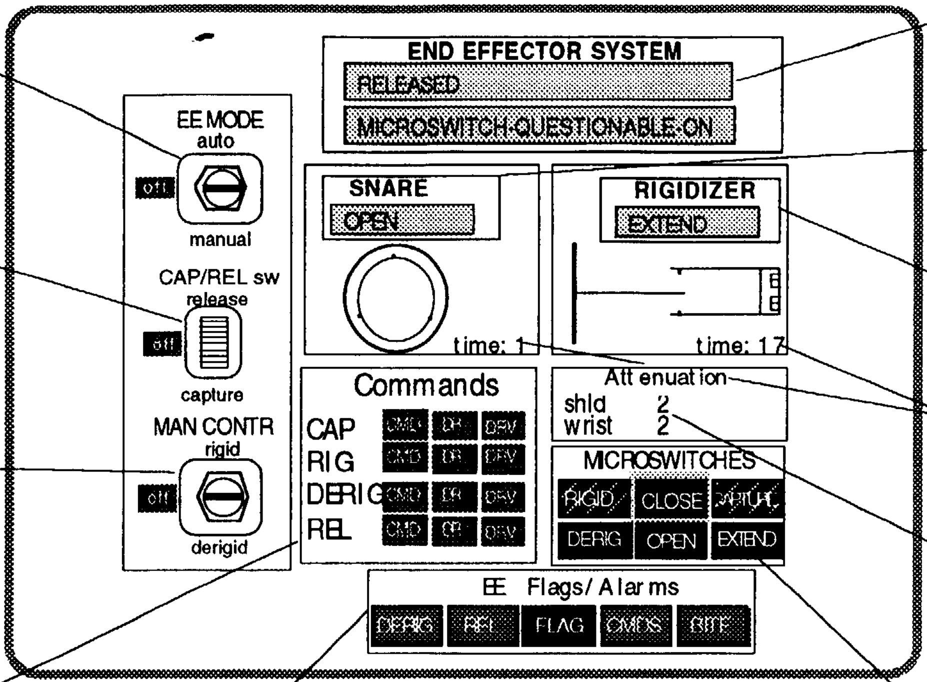
ACTIONS:

- - -	Denotes pop-up window for icon
	Click on a blank space of any window and drag the mouse.
☒	Click on the X to HIDE the window

COLOR CODES:

Status	Data State
Green - nominal	Dark Blue - active
Yellow - degraded	Light Blue - inactive
Red - failed	Orange - LOS

C-2



<u>EE Mode Switch</u>. Direct telemetry for position of switch on Shuttle control panel. Switch is off, indicated by gray background.

<u>Capture/Release Trigger</u>. Direct telemetry for trigger position on RHC. Switch is off, indicated by gray background.

<u>Manual Control Switch</u>. Direct telemetry for position of switch on Shuttle control panel. Switch is off, indicated by gray background.

<u>Command Telemetry Boxes</u>. Direct telemetry for CMDs and DRs. DRVs are "comps". All commanding is off, indicated by gray color. Represents switch on Shuttle control panel.

<u>EE Flags/Alarms Telemetry Boxes</u>. Direct telemetry for alarms annunciated from onboard Shuttle software. The EE FLAG alarm (red) is annunciated due to the fact that both the CLOSE and OPEN microswitches are on simultaneously.

<u>End Effector System Text Display</u>. State and status information derived fro telemetry and expected sequences.

<u>Snare State Display</u>. Both text and graphic display of the current state of the snare mechanism.

<u>Rigidizer State Display</u>. Both text and graphic display of the current state of the rigidizer mechanism.

<u>Timer Displays</u>. Show the times for the snare and rigidizer mechanisms separately for the most recent sequence.

<u>Attenuation Level.</u> Direct telemetry fo attenuation of power to the shoulder and wrist of the Shuttle arm. Provided to enable verification of power attenuati during capture sequences.

<u>EE Microswith Telemetry Boxes</u>. Direc telemetry for end effector microswitche The CLOSE telemetry box has a yellow border indicating its status (health) is questionable.

DESSY End Effector State & Status Values

<u>End Effector System</u>

States	Statuses
initialized	operational
captured	timeout
released	uncommanded-release
no-pin	uncommanded-derig
capturing	ee-flag-alarm
releasing	ee-commands-alarm
aborted-capture	eeeu-bite-alarm
aborted-release	capturing-without-limp
	mode-switch-auto-failed-off
	mode-switch-manual-failed-off

<u>Snare</u>

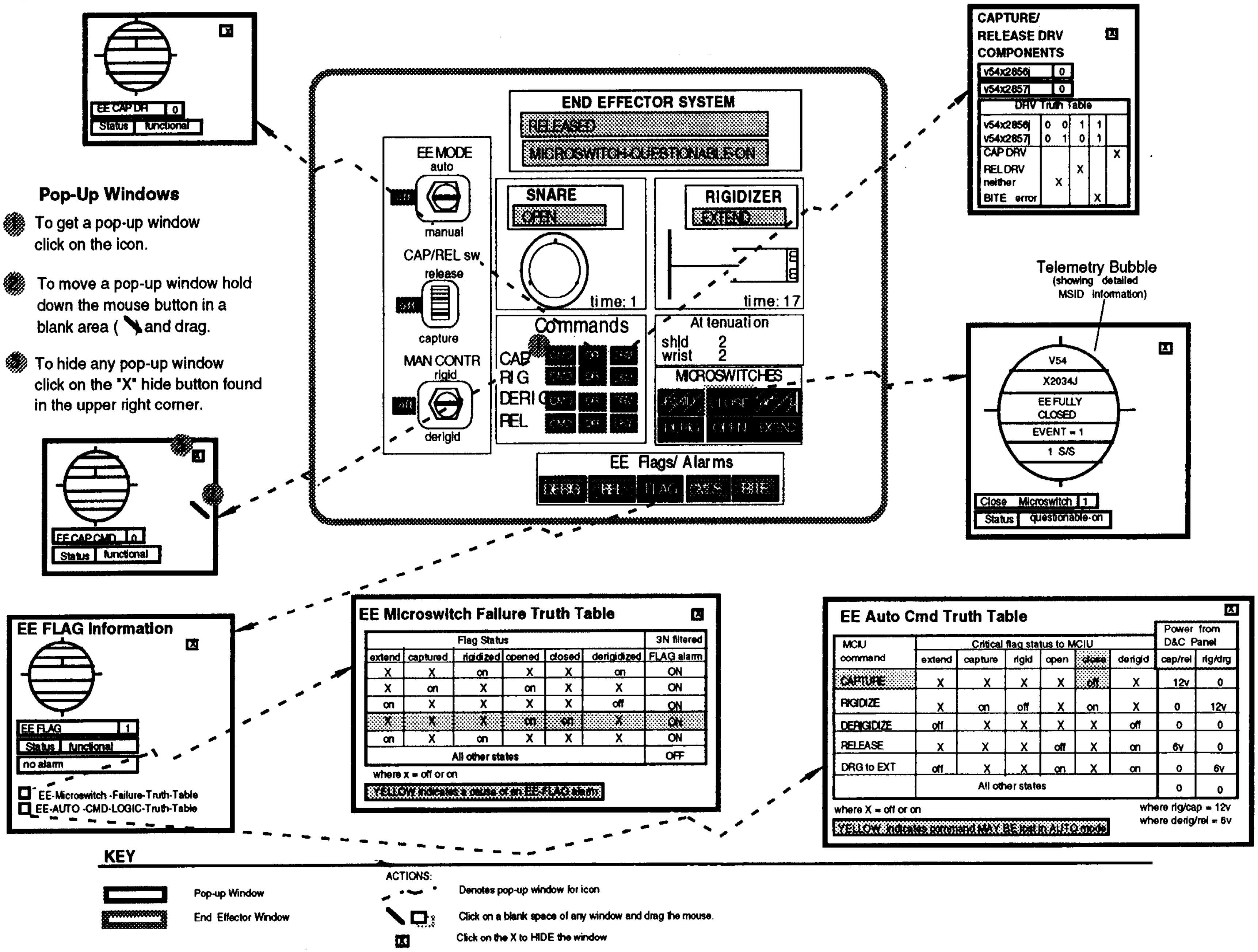
States	Statuses
initialized	operational
open	nominal
closing	snare-opening-timeout
opening	snare-closing-timeout
closed-no-cap	microswitch-questionable-on
captured	microswitch-questionable-off
closed	uncommanded-release
aborted-close	
aborted-open	

<u>Rigidizer</u>

States	Statuses
initialized	operational
rigid	nominal
derigidizing	rigidizing timeout
derigid	derigidizing timeout
extending	extending timeout
extend	microswitch questionable on
rigidizing1	microswitch questionable off
rigidizing2	uncommanded derig
aborted-rigidize	
aborted-derig	
aborted-extend	

Color Codes

blue (red for Flags/Alarms) - active
gray or barber-pole - inactive
orange border - loss of data quality
yellow border - questionable status



Pop-Up Windows

1. To get a pop-up window click on the icon.
2. To move a pop-up window hold down the mouse button in a blank area (and drag.
3. To hide any pop-up window click on the "X" hide button found in the upper right corner.

CAPTURE/ RELEASE DRV COMPONENTS

v54x2856j	0
v54x2857j	0

DRV Truth Table

v54x2856j	0	0	1	1	
v54x2857j	0	1	0	1	
CAP DRV					X
REL DRV			X		
neither		X			
BITE error				X	

EE Microswitch Failure Truth Table

Flag Status						3N filtered
extend	captured	rigidized	opened	closed	derigidized	FLAG alarm
X	X	on	X	X	on	ON
X	on	X	on	X	X	ON
on	X	X	X	X	off	ON
▨ X	▨ X	▨ X	▨ on	▨ on	▨ X	▨ ON
on	X	on	X	X	X	ON
All other states						OFF

where x = off or on

YELLOW indicates a cause of an EE-FLAG alarm

EE Auto Cmd Truth Table

MCIU command	Critical flag status to MCIU						Power from D&C Panel	
	extend	capture	rigid	open	▨ close	derigid	cap/rel	rig/drg
▨ CAPTURE	X	X	X	X	▨ off	X	12v	0
RIGIDIZE	X	on	off	X	on	X	0	12v
DERIGIDIZE	off	X	X	X	X	off	0	0
RELEASE	X	X	X	off	X	on	6v	0
DRG to EXT	off	X	X	on	X	on	0	6v
	All other states						0	0

where X = off or on

where rig/cap = 12v
where derig/rel = 6v

YELLOW indicates command MAY BE lost in AUTO mode

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE July 1995	3. REPORT TYPE AND DATES COVERED Technical Memorandum

4. TITLE AND SUBTITLE A Guide to Developing Intelligent Monitoring Systems	5. FUNDING NUMBERS
6. AUTHOR(S) Sherry A. Land; Jane T. Malin; Carroll Thronesbery*; Debra L. Schreckenghost*	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Lyndon B. Johnson Space Center Automation, Robotics, and Simulation Division Houston, Texas 77058	8. PERFORMING ORGANIZATION REPORT NUMBERS S-790
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, D.C. 20546-0001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER TM-104807

11. SUPPLEMENTARY NOTES

*Metrica Inc., Houston, Texas

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited Available from the NASA Center for AeroSpace Information (CASI) 800 Elkridge Landing Road Linthicum Heights, MD 21090-2934 (301) 621-0390 Subject Category: 63	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This reference guide for developers of intelligent monitoring systems is based on lessons learned by developers of the DEcision Support SYstem (DESSY), an expert system that monitors Space Shuttle telemetry data in real time. DESSY makes inferences about commands, state transitions, and simple failures. It performs failure detection rather than in-depth failure diagnostics. A listing of rules from DESSY and cue cards from DESSY subsystems are included to give the development community a better understanding of the selected model system. The G-2 programming tool used in developing DESSY provides an object-oriented, rule-based environment, but many of the principles in use here can be applied to any type of monitoring intelligent system. The step-by-step instructions and examples given for each stage of development are in G-2, but can be used with other development tools. This guide first defines the authors' concept of real-time monitoring systems, then tells prospective developers how to determine system requirements, how to build the system through a combined design/development process, and how to solve problems involved in working with real-time data. It explains the relationships among operational prototyping, software evolution, and the user interface. It also explains methods of testing, verification, and validation. It includes suggestions for preparing reference documentation and training users.

14. SUBJECT TERMS Expert Systems, Artificial Intelligence, Computer Systems Programs, Monitors, Real Time Operation, Manuals	15. NUMBER OF PAGES 103
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT