SRI International

A SYSTEM FOR REASONING IN DYNAMIC DOMAINS: FAULT DIAGNOSIS ON THE SPACE SHUTTLE

Technical Note 375

January 1986

By: Michael P. Georgeff
Amy L. Lansky
Artificial Intelligence Center
Computer Science and Technology Division

APPROVED FOR PUBLIC RELEASE:
DISTRIBUTION UNLIMITED

This research has been made possible in part by the National Aeronautics and Space Administration under Contract NAS2-11864, SRI Project 7268.

The views and conclusions contained in this paper are those of the authors and should not be interpreted as representative of the official policies, either expressed or implied, of the National Aeronautics and Space Administration or the United States Government.

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Contents

1 Description of the Problem **5**

1.1 Introduction 5

1.2 Fault Isolation and Diagnosis 7

2 Possible Technologies **13**

2.1 Conventional Programming Languages 13

2.2 Conventional Expert Systems 15

2.3 A Simple Example 17

3 Procedural Knowledge **21**

3.1 Representing Procedural Knowledge 21

3.2 Using Procedural Knowledge 25

3.3 Procedural Expert Systems 29

4 RCS Application **34**

4.1 The System 34

4.1.1 The System Data Base 35

4.1.2 Behaviors and Goals 36

4.1.3 Knowledge Areas 37

4.1.4 User Interface and Menu System 39

4.2 Space Shuttle Example 42

4.2.1 The RCS Data Base 46

4.2.2 The JET-FAIL-ON KA 47

5 Reasoning about Procedures **60**

5.1 Metalevel Reasoning 60

5.2 Reasoning about Complex Goals 62

6 Conclusions **65**

A Sample Knowledge Base for the RCS **67**

A.1 Glossary of Identifier Prefixes 67

A.2 RCS State Description (Initial Data base) 68

A.2.1 Top Level Reactant Control Systems 68

A.2.2 Basic Components of Forward RCS 68

A.2.3 Helium Pressurization System Of Forward RCS 70

A.2.4 Propellant Distribution System Of Forward RCS 71

A.2.5 Thruster System Of Forward RCS 73

A.3 Knowledge Areas 75

ABSTRACT

This report describes a reactive system for reasoning about and performing complex tasks in dynamic environments. A powerful and theoretically sound scheme for representing and reasoning about actions and processes has been devised. The representation is sufficiently rich to describe the effects of arbitrary sequences of tests and actions, and the reasoning mechanism provides a means for directly using this knowledge to attain desired goals or to react to critical situations. A declarative semantics for the representation has been constructed that allows a user to specify *facts* about behaviors independently of context. An operational semantics has also been defined that shows *how* these facts can be used by a system to achieve its goals. Possession of both a declarative and an operational semantics provides the system with the ability to reason about complex actions, to explain its reasoning to others, to cope with modifications to its environment, and to be amenable to verification. The system also includes powerful metalevel reasoning capabilities, using the same formalism for representing this knowledge as for object-level knowledge. A practical implementation of such a system has been constructed and applied to some crucial problems in the automation of space operations.

Chapter 1

Description of the Problem

1.1 Introduction

In dynamic problem domains, such as the operation of space vehicles, active intelligent systems need to be able to represent and reason about actions and how those actions can be combined to achieve given goals. Within AI, there have been two approaches to these issues, with a somewhat poor connection between them. In the first category, there is work on theories of action – i.e., on what constitutes an action per se [2,18,22]. This research has focused mainly on problems in natural-language understanding concerned with the meaning of action sentences. Second, there is work on planning – i.e., the problem of constructing a plan by searching for a sequence of actions that will yield a given goal [3,8,26,27,29,30,31,33].

Surprisingly, almost no work has been done in AI concerning the execution of preformed plans or procedures – yet this is the almost universal way in which humans go about their day-to-day tasks, and probably the only way other creatures do so. To actually search the space of possible future courses of action, which is the sole means by which most AI planning systems reason about action, is relatively rare.

On the other hand, so-called "expert systems" have been applied primarily to static problem domains where the interaction between the system and the environment is highly constrained. They provide no representational scheme for actions or processes, and only limited means for reasoning about procedural forms of knowledge.

In this report we describe a system for reasoning about and performing complex tasks in dynamic environments and develop a scheme for *explicitly* representing and reasoning about the kinds of procedural knowledge typical of these problem domains. The knowledge representation is sufficiently rich to describe the effects of arbitrary sequences of tests and actions, and the inference mechanism provides a means of directly using this knowledge to accomplish desired operational goals. Furthermore, the knowledge representation has a declarative semantics that provides for incremental modifications of the system, rich explanatory capabilities, and verifiability. The scheme also provides a mechanism for reasoning *about* the use of this knowledge, thus enabling the system to choose effectively among alternative courses of action. Systems that are based on this scheme are called *procedural expert systems* [14].

The problem we will consider is the automation of the operation of complex physical systems; in particular, those involved in space operations. Such operations include subsystem monitoring, preventive maintenance, malfunction handling, fault isolation and diagnosis, communications management, maintenance of life support systems, power management, monitoring of experiments, servicing of satellites, testing and deployment of payloads and upper stages, orbital-vehicle operations, orbital construction and assembly, and control of extraterrestrial rovers. Automation of these operations can be expected to improve mission productivity and safety, increase versatility, lessen dependence on ground systems, and reduce demands on crew involvement in system operations.

Systems based on conventional automation techniques are unlikely to be effective in most of these applications [25]. Such systems are usually very inflexible and unresponsive to the skill level of the technicians using them. The control and diagnostic procedures that are used cannot be matched to the exigencies of the current situation nor can they cope with reconfiguration or modification of the physical systems under test. Performance of tasks cannot be guided by useful advice from technicians and, when a given task cannot be performed, no explanation is given as to the cause of failure. Because these systems perform a prescribed sequence of tests and actions, they cannot utilize knowledge of a particular situation to focus attention on more likely trouble spots. Consequently, real-time performance is highly unsatisfactory. Furthermore, the cost of developing the software is substantial and time to maturation is excessive.

1.2 Fault Isolation and Diagnosis

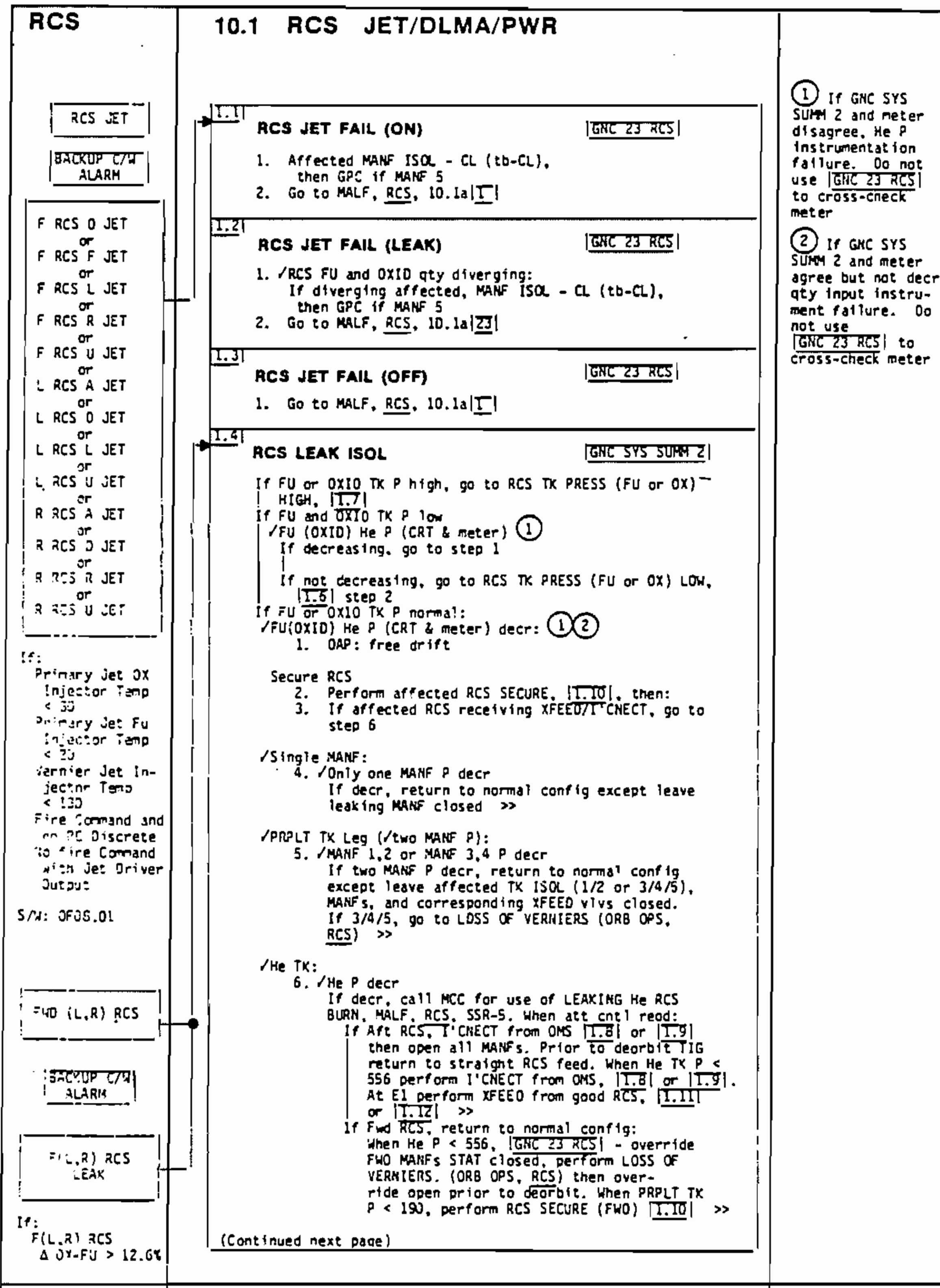
As mentioned above, there are many space operations that require complex reasoning capabilities, including command and control, monitoring and control of experiments, management of various subsystems, and the handling of system malfunctions. The last is of particular importance. On the space shuttle (STS), for example, malfunction handling currently places acute demands on crew and requires a very high level of ground support and manpower. Given the proposed increase in the frequency of shuttle flights during the next few years, such high levels of crew involvement and ground support are patently unacceptable.

A major consideration in choosing a reasoning system suitable for such an application is that much of the domain knowledge is represented procedurally. The procedural nature of this knowledge is critically important not only with regard to the conclusions drawn but also to the safety and efficiency of the operation.

Most of this knowledge is set down in the operational procedures for the various subsystems of the spacecraft. Other knowledge is part of the general technical expertise of mission controllers and astronauts. In addition, there are various constraints that must not be violated in executing the procedures, such as adherence to flight rules and the avoidance of potentially harmful interactions with other subsystems.

The operational procedures include extensive instructions describing various tests to perform and actions to carry out, dependent on the results of previous tests and actions. Most are written as a sequence of steps in English-like language, including conditional statements, "go to" statements, and transfers to named procedures. Many control constructs are unusual, such as procedures that have multiple entry points, are interrupt-driven (i.e., can be invoked on a given condition), or are dynamically modifiable (e.g., "do procedure P except ..."). Sample portions of the malfunction handling procedures for the reaction control system (RCS) on the space shuttle are given in Figures 1.1, 1.2 and 1.3. As can be seen, the procedures are extremely complex.

Some of the procedures that are used to establish particular conditions or draw certain conclusions would be invalid, were they not carried out in the order specified (i.e., the results are *context-dependent* or *time-dependent*). For example, the conclusions to be drawn after a hot fire of the RCS depend entirely on the context of the particular maintenance procedure being executed: if the desired response is obtained,



RCS

10.1 RCS JET/DLMA/PWR

RCS JET

BACKUP C/W ALARM

F RCS D JET or F RCS F JET or F RCS L JET or F RCS R JET or F RCS U JET or L RCS A JET or L RCS D JET or L RCS L JET or L RCS U JET or R RCS A JET or R RCS D JET or R RCS R JET or R RCS U JET

If:
Primary Jet OX Injector Temp < 30
Primary Jet Fu Injector Temp < 20
Vernier Jet Injector Temp < 130
Fire Command and no PC Discrete
No Fire Command with Jet Driver Output

S/W: OF0S.01

FWD (L,R) RCS

BACKUP C/W ALARM

F(L,R) RCS LEAK

If:
F(L,R) RCS
Δ OX-FU > 12.6%

1.1 RCS JET FAIL (ON) |GNC 23 RCS|

1. Affected MANF ISOL - CL (tb-CL), then GPC if MANF 5
2. Go to MALF, RCS, 10.1a |1|

1.2 RCS JET FAIL (LEAK) |GNC 23 RCS|

1. √RCS FU and OXID qty diverging:
   If diverging affected, MANF ISOL - CL (tb-CL), then GPC if MANF 5
2. Go to MALF, RCS, 10.1a |23|

1.3 RCS JET FAIL (OFF) |GNC 23 RCS|

1. Go to MALF, RCS, 10.1a |1|

1.4 RCS LEAK ISOL |GNC SYS SUMM 2|

If FU or OXID TK P high, go to RCS TK PRESS (FU or OX) HIGH, |1.7|
If FU and OXID TK P low
  √FU (OXID) He P (CRT & meter) (1)
  If decreasing, go to step 1
  If not decreasing, go to RCS TK PRESS (FU or OX) LOW, |1.6| step 2
If FU or OXID TK P normal:
√FU(OXID) He P (CRT & meter) decr: (1)(2)
  1. DAP: free drift

Secure RCS
  2. Perform affected RCS SECURE, |1.10|, then:
  3. If affected RCS receiving XFEED/I'CNECT, go to step 6

√Single MANF:
  4. √Only one MANF P decr
     If decr, return to normal config except leave leaking MANF closed >>

√PRPLT TK Leg (√two MANF P):
  5. √MANF 1,2 or MANF 3,4 P decr
     If two MANF P decr, return to normal config except leave affected TK ISOL (1/2 or 3/4/5), MANFs, and corresponding XFEED vlvs closed.
     If 3/4/5, go to LOSS OF VERNIERS (ORB OPS, RCS) >>

√He TK:
  6. √He P decr
     If decr, call MCC for use of LEAKING He RCS BURN, MALF, RCS, SSR-5. When att cntl reqd:
       If Aft RCS, I'CNECT from OMS |1.8| or |1.9| then open all MANFs. Prior to deorbit TIG return to straight RCS feed. When He TK P < 556 perform I'CNECT from OMS, |1.8| or |1.9|. At EI perform XFEED from good RCS, |1.11| or |1.12| >>
       If Fwd RCS, return to normal config:
         When He P < 556, |GNC 23 RCS| - override FWD MANFs STAT closed, perform LOSS OF VERNIERS. (ORB OPS, RCS) then override open prior to deorbit. When PRPLT TK P < 190, perform RCS SECURE (FWD) |1.10| >>

(Continued next page)

(1) If GNC SYS SUMM 2 and meter disagree, He P instrumentation failure. Do not use |GNC 23 RCS| to cross-check meter

(2) If GNC SYS SUMM 2 and meter agree but not decr, qty input instrument failure. Do not use |GNC 23 RCS| to cross-check meter

Figure 1.1: RCS Malfunction Procedure 10.1

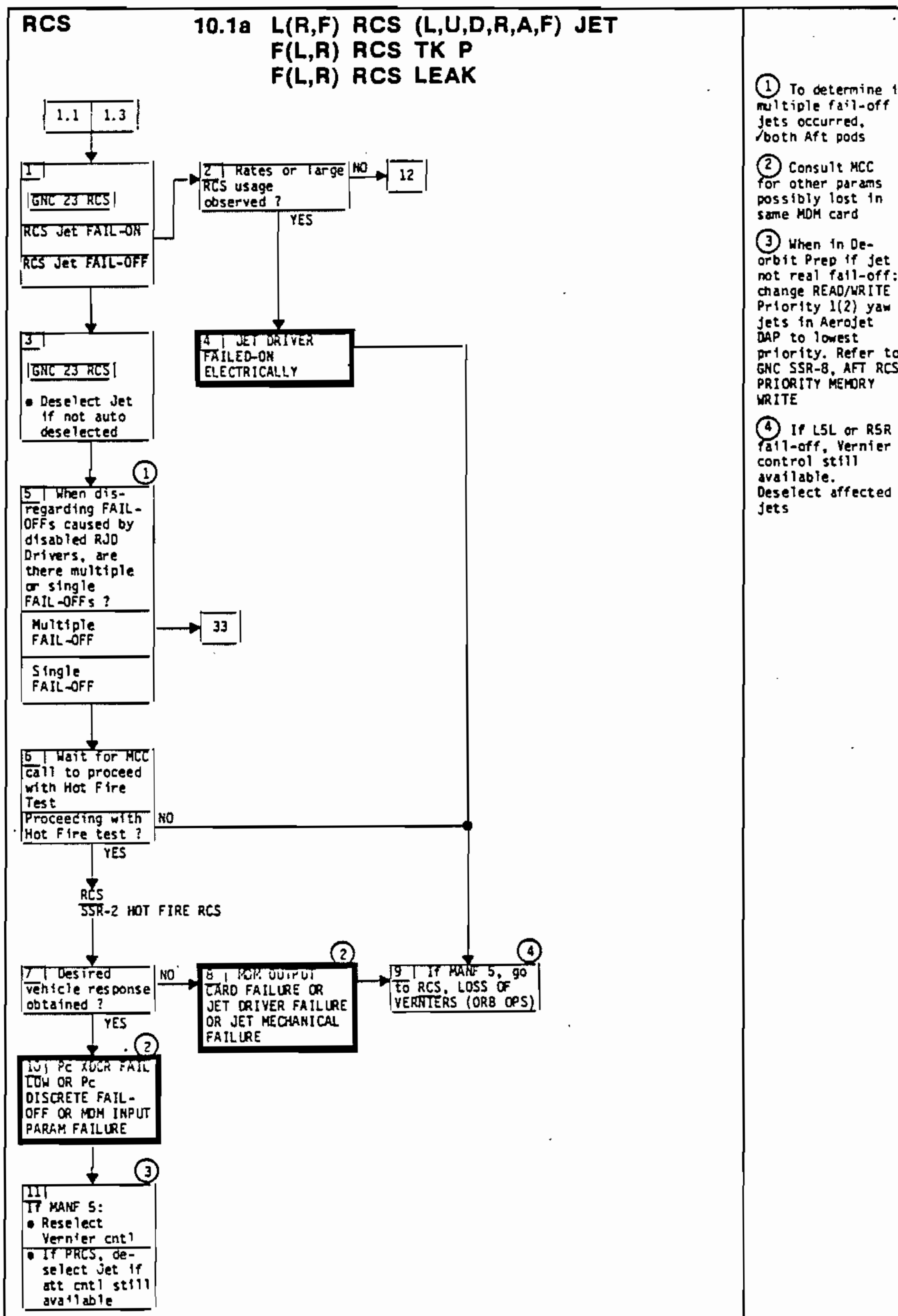


Figure 1.2: RCS Malfunction Procedure 10.1a

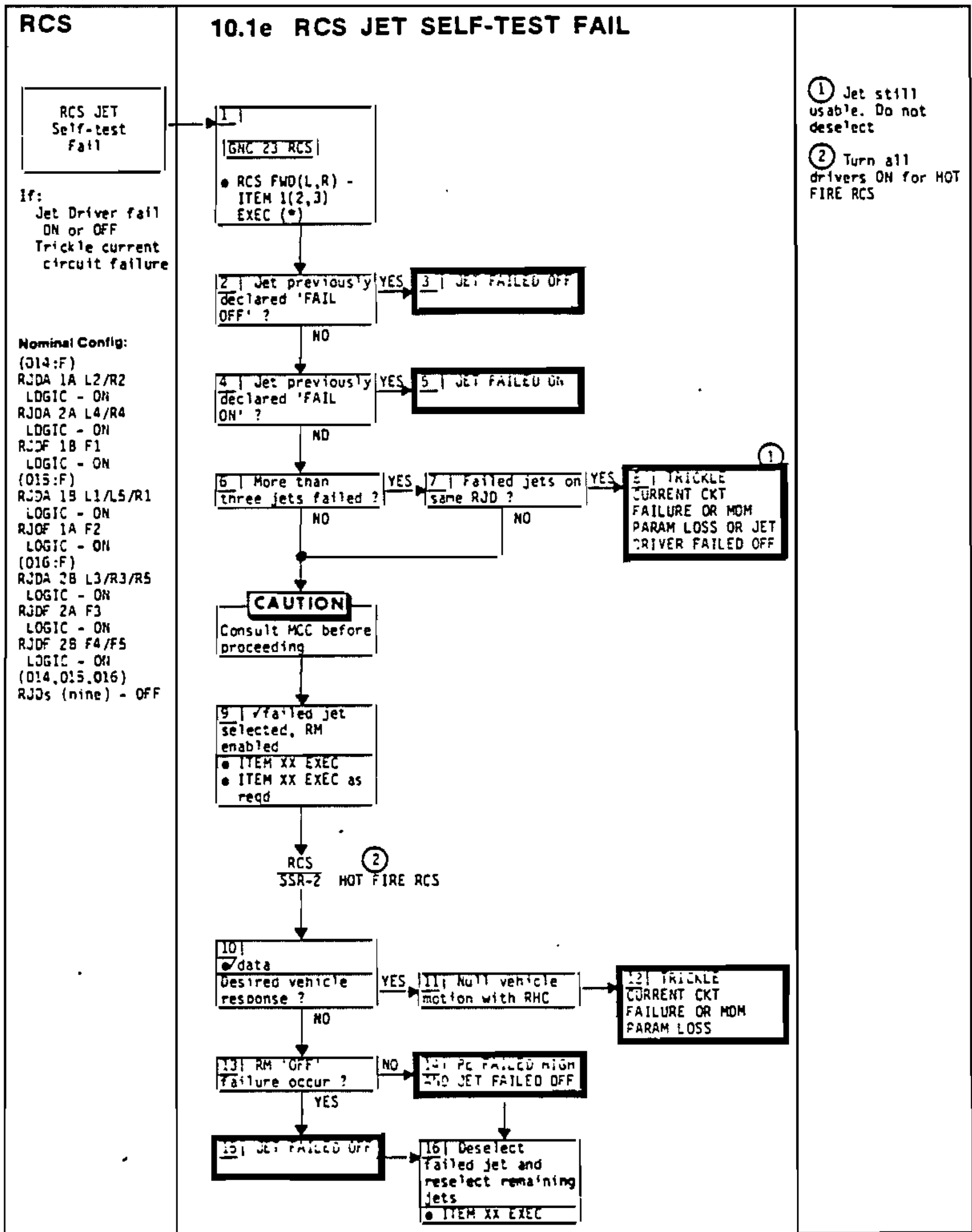


Figure 1.3: RCS Malfunction Procedure 10.1e

then, in the context of Procedure 10.1a (Figure 1.2) we could conclude that either a Pc (processor) or an MDM input parameter had failed, whereas the same observations in the context of Procedure 10.1e (Figure 1.3) would indicate a TRICKLE CURRENT circuit failure or an MDM parameter loss.

The ordering of actions and tests may not only influence the conclusions drawn, but can also have a vital effect on the entire mission. For example, in Figure 1.1, if the aft RCS is affected (Step 6, Block 1.4), all manifolds must be opened *after* interconnection with the orbital maneuvering system (OMS). If this were not done, the result could be catastrophic.

Other procedures reflect ease of maintenance, or trade-offs between the likelihood that a particular component is faulty and the ease with which it can be examined or replaced. For example, it might not be *necessary* to check the setting of one valve switch at the same time the setting of another is being examined, but, if both switches are adjacent to each other, it is *sensible* to do so.

The procedures are also designed so that the system is "made safe" prior to any attempt at fault isolation and diagnosis. Furthermore, even as the system is being brought to safety, the ordering of actions and tests must often be done in a way that ensures against loss of critical information regarding the cause of the failure.

Most of these procedures are applied to satisfy some particular goal, such as to isolate a fault in some subsystem or to determine whether or not some condition holds. However, some procedures, especially those of a precautionary nature, need to be invoked whenever a certain condition is observed. For example, the mission control center (MCC) must be advised before *any* hot fire of the jets in the RCS. Sometimes a procedure will be *primed* for invocation at a particular time as a result of some other procedure's having been executed. For example, in Figure 1.1, Step 6, if the OMS has been interconnected with the RCS, the systems have to be restored to straight feed, and various other measures taken, prior to deorbit.

Not all the domain knowledge is represented procedurally, some being in the form of general rules about the state of the system. Flight rules, for example, are often given in this form. A typical sample of such knowledge might be

> An internal OMS or RCS leak resulting in the violation of minimum thermal operating constraints is cause for a deorbit delay.

In about 10 to 20 percent of cases, no malfunction procedures are appropriate. It is then necessary for mission controllers, engineers, and astronauts to devise a test from "first principles." Much of this additional expert knowledge is also procedural in nature, although the procedures are often based on functional considerations rather than being related to a specific spacecraft system. For example, to isolate a fault in an electrical system, a typical procedure is the feed-device-ground strategy [6]: the expert focuses on the device, considers its input and output behavior, tests it by using alternate feeds and grounds, and then, depending on the outcome, moves along the feed or ground chain to another device. A similar method can be used for fault isolation in hydraulic systems. However, much of the reasoning required in constructing tests from "first-principles" involves an extensive understanding of physical systems and is currently beyond the capacity of any automatic reasoning system.

Skilled astronauts and mission controllers also know how best to apply their knowledge, such as when to terminate a diagnostic test if some particularly unusual fact suggests an alternative hypothesis or a mission-critical condition arises that requires immediate attention. Such utilitarian knowledge, often called *metalevel* knowledge [4], is very important for effective practical reasoning [10].

Chapter 2

Possible Technologies

In this chapter we examine some of the possible approaches to automating the malfunction handling procedures for space vehicles.

2.1 Conventional Programming Languages

One possible approach to automating maintenance procedures is to employ conventional programming techniques. However, this entails a number of serious problems.

One problem is that the order of task execution within a program is determined entirely by the control structure of its code. This renders such systems unresponsive to unanticipated external events and very inflexible. In space operations this is a critical deficiency, as it is essential that the system be able to respond appropriately to newly perceived data or changing goals.

Another problem is that conventional programming languages use *arbitrary* names for the procedures, tasks, and actions that are to be performed. That is, the various subroutine names and input/output commands serve merely to identify particular procedures, and are not descriptive of the goals or conditions that the procedures aim to achieve or test. This has at least three serious consequences.

The first is that the system loses its robustness and potential for change. For example, there may be many ways to normalize tank pressure in the RCS, each of which has a certain utility in different contexts. Yet a call to Procedure 1.7 (see Figure

1.1), or any other subroutine name, will invoke only one of these (i.e., the one so named), irrespective of the context. And the addition of another (perhaps better) pressure normalization procedure will go unnoticed by the system (unless one digs into the code and replaces all calls to Procedure 1.7 with calls to the new procedure). Worse yet, the deletion of Procedure 1.7, or an inadvertent renaming, could cause the system to fail, despite the fact that other pressure normalization procedures may be available for use.

The second consequence of invoking actions by name is that one loses reasoning and explanatory capabilities. Thus, for example, if the system is asked why it is performing Procedure 1.7, the best it can respond is that it is required for RCS-LEAK-ISOL Procedure 1.4 whenever the tank pressure is found to be high. But the user does not know what Procedure 1.7 was intended to achieve, and thus cannot use any expertise to OK the procedure or revise it when necessary. The situation becomes worse as procedures become larger. The user has little idea as to the purpose of the tests and actions within the procedure and quickly loses understanding of what the system is trying to achieve and how it is attempting to do so.

Equally important, the system itself cannot reason about tasks and decide how they can be combined to accomplish composite goals. For example, the system could not reason that a conjunctive goal (such as isolating a jet failure fault *and* opening the shuttle bay doors) could be realized by trying to perform one task before the other, or perhaps by interleaving the tasks with one another.

The third consequence is the most serious. The problem is that, by giving the actions arbitrary names, it is not possible to determine the validity of a procedure independently of the other procedures invoked by it. For example, the validity of Procedure 1.4 (suitably coded in a programming language) will depend on the definition of Procedure 1.7 (among others), which in turn will depend on the definitions of the procedures it calls, and so on. As the actual calling sequence will vary from one problem to the next, it is extremely difficult to verify the correctness of the system. Therefore, one could not be certain that a situation would never arise in which some particular procedure is improperly invoked, possibly with disastrous consequences.

In summary, one would prefer to be able to specify procedures in terms of the desired sequences of *goals* to be achieved – i.e., to specify *what* is desired at each point in the procedure – so that the *system itself* can reason about how best to attain these

goals given the current circumstances. In contrast, the use of calls to named procedures forces one to choose a particular way of realizing each goal at program creation time, rather than allowing this to be dynamically determined on the hasis of the current situation. More importantly, the use of named procedures also leaves the actual goal or intention unspecified and thus inaccessible to reason and explanation.

There are numerous other problems, such as the complexity of the control constructs, the use of dynamic procedure modification, and the priming of tasks for later invocation, that complicate further the employment of conventional programming languages for representing the malfunction handling-procedures for space vehicles.

2.2 Conventional Expert Systems

Another approach to representing malfunction-handling procedures is to use the rule- or frame-based representations utilized by most current expert systems (such as KEE, ART, and S1)[1,7,17,28]. However, these representations are not well suited to dynamic problem domains, where much of the knowledge is *procedural* in nature. Indeed, the formalisms try to avoid any notion of "procedure."

The major problem in using these systems is that of capturing the context in which tests and actions are performed. Because the means of fault isolation for the RCS is procedural, the various tests and actions have diverse outcomes that have different implications in different contexts. The only way to represent this in a rule-based formalism is to keep track of the procedural context by the use of "control conditions."

For example, rules expressing knowledge of the system would have to include information about the current control point and procedure, such as

> *If* [the data base contains] "at Control Point 1.1" and "in Block 1.4" and " in Procedure 10.1" and "observed pressure decreasing" *then* [add to the data base] "not at Control Point 1.1" and "at Control Point 1.2"

Clearly, this becomes very clumsy, reduces efficiency, and nullifies most of the desired properties of an expert system. In essence, the rule-based approach makes things implicit that should be explicit (i.e., the flow of control) and makes things explicit that should be implicit (i.e., the context).

With the addition of the "control conditions" necessary to represent procedural information, extensibility and robustness are lost; each control condition must be unique and should not be used by any rule other than the one for which it was intended. Explanatory capability is poor, as there is no direct access to the entire procedure; each rule must be explicated in isolation – with no satisfactory explanation offered for the meaning or use of the control conditions. Moreover, the validity of a rule containing a control condition depends on the rule or rules that inserted that control condition into the data base, which in turn depend on the rules that inserted their control conditions into the data base, and so on. Again, one could never be certain that a rule would not be invoked unexpectedly, with perhaps catastrophic effects. Furthermore, it is not possible to reason about a procedure as a whole – for example, to assess its usefulness or criticality in a given situation.

In this respect, the popular view that rule-based systems are intrinsically modular is a myth. Modularity is useful only to the extent that it captures some semantic whole, some independent piece of information. While accepted programming methodology encourages the subdivision of programs into subroutines, few people would suggest that all subroutines should be one-line pieces of code.

Similarly, it is worth reflecting on why recipe books, maintenance manuals, and descriptions of interpreters for expert systems are never given in rule form. The reason is obvious: such an approach would complicate things to absolutely no advantage. The subroutine (recipe, maintenance procedure, etc.) is intended to capture a useful functional entity; to subdivide it into meaningless parts would be counterproductive. Nor, by the same token, is there any purpose in arbitrarily subdividing a method or procedure used by some domain expert into individual but dependent rules.

Experience in trying to apply conventional expert systems to problems in fault diagnosis and maintenance has shown that expert knowledge is often procedural in nature; a number of expert systems therefore provide some facilities for representing procedures (e.g., Centaur [1] and ART). In most cases, however, such procedures are represented simply by LISP code (or some equivalent) that can be invoked via the data base. The procedures are *ad hoc* additions, have limited control constructs, cannot be reasoned about, and cannot be interrupted on the basis of newly observed data or newly established goals.

2.3 A Simple Example

To examine some of the difficulties in using current expert systems for problems of this kind, let us take a very simple example. It is interesting that, even in this elementary case, an adequate representation of procedural knowledge proves to be crucially important.

The example we shall consider is the mechanism for removing CO_2 in the environmental control and life support system (ECLSS) of the proposed space station. This is to be accomplished by operating a number of fuel cell arrays out-of-limits, where, instead of producing useful power, they absorb CO_2.[1]

A single module consists of an array of fuel cells. A stream of hydrogen flows *across* the array (in serial), passing from one cell to the next. The contaminated air flows *through* the cells (in parallel), entering one end and exiting the other. That is, unlike the hydrogen, the air does not flow from one fuel cell to another. If the fuel cells are operating correctly, most of the CO_2 has been removed by the time the air has passed through the module. A cooling system also operates to keep the module within an appropriate temperature range.

One of the primary symptoms of a module malfunction is a loss in voltage. By considering the pattern of voltage loss through the component cells in one of the modules, it is possible to narrow down the source of the fault. For example, because hydrogen flows across the cells, a problem with the hydrogen supply will tend to affect the closest cells differently from those farther away. This manifests itself as a pattern of decreasing voltage across the cells. On the other hand, as the air flows through the cells, any problem in the air supply will affect each cell equally, thus resulting in a uniform voltage drop among all of them.

Unfortunately, the pattern of voltage loss does not identify the fault uniquely. For example, if a uniform voltage drop is observed, we can infer a problem with the air supply or the current supply. Furthermore, there are at least two possible problems with the air supply: too high a humidity or too low a humidity. Consequently, we need to conduct further tests to help close in on the fault.

[1] Researchers at Johnson Space Center have applied the expert system KEE to this problem [21]. It is interesting to note how they are forced to use procedures for the diagnosis, and how the representation of these procedures manifests all the failings of conventional programming techniques as discussed in Section 2.1.

Note that now we have a *sequence* of tests to perform if we suspect a problem in a fuel cell module:

1. Test for voltage drop

2. If the voltage has dropped, examine the pattern of voltage loss

3. If the pattern is uniform, test humidity and temperature; otherwise, if the pattern is ..., test for ...; etc.

Not only is this a sequence of tests, but it involves conditionals as well. In other words, it is a fully-fledged *procedure* for isolating the fault. Furthermore, each of these tests might itself be quite a complex procedure. For example, examination of the pattern of voltage loss might require a sequence of probes.

Sometimes, parameters may not be examinable directly, either because no sensor is provided or because a sensor is faulty. In the above example, it may be that the humidity sensor is faulty, in which case it could not be used to help isolate a fault involving a fuel cell module (indeed, it could be the *cause* of the problem). In such a situation, one way to distinguish between too high and too low a humidity is to adjust the humidity in one direction and see if that improves things or not.

But even this method has its complications. If we *increase* the humidity, nothing much happens except that module performance gets either steadily better or steadily worse. But if we *decrease* the humidity we have a chance of bringing the module into a critical region that would require it to be shut down immediately. Thus it is important that we try to increase the humidity and observe the outcome, ratber than decrease it.

Let us now consider how this knowledge about fault isolation of a fuel cell module might be represented in a standard rule-based expert system, such as EMYCIN [32], OPS [9], or such commercially available systems as S1 (Teknowledge), ART (Inference Corp) and KEE (Intellicorp).

For this problem, a possible set of rules for a forward-chaining system might include the following:

1. `if do(isolate-problem module) then do(volt-test module)`

2. `if voltage-drop(module) then do(patt-test module)`

```
3. if done(patt-test module) and pattern(module, uniform)
   then do(hum-test module)

4. if done(patt-test module) and pattern(module, ...)
   then do(...)

5. if done(hum-test module) and high(humidity)
   then done(isolate-problem module) and do(red-hum)

6. if done(hum-test module) and low(humidity)
   then done(isolate-problem module) and do(inc-hum)
```

and so on.

Of course, this is not the only representation possible with a rule-based scheme. For example, as an alternative to the condition about having done the pattern test in Rule (3) (i.e., `done(patt-test module)`), we could have added a condition on the voltage drop (i.e., `voltage-drop(module)`). This would be more or less equivalent to the formulation given above, but in general one has to be careful. For example, the pattern test might have a temporary effect of restoring the voltage to normal.

We could not, however, remove the contextual information. That is, we could not use the rule

```
if pattern(module, uniform) then do(hum-test module)
```

as an alternative to Rule (3). The fact that we have just done the pattern test must be included (one way or another) in the antecedent, as otherwise the rule could be invoked (be "fired") when the module is operating normally (in which case the voltage pattern is likewise uniform).

The system must also include rules describing how to conduct the various tests, such as

```
if do(hum-test module) then do(x) and do(y) and do(z)
```

The intent here is that the humidity test involves performing, in order, the actions x, y, and z. Furthermore, these actions may themselves be defined by quite complex procedures.

We again have the problem of specifying the context in which actions and tests are performed. For example, the various actions above might have diverse outcomes that could have different implications in different contexts. If this were the case, we could not include as the consequent of each action simply the results of performing that action in isolation: doing so would fail to capture the fact that the consequents are context-dependent. So we would probably need rules of the kind

```
if done(x) and done(y) and done(z) then done(hum-test module)

if done(hum-test module) and voltage-decrease(module)
then high(humidity)
```

As can be seen, things are beginning to get very messy and complex. And the situation becomes commensurately worse for the far more complex components typically found in space systems.

Chapter 3

Procedural Knowledge

3.1 Representing Procedural Knowledge

It is clear from the preceding discussion that operational procedures involve very complex control structures and are based on a wealth of knowledge about operational conditions, usage and experience with similar equipment, best available engineering judgment, technical edicts, operational flight rules, and safety considerations. It is clearly not sensible to try and "deproceduralize" this knowledge so that it can be represented in a form suitable for conventional expert systems.[1] We therefore need to develop a knowledge representation that allows arbitrary facts to be stated regarding procedures and their effects, and that, at the same time, enables the use of this knowledge to achieve desired operational goals. But first we must define some basic concepts that we will be dealing with.

We view a system as trying to to attain certain goals by performing certain actions in some environment. At any given instant, the world is in a particular *world state.* The world includes both the environment external to the system and the system's

[1]This is not to say that formalizing the knowledge used in the *construction* of the operational procedures would not be worthwhile. However, any simple reformulation of the procedures into rule form would gain nothing (and as we have shown, would actually be disadvantageous). Furthermore, should it eventually be possible to formalize the designer's knowledge (and this is currently well beyond the state of the art), it would be better to use this knowledge to construct operational procedures (in effect, to "compile" some of the knowledge) rather than always being forced to "reason from first principles".

own internal state. World states are described by statements in predicate calculus, including conjunction, disjunction, and negation. For example, the state description

(block a) ∧ (block b) ∧ (on a b)

describes all those worlds in which one block (a) is on top of another (b).

A *behavior* is a sequence of world states that is generated by the system; an *action* (or *action type*) is a set of such behaviors. We use so-called *temporal statements* to describe actions (or behaviors). A temporal statement consists of one of the temporal operators !, ?, or #, followed by a [nontemporal] state description. For a given state description p (such as the one given above), the meaning of these temporal operators is as follows:

- The expression $!p$ is true of a sequence of states if p holds in the last state of the sequence. Thus, it represents the performance of an action that tries to *achieve* the condition p.

- The expression $?p$ is true of a sequence of states if p holds in the first and last states of the sequence, thus representing a [nondestructive] *test* for the condition p.

- The expression $\#p$ is true of a sequence of states if the truth value of p is maintained throughout the sequence, i.e., it *preserves* p.

We describe a *procedure* by specifying the possible sequences of goals that the system will try to achieve in executing that procedure. A *goal* specifies a desired behavior of the system. This view of goals as behaviors is unlike that found in most planning and reasoning systems, where goals are usually represented as nontemporal statements denoting states of the world to be achieved. The scheme adopted here allows a much wider class of goals to be represented, including goals of maintenance (e.g., "achieve p while maintaining q true") and goals with resource constraints (e.g., "test p within the next 10 minutes").

The procedure itself is specified by using a recursive transition network (RTN), which can be viewed as a kind of flowchart. We will call this the *body* of the procedure. The arcs of the RTN are labeled with goal descriptions, and the various possible paths

through the network from the start node to a final node represent the possible *executions* of the procedure. Because the procedure is represented as a network, arbitrarily complex control constructs can readily be expressed.

Finally, we associate with each procedure an *effect*, which is a description of the behaviors that will be realized if the procedure is successfully executed.

A sample procedure description, representing a method for returning a fuel cell module to proper operation as described in the previous chapter, is shown in Figure 3.1. (The invocation condition is discussed in the next section.) The start node is labeled START and final nodes are labeled END. As mentioned above, one of the crucial features of the representation is that the arcs are labeled with *goals*, that is, conditions to be achieved or tested for, rather than arbitrary procedure names.

This means of representation allows us to state *facts* about the procedure without regard to how these facts may be used. Thus, for example, one of the facts represented by the procedure shown in Figure 3.1 is

> *If* a fuel cell has a voltage drop and it can subsequently be determined that the pattern is uniform, and *if* it can thereupon be established that the humidity is high, and *if* finally it is possible to achieve a lower humidity, *then it follows that* the fuel cell will be rendered operable.

This is a statement about the problem domain, and its truth can be determined without regard to any other procedures or any other statements about the domain. This is what we call the *declarative semantics* of the procedural representation.

Such a semantics is essential for providing a system with explanatory capability, reasoning ability, evolutionary potential, and verifiability. The explanation of a procedure can be more meaningful, as the reasons for performing the various actions and tests are specified. For example, if the system fails to achieve a given goal, it can explain how it was trying to achieve it and what task it failed to complete, thus allowing a user to suggest alternative approaches. Moreover, the system can reason about composite goals. For example, it could determine that a given conjunctive goal could be realized by achieving all the component goals, one after the other. System evolvability is ensured because each procedure expresses a fact about the world that is *independent* of any other facts about the world. This independence also means that the validity of a procedure can be examined irrespective of the definition of any other procedure.

INVOCATION: (FACT (COMPONENT $MODULE ECLSS))
AND (GOAL (! (OPERABLE $MODULE)))

EFFECTS: (! (OPERABLE $MODULE))

BODY: .

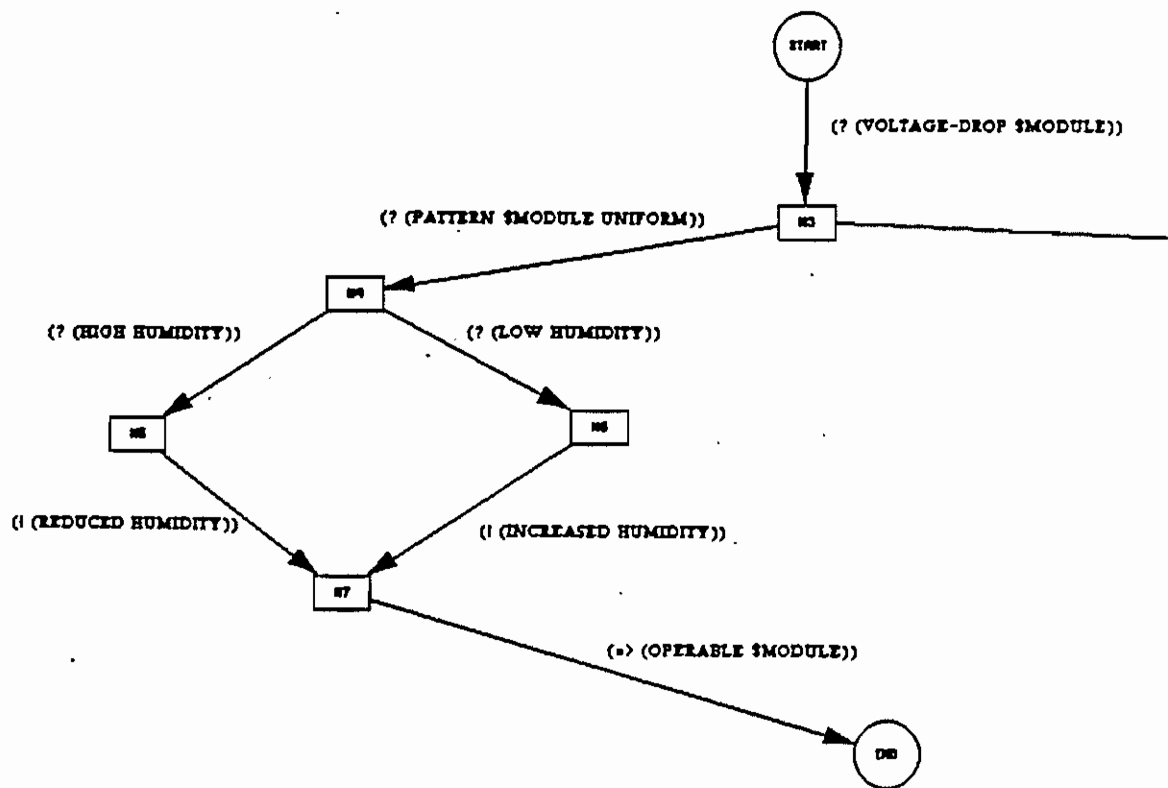


Figure 3.1: Portion of KA for Fuel Cell Malfunction

Thus, once all procedures have been independently validated, we can be certain that *no* situation could possibly arise that would cause these procedures to be executed incorrectly.

A more formal treatment of the semantics of the representation and the underlying process model is described elsewhere [15,12].

3.2 Using Procedural Knowledge

Procedural descriptions provide a way of describing the effects of actions in some dynamic problem domain. That is, they state that the realization of certain sequences of goals (specified in the body of the procedure description) will result in a particular class of behaviors (specified in the effects of the procedure). But how can a system *use* this knowledge to achieve its goals?

One way to accomplish one's goals is to select a procedure whose effects imply that the goal will be achieved, i.e., whose set of possible behaviors is included in the set of behaviors denoted by the goal. A simple interpreter based on this idea could be usefully implemented. However, it is not always wise to invoke a procedure simply on the basis of its effects. For example, one way to reduce pressure in a fuel tank is to blow it up, but this is not a sensible procedure for achieving this goal. Similarly, some procedures (such as emergency procedures) need to be invoked solely because some critical event has occurred, and thus do not have to be responsive to any particular goal having been set. In these cases, admittedly, there is indeed some underlying goal that is being achieved (such as maintaining the safety of the spacecraft), but it is implicit in nature. Rather than be compelled to make such goals explicit, it is preferable to have a mechanism that allows procedures to be invoked on the basis of either explicit or implicit goals.

To accomplish this, we associate with each procedure a form of metalevel knowledge that specifies under which circumstances invocation can occur. This is called the *invocation condition* of the procedure. The combination of a procedure and its invocation condition is called a *knowledge area* (KA). The invocation condition is an arbitrary logical expression, which may include constraints on both currently known facts and currently active goals. A KA can be executed or invoked only if its invocation part evaluates to "true".

For the KA given in Figure 3.1, the invocation condition indicates that this KA may be useful when the current goal is to isolate a problem involving the operation of a fuel cell module in the ECLSS.

A selected KA can be used to achieve a given goal by achieving, in order, each of the goals appearing in some path through the body of the KA. Thus, when an arc of a KA is to be traversed, the goal labeling that arc is set up as a new goal of the system. This new goal may be attained either by realizing that it has already been achieved, by performing some primitive action directly, or by executing other KAs whose invocation conditions match the goal. If any of these possible methods succeeds in accomplishing the goal, the arc of the original KA can be traversed and execution can progress to the next node in the network. The KA is considered to have been successful when, and if, execution reaches the end node. We call this the the *operational* or *procedural* semantics of the KA.

For example, if the KA in Figure 3.1 were invoked, it would first try to test whether there was a voltage drop in the given fuel cell. This might involve executing some simple test directly, or may involve invocation of some other KA to perform the test. If a voltage drop were not detected, the KA would fail and perhaps some other KA for isolating fuel cell faults could be tried instead. However, if a voltage drop were detected, the KA would then try to test the pattern of voltage loss in the fuel cell. If this were normal, it would test humidity, and finally try to modify the humidity appropriately.

As this example illustrates, we can view the body of a KA either as a statement about the way the world is (its declarative semantics) or as a procedure to achieve something (its operational semantics). This is similar to the manner in which a Prolog statement can be viewed either procedurally or declaratively. The importance of a declarative semantics for knowledge representation schemes was emphasized early on in AI [16], and the combination of an operational and a declarative semantics is one of the major reasons for the success of Prolog [19]. However, Prolog and most knowledge representation languages (e.g., predicate calculus, rule- or frame-based languages [1,28]) are concerned with inference regarding a single state of the world. Procedural expert systems extend these ideas to reasoning about actions, tests, and *sequences* of states — that is, to entire *histories* of the world.

Another important point to note about the KA in Figure 3.1 is that it captures all

the information contained in the rule-based representation, yet is much less cumbersome and much more natural. Furthermore, it provides a considerable gain in efficiency, as the conditions used for representing the control structure in the rule-based scheme (e.g., done(module patt-test)) do not have to be matched with some global data base, but instead are represented explicitly in the flowchart structure.

As mentioned above, we may need other KAs to tell us how to perform particular tests and achieve given goals. For example, if there were a procedure for determining humidity (see Section 2.3), we could represent this by the KA given in Figure 3.2. As for the previous KA, there is no need for the explicit contextual information about having done actions x, y, and z (which is needed for the rule-based representation); that information is implicit in the structure of the KA's body.

During execution of the body of a KA, other KAs become applicable (and thus available for execution) whenever a new goal is established that matches their respective invocation conditions. A KA that responds in this way is called *goal-directed*. Alternatively, a KA may respond to the discovery of some new fact about the current state of the world. This happens whenever the invocation part of a KA matches the new fact. Such a KA is said to be *data-driven*.

Such a reactive capability is indispensible for safe and efficient operation of the space shuttle. For example, if the interconnection of the OMS and RCS were done as in Figure 1.1, we would want the system to reconfigure the RCS upon noticing intent to deorbit. Similarly, upon sensing a violation of minimum thermal operating constraints, the system should check as to whether it resulted from an OMS or RCS leak and, if so, delay deorbit (see Section 1.2). In this case, we could use an invocation condition of the form

```
(fact (violated minimum-thermal-constraints))
```

and have the corresponding KA respond as soon as that fact becomes known.

KAs can also be partly goal-directed and partly data-driven, since, in general, the invocation part can be any logical expression involving current facts and goals. Thus, the system can be opportunistic in the more general sense that KAs might be invoked because certain facts were observed during an attempt to establish particular goals. This is particularly important because we often need the system to react in different ways to observed conditions, depending on the current operating mode.

INVOCATION: (GOAL (? (HIGH HUMIDITY)))

EFFECTS: (GOAL (? (HIGH HUMIDITY)))

BODY:

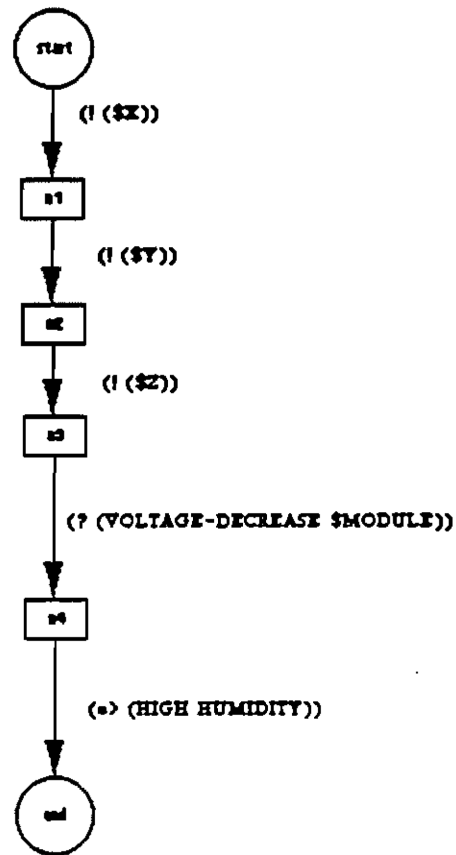


Figure 3.2: KA for Testing Humidity

3.3 Procedural Expert Systems

In the previous section we described how KAs could be used to achieve given goals and react to particular situations. In this section we describe the basic architecture of a system based on these ideas, called a *procedural expert system* (PES).

The overall structure of a procedural expert system is shown in Figure 3.3. The system consists of a *data base* containing currently known *facts* about the world, a set of current *goals* or tasks to be performed, a set of KAs (procedure descriptions together with invocation criteria) describing how certain sequences of actions and tests may be performed to achieve given goals or react to particular situations, and an *interpreter* (or *inference mechanism*) for manipulating these components. At any moment, the system will also have a *procedure stack*, containing all currently active KAs, that can be viewed as the system's current *plan* for achieving its goals or reacting to some observed situation.

Since the data base is intended to describe the state of the world at the current instant of time, it contains only *state* descriptions. However, some of this information will be about static properties of the domain, such as the fact that a particular fuel tank is part of a particular RCS. Other information will be highly dynamic, such as the fact that the current pressure in the fuel tank is 130 psi.

The system interpreter runs the entire system. From a conceptual viewpoint, it operates in a relatively simple way. At any point in time, certain goals are active, and certain facts or beliefs are held in the data base. Given these extant goals and facts, and depending on their invocation parts, a subset of KAs will be deemed relevant (applicable). One of these KAs will then be chosen for execution. In the course of traversing the body of the chosen KA, new goals will be formulated and new facts and beliefs will be derived. At such points, once again, newly relevant KAs are found and possibly invoked.

Thus, when new goals are pushed onto the goal stack, the interpreter checks to see whether any new KAs are relevant, and, if there are, chooses one and executes it. Likewise, whenever a new fact is entered into the data base, the interpreter will perform appropriate truth maintenance procedures and possibly activate newly applicable KAs. The system is therefore *reactive*, rather than merely goal-driven: KAs may respond not only to goals, but also to facts. For example, when a new fact enters the system

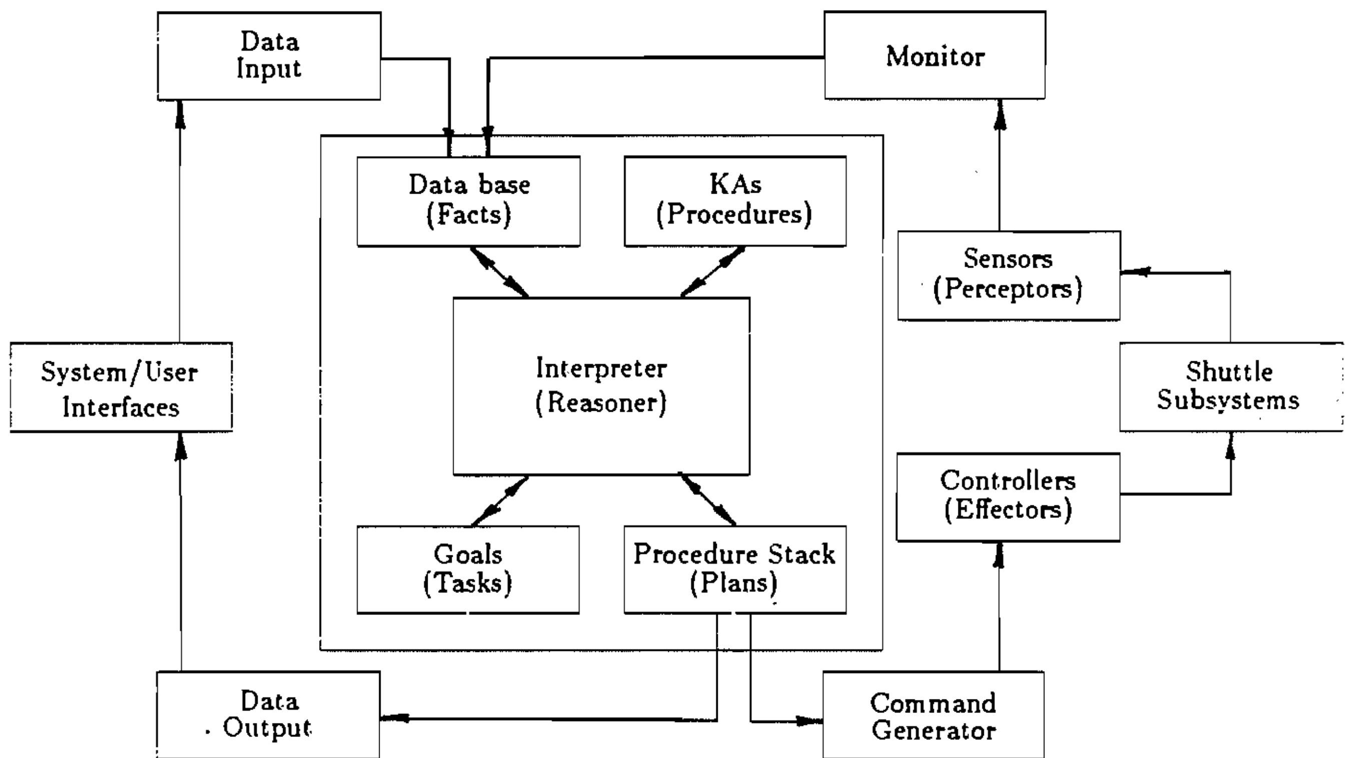


Figure 3.3: System Structure

data base, execution of the current KA might be suspended, with a newly relevant KA taking over.

One of the key aspects of the system is the mechanism which determines when KAs are applicable. This works by matching the invocation conditions of KAs with the facts in the data base and the goals on the goal stack. As the invocation conditions may be parameterized, it is necessary that there be some scheme for matching the variables and constants appearing as parameters of an invocation condition with those appearing in the expressions representing the goals and facts of the system. To do this, the interpreter employs a form of pattern matching called *unification* to determine whether or not the invocation part of a given KA matches the extant system goals and facts. This is similar to the approach used in the programming language Prolog.

One of the advantages of unification that is unlike parameter binding conventions in standard programming languages, is that it is unnecessary to decide prior to execution which of the variables are to count as input variables and which as output variables. This is important from the standpoint of flexibility and ease of verification. Unification also confers other important benefits. In particular, it avoids binding variables until absolutely necessary, which can often be advantageous in allowing difficult decisions to be avoided or deferred.

An abstract interpreter for the system is given in Figure 3.4. The interpreter works by exploring paths from a given node n in a KA, P, in a depth-first manner. To transit an arc, it unifies the corresponding arc assertion with the invocation conditions of the set of all KAs, and executes those that unify, one at a time, until one terminates satisfactorily. If none of the matching KAs terminate successfully, and all leaving arcs fail, the execution of P fails.

The function KAs-that-unify takes a set of goals or data base facts and returns the set of KAs that unify with some element in the set. The function corresponding-arc returns the arc corresponding to the selected KA instance (i.e., the arc with which it unified). The function return returns from the enclosing function (eval in this case), not just the enclosing do. The initial system goal is explicitly placed on the goal stack by the user.

The function select selects an element from a set, destructively modifying the set as it does so. In the real system, this is done by forming a metalevel goal to select which KA to next execute. The appropriate metalevel KAs respond and make the selection.

```
function eval (P n)
  if (is-end-node n) then
    return true
  else
    arc-set := (outgoing-arcs n)
    goal-pr-set := (KAs-that-unify arc-set)
    fact-pr-set := (KAs-that-unify data-base)
    pr-set := (append goal-pr-set fact-pr-set)
    do until (empty pr-set)
      proc := (select pr-set)
      if (fact-invoked proc) then
        (eval proc (start-node proc))
      else arc := (corresponding-arc proc)
        if (eval proc (start-node proc)) then
          return (eval P (terminating-node arc))
    end-do
    return false
end-function
```

Figure 3.4: An Abstract Interpreter for PES

These metalevel KAs are manipulated and invoked by the system in the same way as any other KA. However, they respond to facts and goals pertaining to the system itself, rather than just those of the application domain. In this way it is possible to include both domain-independent and domain-dependent selection criteria, and to represent this knowledge in the same formalism as other knowledge of the domain.

At the bottom level, the system must actually perform primitive actions and tests to realize its goals. These are performed by sending appropriate messages to other systems or by sending commands to specific effectors and sensors. Currently, the performance of actions and tests is all mediated through the global data base, which represents the system's current beliefs about the world. In this view, a test directly updates the data base with new facts as they become known. Similarly, successful

completion of an action results in updating of the data base by the goal that the action achieved.

In the design of the system, there is no assumption that the performance of an action by some effector will actually accomplish the desired goal. The device called to perform an action has either to assert that it has achieved the goal or has to invoke a test to check that it has been achieved.

Neither is there any inherent assumption that sensors are accurate or error-free. To model this possibility, a sensor, for example, may put into the data base the fact that *it* observed some predicate p to be true. Further reasoning by the system (as well as, perhaps, integration with the views of other sensors) might be necessary before the system itself believed p (i.e., before p itself was added to the data base).

For example, a sensor, say s-101, might like to enter into the data base the fact that it observed "`(holding A)`" to be true. One way to do this would be to add something like "`(believes s-101 (holding A))`" to the data base. Various KAs may then respond to this fact and, on the basis of other information contained in the data base, determine whether or not this belief is likely to be true.

Chapter 4

RCS Application

The development of an adequate knowledge representation requires both theoretical research and experimentation with a real system. In this chapter we describe an implemented experimental procedural expert system and discuss an application on which the system was tested.

4.1 The System

We have implemented an experimental system, PES (the SRI Procedural Expert System), based on the ideas presented in the previous chapter. The implemented system is written in LISP and runs on a Symbolics 3600 machine. In this section we present an overview of the system structure, describe how domain descriptions are encoded in the system, and also try to bring across the flavor of its user interface. Quite an elaborate window system has been constructed for interacting with PES. Among the facilities provided is a graphical package that allows direct entry and manipulation of KA networks as well as visualization of system execution in terms of these graphical networks.

The basic structure of PES is shown in Figure 3.3 of the previous chapter. From the user's point of view, the important components are: (1) the system data base representing the current "beliefs" of the system; (2) the set of KAs representing procedural knowledge about the problem domain; and (3) the set of current goals that the system is attempting to achieve. Each of these must be initially set up by the user. A

complete domain description might thus consist of, say, a data base that describes the structure of a complex piece of equipment as well as current failure indications, a set of KAs that describe procedures used for trouble-shooting the equipment, and domain goals that seek the determination of a faulty module.

A description of each of these components and their usage is given below.

4.1.1 The System Data Base

The data base of PES may be thought of as the current "beliefs" of the system. Some of these beliefs may be provided initially by the system user. Typically, these will include facts about static properties of the application domain – for example, the structure of some subsystem, or the physical laws that some mechanical components must obey. Other facts are derived by PES itself as it executes its KAs. These will typically be current observations about the world or conclusions derived by the system from these observations. It is clear then, that the PES data base is *nonmonotonic* – at some times, for example, the system may believe that a particular valve is open – at other times, closed. Thus, part of the PES data base implementation involves truth maintenance – making sure that the system's data base is consistent within itself at any particular time.

The system data base consists of a set of *state descriptions* describing what is true (or what is believed to be true) at the current instant of time. We use first-order predicate calculus for the state description language. The standard logical connectives – ¬ (negation), ∧ (conjunction), and ∨ (disjunction) – are allowed and have their usual meaning. We use prefix notation (as in LISP) and both ∧ and ∨ take an arbitrary number of arguments. Quantification is usually implicit and, depending on context, may be either existential or universal. Within the data base, free variables are assumed to be universally quantified, and are represented by symbols prefixed with a $ sign.

For example, in the system data base the statement `(on a table)` can be taken to represent the fact that the object denoted by `a` is on top of the object denoted by `table`. The statement `(red (color $x))` means that every object is colored red, and the statement `(∨ ( ¬ (on $x table)) (red (color $x)))` means that every object on the table is red. Note that, in this case, the free variables are assumed to be universally quantified.

State descriptions are not limited to describing states of the *external* environment, but can also be used for describing *internal* system states. Expressions that refer to internal system states are called *metalevel* expressions. Because these expressions refer to the system itself, all the basic metalevel predicates and functions are predefined by the system. For example, **goal** is a predefined metalevel predicate that is true if its first argument is a current goal of the system.

4.1.2 Behaviors and Goals

Goals appear both on the system goal stack and as labels on the arcs of KAs. Unlike most expert systems, these goals represent desired *behaviors* of the system, rather than static world states.

To specify goals, we need some language for describing behaviors. A *behavior description* (or *action description*) is a condition that is true of some interval of time, i.e., that is true of some sequence of world states. Such sequences may be described by a *temporal predicate* applied to an n-tuple of terms. Each temporal predicate denotes an *action type* or a *set* of state sequences. That is, an expression like "(walk a b)" can be considered to denote the set of walking actions from point a to b.

A behavior description can also be formed by applying a temporal operator to a state description. The temporal operators currently used are !, ?, and #. The statement (!p), where p is some state description (possibly involving logical connectives), is true of a sequence of states if p is true of the last state in the sequence; that is, it denotes a behavior that *achieves* p. For example, we might use a behavior description of the form (!(walked a b)) rather than (walk a b). Similarly, (?p) is true if p is true of the first and last states in the sequence, and can be considered to denote a behavior that *tests* for p. Finally, (#p) is true if p is preserved (maintained invariant) throughout the sequence.

Behavior descriptions can be combined using the logical operators $\wedge$ and $\vee$, representing intersection and union operations, respectively. The interpretation of variables is fixed over the interval (sequence of states) to which the behavior description is applied. Quantification is usually implicit, its type depending on the particular context in which the expression is used (see below).

As with state descriptions, behavior specifications are not restricted to describing the external environment, but can also be used to describe the internal behavior of the

system. Such behavior specifications are called metalevel specifications. One important metalevel form is (=> *p*), which specifies a behavior that places the state description *p* in the system data base.

4.1.3 Knowledge Areas

Knowledge about procedures is represented in PES by KAs. Each KA consists of a *body* represented within the system as a graphical network that encodes the steps of the intended procedure. A KA must also include an *invocation condition* that specifies under what situations the KA may be used, as well as what it is useful for (i.e., a declaration of what types of goals the procedure can be used to achieve, and under what situations it is truly applicable). The user of PES inputs all of this procedural information via a graphical network editor that is part of PES.[1]

Each PES application is associated with a set of KAs that describe how to achieve particular goals in the given application domain as well as how to react to specific facts in the data base. Some of these KAs may be *meta-level* KAs – that is, they contain information about the manipulation of PES itself (for example, how to choose between multiple relevant KAs, or how to achieve a conjunction, disjunction, or the negation of goals). In addition to those KAs that are supplied by the user, each PES application contains a set of KAs that are a default part of every system. These typically are domain-independent metalevel KAs.

The bodies of KAs are represented using a recursive transition network whose arcs are labeled with *goals*. Variables used in the body of a KA are classified as either *global* (represented by symbols prefixed by a $ sign) or *local* (represented by symbols prefixed by a % sign). Informally, the interpretation of a local variable is fixed in the interval during which a given arc is transited, but can otherwise vary. A global variable, on the other hand, has a fixed interpretation during the execution of the entire KA. (Local variables are often needed in loops where it is necessary to identify different elements from one iteration to the next.) The current system also makes use of program variables (prefixed by @) that behave like local variables (i.e., they may change value on each new arc) but whose value is retained or "remembered" from one arc to the next. A program variable @x may only be rebound within a behavior of the form

[1]The graphical network editor is called GRASPER II and was developed at SRI International's Artificial Intelligence Center.

`(! (= @x` *expression*`))`. Such a variable thus behaves much like a program variable in standard programming languages. We do have a proper semantics for program variables, however, that is consistent with the semantics for the more standard logic variables of form `$x` and `%x`.

In addition to the KA body, we also need to specify the invocation condition associated with each KA, which states under what situations the KA should become *applicable* (i.e., be made available for execution). Currently, this is done by specifying to which goals it should respond, to which facts it should react, or some logical combination of these. We do this by using *metalevel* predicates, which refer to the system's internal state rather than the external environment. There are two metalevel predicates that are important in this case: (1) `goal`, which takes a behavior description g as its argument, and is true if g unifies with a goal of the system; and (2) `fact`, which takes a state description f as its argument, and is true if f unifies with a statement in the data base of the system. These metalevel primitives may be combined using either conjunction (represented by `AND` at the metalevel) or disjunction (represented by `OR`). A sample metalevel statement for specifying applicability conditions is the following:

```
(AND (goal (!(¬ (p $x $y))))(fact (g $x unit-1)) )))
```

It states that the particular KA being specified is applicable precisely when `(!(¬ (p $x $y)))` unifies with a current system goal *and* `(g $x unit-1)` unifies with some fact that is currently known by the system. Note that any global variable that appears in a KA is implicitly universally quantified, its scope extending over both the invocation part and the body of the KA.

Within the current version of PES, we do not explicitly specify the *effects* of KAs. Instead, we assume that any goals that appear in the invocation part of a KA are achieved by a successful execution of the KA (provided the facts appearing in the invocation part are also satisfied) – that is, the goals appearing in the invocation part are also considered to be the effects of the KA. The reason for this is simply convenience in the specification of KAs. If the user desires to represent an effect of a KA but *not* have it appear as an invocation condition, that effect can instead be asserted in the data base by labeling a final arc of the KA with a metalevel goal of the form `(=>` *effect*`)`. In later versions of this system, we may find it useful to separate out the invocation conditions and effects as was done in the KAs that were described in previous chapters.

Sometimes it is convenient to represent procedures directly as Lisp code rather than in the graphical form expected of KA bodies. To retain the reactive and flexible nature of KAs, such procedures are specified and invoked as normal KAs, except that the body of the KA is replaced with Lisp code. Such KAs are called *Lisp KAs*. In the current interface, the appropriate Lisp code is actually placed on a property list of the corresponding Lisp KA under the key *ACTION*. One example of a Lisp KA is the default KA called <. It can be invoked by a goal of form `(!(< $x $y))` or `(?(< $x $y))`, but its invocation results in execution of the normal Lisp function <, applied to the two parameters **$x** and **$y**, rather than in the execution of a KA body. Another important metalevel Lisp KA is `S-=`. It reacts to goals of the form `(!(= $x $y))` and results in the execution of a Lisp function that manipulates the internal bindings of global variables in an appropriate fashion. Users may also define Lisp KAs as part of their own applications.

4.1.4 User Interface and Menu System

The PES user interface is the medium through which a user creates a new application system, loads or modifies an existing system, or runs a system. Part of this interface is a sophisticated window system that aids the user in all of these tasks.

The main PES window is divided into four parts (see Figure 4.1). The top pane is used for a textual trace of KA execution. As KAs are invoked and their edges traversed, the textual trace reads out KA names, edge expressions, as well as variable bindings.

The second pane is a user run-time interaction window - the input/output pane. Here, messages that are "sent to" the user by a KA (for example, questions that ask the user for particular kinds of information) are printed out, and user responses are typed in and sent back to the KA. For instance, in the RCS application, we use the input/output pane to get simulation information from the user. Thus, a KA may send a message of the form "Are we using orbiter OV102?" and the user may respond "yes" or "no."

The third pane is the graphical KA-tracing pane. At any point in time, a user may specify that they want a subset of the KAs traced (of course, all or none of the KAs may be traced as well). When those KAs that have been selected for tracing

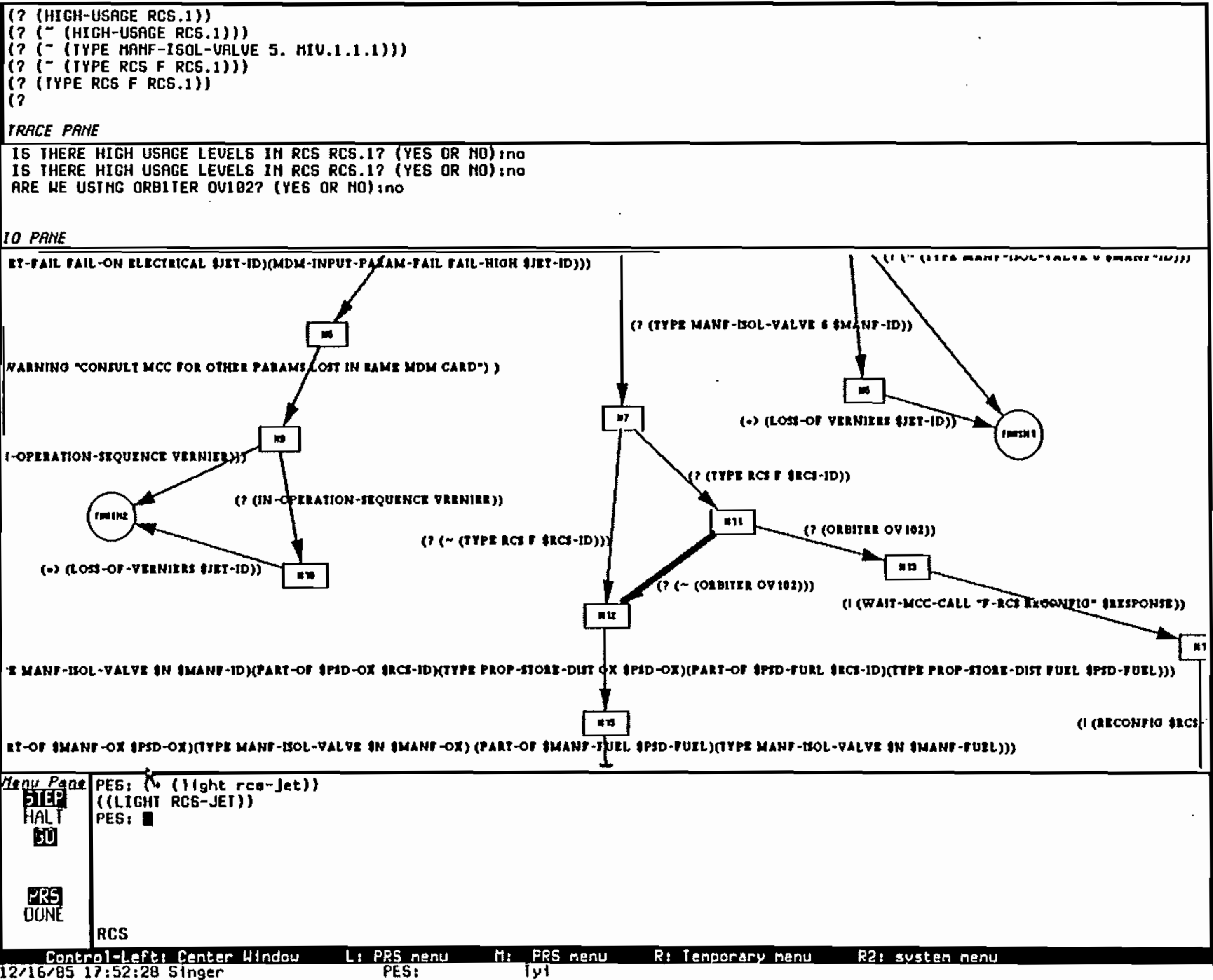


Figure 4.1: PES User Interface

are executed, they are displayed in the graphical tracing pane. As their edges are traversed, these edges are graphically highlighted. If the user desires, edge tracing may be given a certain "tail length" - e.g., given a "tail length" of three, the three most recently executed edges are highlighted, the most recently executed edge being highlighted the darkest. This graphical tracing facility enables the user to see what is going on visually and in the full context of an entire KA body, rather than trying to follow a more complex textual trace.

Finally, the fourth window pane is the menu and PES system interaction pane. The user may execute any Lisp function in this window. In addition, an entire hierarchy of pop-up menus are available for loading, editing, starting up execution, and interacting with the PES application. Right now, the primary top-level menus are the following:

LOAD: Guides the user through loading an application that has already been set up.

EDIT: Serves as an interface for creating and modifying (editing) the system data base and KAs.

RUN: Guides the user in running a loaded application system. A lower-level menu is provided for asserting facts in the system data base, or putting new goals on the goal stack, and is thus a vehicle for getting system KAs to respond and execute.

TRACE: Enables one to turn graphical and textual tracing of KA execution on and off, as well as adjust other tracing parameters.

HELP: Prints documentation of these commands.

Usage of PES will normally follow this pattern:

- *Creation of Application System:*

  Use `EDIT`.

- *Testing of Application System:*

  Repeated use of the following cycle:

  1. `LOAD` to load the system.
  2. `RUN` to run the system.
  3. `EDIT` to modify the system.

There are a large number of potential enhancements for the PES window system and user interface, as well as the system internals. As it stands, it is already a sophisticated framework for building KAs and visualizing their execution. Areas for future progress include:

- Augmenting the tracing facility with a full run-time debugging facility. This would include run-time interaction with the system data base, goal stack, and KA descriptions. In addition, color graphics could be used to encode more kinds of tracing information. For example, a different color edge could indicate success or failure of the goal labeling that edge.

- Setting up an environment for creating, running, and visualizing the execution of multiple, interacting PESs. This is necessary for dealing with environments in which parallel forms of reasoning and interaction must take place, and is particularly suited for the envisioned space station environment.

- Building a *structure editor* to enable the graphical representation of an application system (for instance, a schematic of a subsystem being tested), the derivation of structural information from that representation, and finally, the integration of that information into the PES data base. This would represent a significant advance over the manual encoding of the structure of an application system into predicate calculus form. A graphical representation of system schematics could also be used as a vehicle for run-time explanation and interaction with a PES user.

- Enhancing the metalevel procedures within the system and providing a richer set of basic metalevel predicates.

4.2 Space Shuttle Example

A potentially useful system for experimentation is the reaction control system (RCS) of the space shuttle mentioned in the previous chapter. The system structure is depicted in the schematic of Figure 4.2. One of the aims of our research is to try and automate the malfunction procedures for this subsystem. Sample malfunction procedures are presented in Figures 4.3 and 4.4.

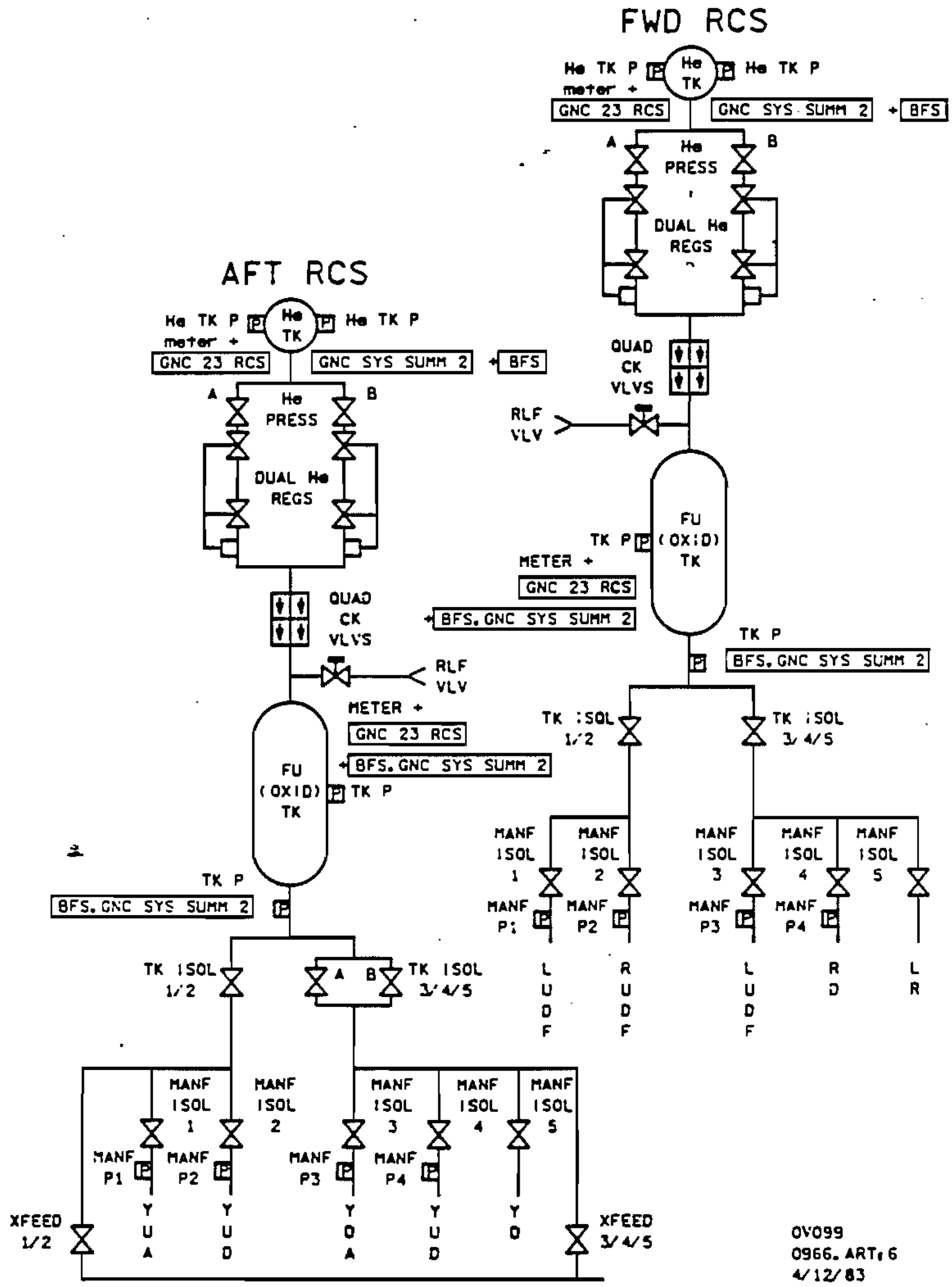
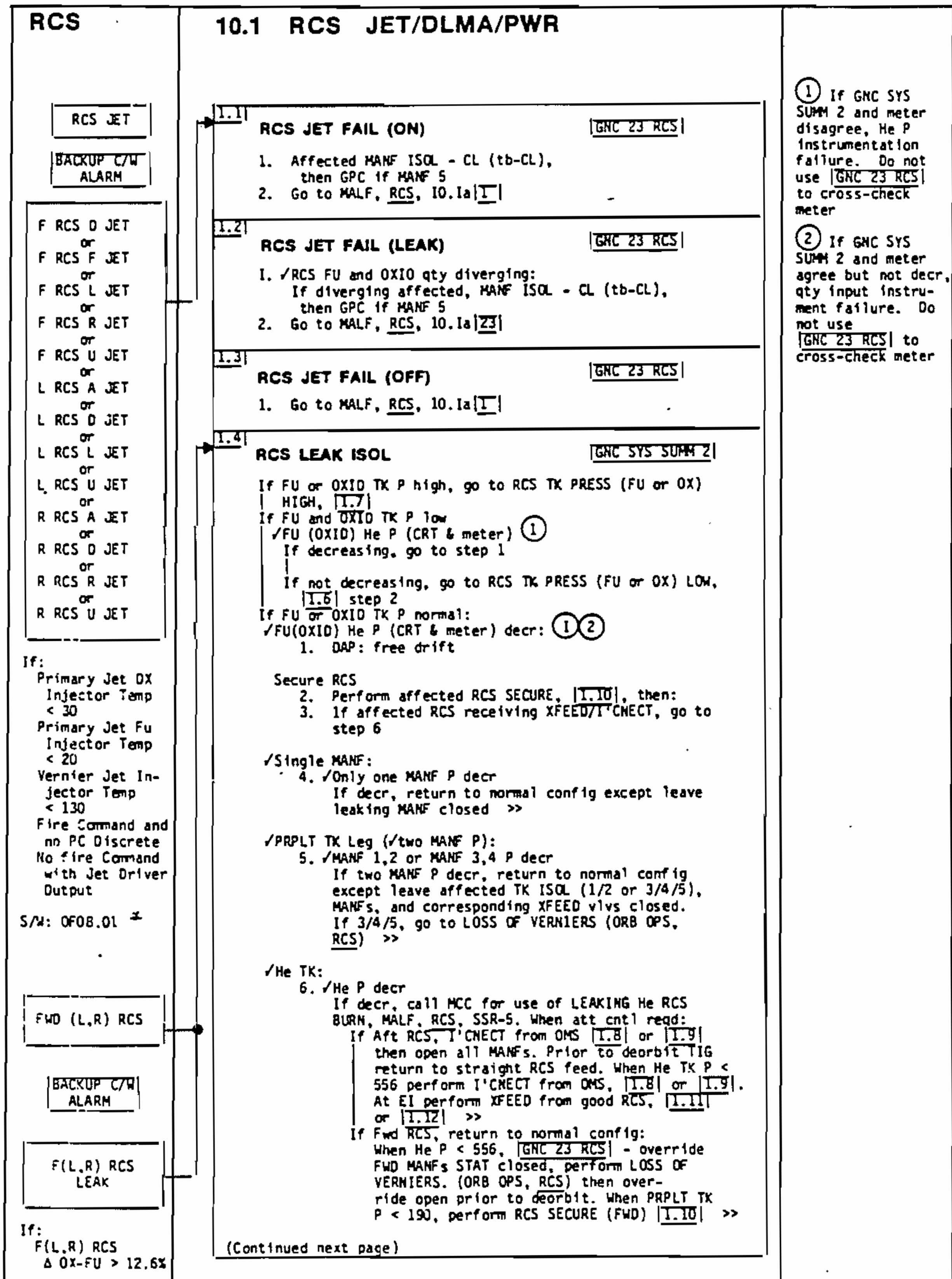


Figure 4.2: System Schematic for the RCS



RCS

10.1 RCS JET/DLMA/PWR

RCS JET

BACKUP C/W ALARM

F RCS D JET
or
F RCS F JET
or
F RCS L JET
or
F RCS R JET
or
F RCS U JET
or
L RCS A JET
or
L RCS D JET
or
L RCS L JET
or
L RCS U JET
or
R RCS A JET
or
R RCS D JET
or
R RCS R JET
or
R RCS U JET

If:
Primary Jet OX Injector Temp < 30
Primary Jet Fu Injector Temp < 20
Vernier Jet Injector Temp < 130
Fire Command and no PC Discrete
No fire Command with Jet Driver Output

S/W: OF08.01

FWD (L,R) RCS

BACKUP C/W ALARM

F(L,R) RCS LEAK

If:
F(L,R) RCS
Δ OX-FU > 12.6%

1.1

RCS JET FAIL (ON) GNC 23 RCS

1. Affected MANF ISOL - CL (tb-CL), then GPC if MANF 5
2. Go to MALF, RCS, 10.1a 1

1.2

RCS JET FAIL (LEAK) GNC 23 RCS

1. √RCS FU and OXID qty diverging:
   If diverging affected, MANF ISOL - CL (tb-CL), then GPC if MANF 5
2. Go to MALF, RCS, 10.1a 23

1.3

RCS JET FAIL (OFF) GNC 23 RCS

1. Go to MALF, RCS, 10.1a 1

1.4

RCS LEAK ISOL GNC SYS SUMM 2

If FU or OXID TK P high, go to RCS TK PRESS (FU or OX) HIGH, 1.7
If FU and OXID TK P low
√FU (OXID) He P (CRT & meter) ①
If decreasing, go to step 1
If not decreasing, go to RCS TK PRESS (FU or OX) LOW, 1.6 step 2
If FU or OXID TK P normal:
√FU(OXID) He P (CRT & meter) decr: ①②
1. DAP: free drift

Secure RCS
2. Perform affected RCS SECURE, 1.10, then:
3. If affected RCS receiving XFEED/I'CNECT, go to step 6

√Single MANF:
4. √Only one MANF P decr
   If decr, return to normal config except leave leaking MANF closed >>

√PRPLT TK Leg (√two MANF P):
5. √MANF 1,2 or MANF 3,4 P decr
   If two MANF P decr, return to normal config except leave affected TK ISOL (1/2 or 3/4/5), MANFs, and corresponding XFEED vlvs closed.
   If 3/4/5, go to LOSS OF VERNIERS (ORB OPS, RCS) >>

√He TK:
6. √He P decr
   If decr, call MCC for use of LEAKING He RCS BURN, MALF, RCS, SSR-5. When att cntl reqd:
   If Aft RCS, I'CNECT from OMS 1.8 or 1.9 then open all MANFs. Prior to deorbit TIG return to straight RCS feed. When He TK P < 556 perform I'CNECT from OMS, 1.8 or 1.9. At EI perform XFEED from good RCS, 1.11 or 1.12 >>
   If Fwd RCS, return to normal config: When He P < 556, GNC 23 RCS - override FWD MANFs STAT closed, perform LOSS OF VERNIERS. (ORB OPS, RCS) then override open prior to deorbit. When PRPLT TK P < 190, perform RCS SECURE (FWD) 1.10 >>

(Continued next page)

① If GNC SYS SUMM 2 and meter disagree, He P instrumentation failure. Do not use GNC 23 RCS to cross-check meter

② If GNC SYS SUMM 2 and meter agree but not decr, qty input instrument failure. Do not use GNC 23 RCS to cross-check meter

Figure 4.3: Some RCS Malfunction Procedures

One of the basic difficulties faced with building any knowledge base is that of *axiomatizing* the problem domain — that is, determining the entities of the problem domain, their properties, and their mutual relationships. This is a task that can be accomplished only by extensive discussions with specialists in the given domain.

One then has to acquire from these experts the rules and techniques used for reasoning about problems in the domain of interest. In the application proposed above, much of this information is provided by the malfunction procedures. This saves an enormous amount of effort in building a practical system, as much of the work of knowledge acquisition has already been done.

Unfortunately, our task is still not as straightforward as one might have hoped. The reason is that the procedures do not specify the purpose of the individual tests and actions, and thus do not lend themselves to direct translation into the form desired for procedural expert systems. Had the designers of these procedures followed recommended programming practice and annotated the procedures with descriptions of the overall *intent* of each of its steps (in other words, the conditions that are being made true by each particular step), the situation would be entirely different and a more-or-less direct translation would have been possible. As it is, this information will have to be sought by interviewing mission controllers and engineers.

The manner in which we have represented the actions reflects what we said in the preceding chapter — i.e., that actions and tests must be represented by whatever condition they achieve or test for, rather than by some arbitrary name. For example, there are some malfunction procedures in which one must lower the pressure of a tank that has a high pressure reading, and likewise, raise the pressure of a tank with low pressure. In such cases, the goal is actually to "normalize" the pressure of the tank, and thus, a KA reflecting this procedure would be identified as achieving this goal. This results in a more modular and useful system. Given a set of KAs associated with the actual goal that they achieve, the KAs may then be reused in other circumstances in which they might be useful, or easily replaced by other KAs that achieve their particular goal in a better way.

To get a more in depth view of our RCS application system and to illustrate various advantages of the procedural approach, we now look at some of the RCS KAs and their execution in more detail.

4.2.1 The RCS Data Base

Our first task in encoding this application was to capture the structure of the RCS (depicted in Figure 4.2) as a set of initial data base facts. For this particular application the facts were derived manually; in the future, they could be derived automatically by having the user input the system schematic (for example, the schematic given in Figure 4.2) to our proposed structure editor. Once inserted into the system data base, these facts are used during fault diagnosis to identify particular components of the system and their properties.

For example, a sample set of structural facts is given below. (The entire set of structural facts for the RCS is given in Appendix A.)

```
(type rcs f rcs.1)
(type he-pressurization ox hep.1.1)
(type he-pressurization fuel hep.1.2)
(part-of hep.1.1 rcs.1)
(part-of hep.1.2 rcs.1)
(type he-tank het.1.1.1)
(part-of het.1.1.1 hep.1.1)
(type he-tank het.1.2.1)
(part-of het.1.1.1 hep.1.2)
```

For the purposes of the current system, there are two types of structural facts – `type` facts, which declare specific components or subsystems and associate them with unique identifiers, and `part-of` facts, which state which components and subsystems are part of other subsystems. For example, `(type rcs f rcs.1)` specifies that the system `rcs.1` is a front reactant control system (there are two other reactant control systems: the left aft and right aft). Each RCS contains two helium pressurization subsystems, one for the oxidant part of the system, the other for the fuel subsystem. For RCS `rcs.1` these are labeled as `hep.1.1` and `hep.1.2`, respectively. Finally, each helium pressurization system contains its own helium tank. These tanks are assigned the identifiers `hep.1.1.1` for helium pressurization system `hep.1.1`, and `hep.1.2.1` for the tank in helium pressurization system `hep.1.2`. As the reader may have noticed, the identifiers themselves reflect some of the structure of the RCS. The form of identifier names, however, should only be regarded as a mnemonic device for users; within PES these identifiers are simply regarded as unique tokens, void of semantic meaning.

Once we encode the structure of the RCS in this fashion, our diagnostic procedures can make use of this information to perform what might be considered simple common sense tasks for an astronaut. For example, if a malfunction procedure had the test "Is the oxidant helium tank pressure greater than the fuel helium tank pressure for the front RCS?" our system could encode the test in a way that is impervious to system reconfiguration and is not hard-wired to particular identifiers. This is done using the process of unification – matching data base facts against queries that have a particular form. For this particular test, we might use the query:

```
(? (& (type rcs f $rcs-id)
      (type he-pressurization ox $hep-ox)
      (part-of $hep-ox $rcs-id)
      (type he-pressurization fuel $hep-fuel)
      (part-of $hep-fuel $rcs-id)
      (type he-tank $he-ox-tank)
      (part-of $he-ox-tank $hep-ox)
      (type he-tank $he-fuel-tank)
      (part-of $he-fuel-tank $hep-fuel)
      (pressure $he-ox-tank $ox-press)
      (pressure $he-fuel-tank $fuel-press)
      (> $ox-press $fuel-press)))
```

This type of conjunctive unification is actually used in the sample malfunction procedure discussed next.

4.2.2 The JET-FAIL-ON KA

Figure 4.4 shows a portion of the malfunction handling procedures for the RCS on the space shuttle. We will be concentrating on the procedure called `RCS JET FAIL (ON)`, which can be seen as Step 1.1 of Procedure 10.1, as well as 10.1a (only a portion of the entire malfunction procedure is shown in the figure). Notice how diagnostic conclusions (such as "JET DRIVER FAILED-ON ELECTRICALLY") are displayed in highlighted boxes.

In the PES implementation of these diagnostic procedures, the main top-level KA for dealing with the "JET FAIL (ON)" failure is called `JET-FAIL-ON` and is shown in

10.1 RCS JET/DLMA/PWR

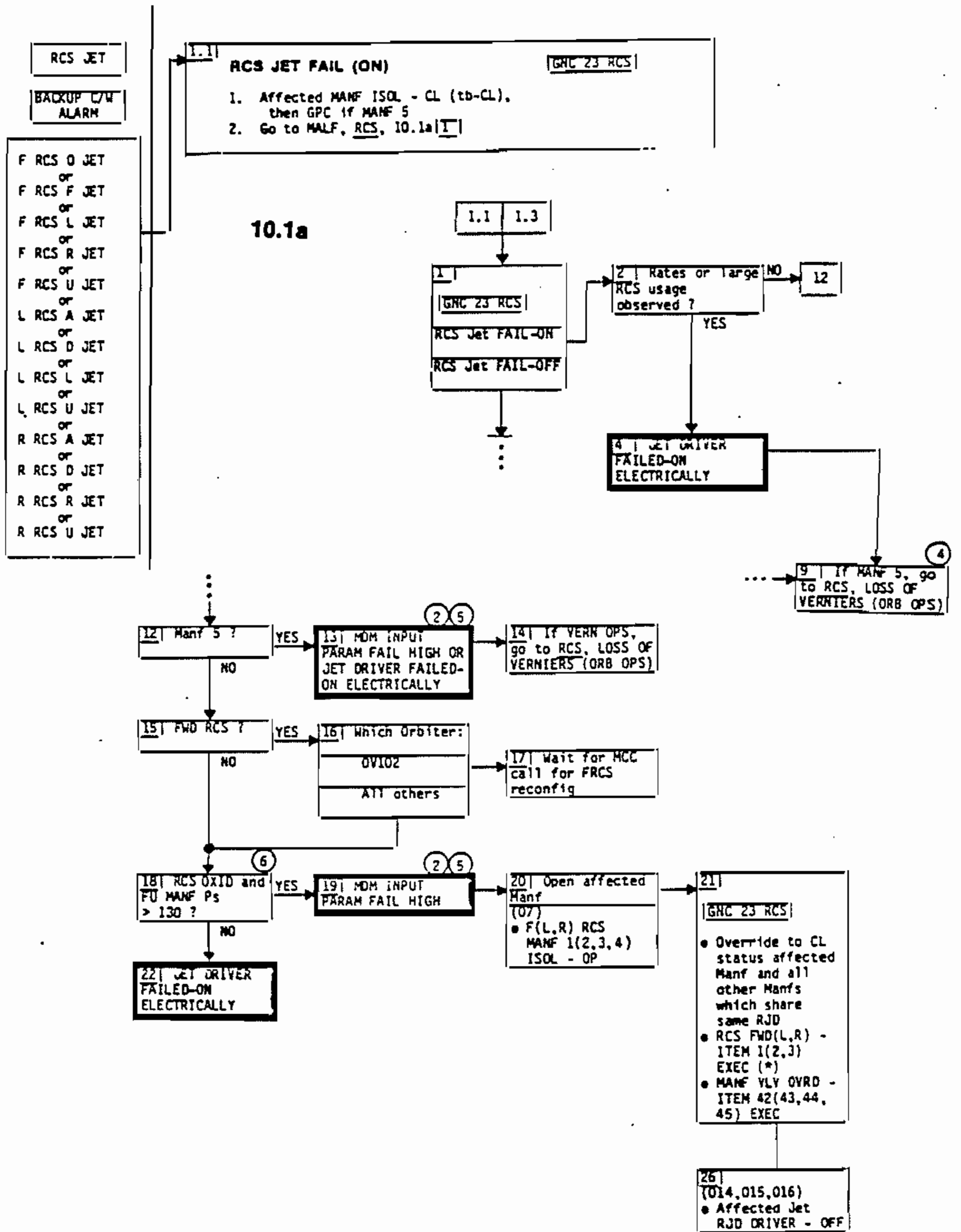


Figure 4.4: RCS JET FAIL (ON) Malfunction Procedure

Figure 4.5.[2] This KA is fact-invoked – that is, it responds when the system notices that certain lights, alarms, and computer monitor readings appear. For this reason, the invocation part of the JET-FAIL-ON KA has the form:

```
(AND (fact (light rcs-jet))
     (fact (alarm backup-cw))
     (fact (fault $rcs-id rcs $jet-id jet))
     (fact (jetfail-indicator on $manf-id)))
```

Thus, in order to get this particular RCS application running, these four facts (with instantiations of the three variables: `$rcs-id`, `$jet-id`, `$manf-id`) must be added to the system data base. For example, we might add the facts:

```
(light rcs-jet)
(alarm backup-cw)
(fault rcs.1 rcs thr.1.1 jet)
(jetfail-indicator on miv.1.1.1)
```

This tells the system that *there is* an actual malfunction in a specific reactant control subsystem (`rcs.1`), jet (`thr.1.1`), and manifold (`miv.1.1.1`) and the system will then react and proceed with the diagnosis procedure.

Starting at its START node, the JET-FAIL-ON KA execution will begin and try to traverse its first edge, labeled with the goal expression (`!(closed-manifold $manf-id)`) (see Figure 4.6). In other words, the system must find some way to close the given manifold. (This corresponds to the first step of the malfunction procedure in Figure 4.4, which reads: "Affected MANF ISOL - CL (tb-CL), then GPC if MANF 5." Notice how we have abstracted the overall *goal* of this step (to close the manifold) from a particular instruction in the malfunction book, which only states *how* to achieve the goal.

Moreover, in this case, there are two different ways of achieving the goal - for all manifolds, a talk-back switch is set to the closed position. For vernier manifolds (of type 5), a setting must also be made on the computer console. These two ways of achieving a behavior of form (`!(closed-manifold $manf-id)`) are reflected in the two KAs shown in Figure 4.7, CLOSED-MANIFOLD or CLOSED-MANIFOLD-VERNIER. As indicated

[2] All of the RCS procedures as well as the initial data base facts reflecting the structure of the RCS are given in the appendix.

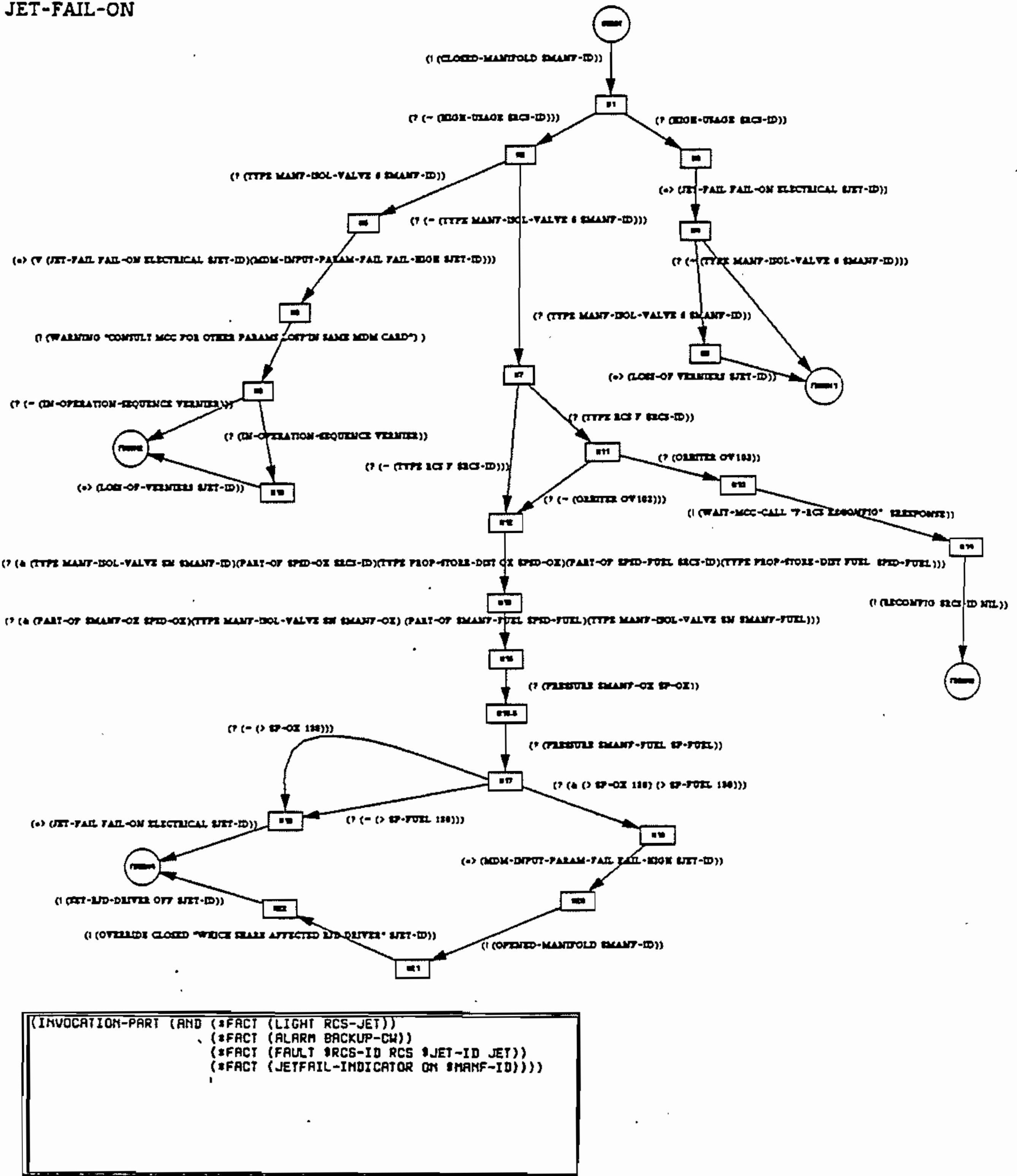


Figure 4.5: JET-FAIL-ON KA

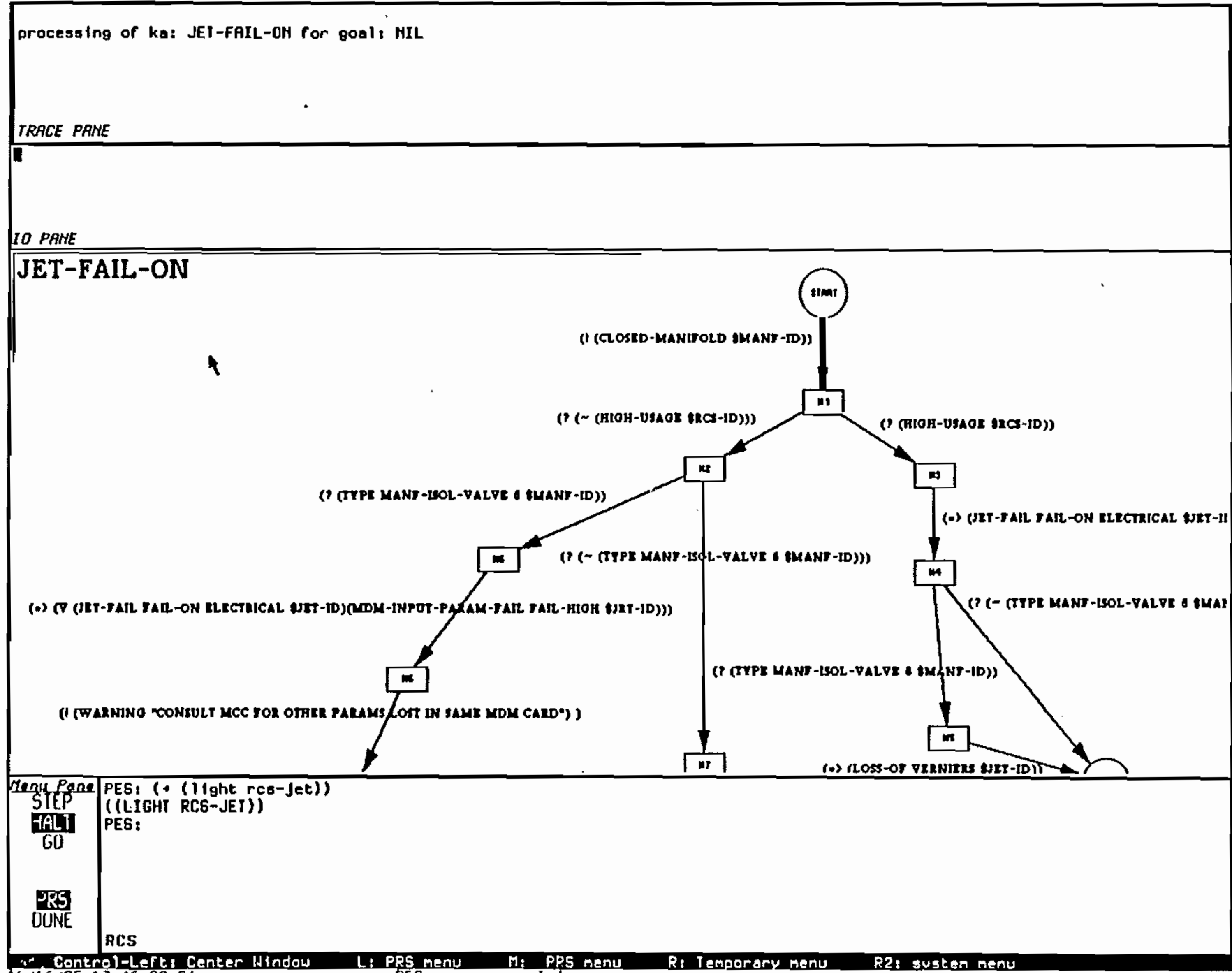


Figure 4.6: The CLOSED-MANIFOLD Step.

in their invocation parts, each responds to a goal of the form `(!(closed-manifold $manf-id))`. However, the invocation parts also constrain their applicability further – `CLOSED-MANIFOLD` will only be truly applicable if the manifold in question is not of type 5, and `CLOSED-MANIFOLD-VERNIER` will only be applicable if the manifold is of type 5.

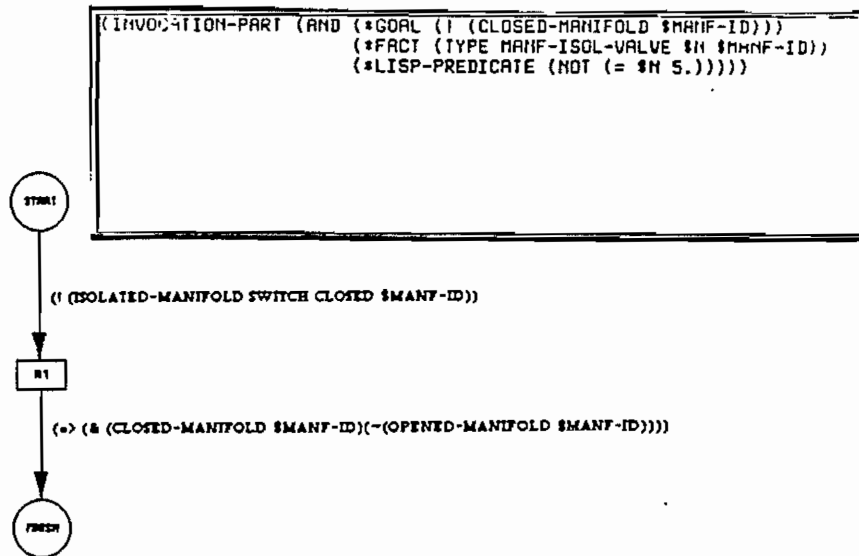
In this particular case then, both of these KAs will respond to the goal `(!(closed-manifold $manf-id))`, but only one of them will be truly applicable. Of course, for other situations and other goals, more than one KA may actually be applicable to a given goal. In these cases, metalevel KAs are used to resolve which KA is most useful in the particular situation. In some situations, a metalevel KA might decide to choose one of the applicable KAs at random, trying each of them till one succeeds (or till they all fail).

Of course, because of the semantics of our KAs, there is yet another way to achieve `(!(closed-manifold $manf-id))` besides executing KAs. In particular, a goal of the form (!*p*) will automatically be achieved if the system already believes that *p* is true. For our example case, if the system already has in its data base a fact of the form `(closed-manifold miv.1.1.1)` (in other words, it believes manifold `miv.1.1.1` to already be closed), a goal of the form `(!(closed-manifold miv.1.1.1))` will automatically succeed – no executions of the KAs `CLOSED-MANIFOLD` and `CLOSED-MANIFOLD-VERNIER` need be undertaken.

It is precisely the lack of this kind of goal semantics and reasoning ability that caused a recent space shuttle flight to abort. Although the shuttle system knew that a particular manifold was closed, it found itself unable to proceed when an instruction of the form "close the manifold" was given to it. This is because all of the manifold-closing procedures available to it *presumed* an open manifold – it could not close a manifold that was already closed! If the system had been structured properly (i.e., in terms of abstract goals and procedures, rather than as fixed hard-wired procedure calls), the shuttle system would have realized that its goal to close the manifold had already been achieved.

In the current PES implementation of the RCS, this very same situation actually comes into play. For example, in testing out the system, we often run through the `JET-FAIL-ON` diagnostic procedure several times. In the course of this procedure, affected manifolds are closed, and while in some circumstances they are reopened again, in

CLOSED-MANIFOLD



CLOSED-MANIFOLD-VERNIER

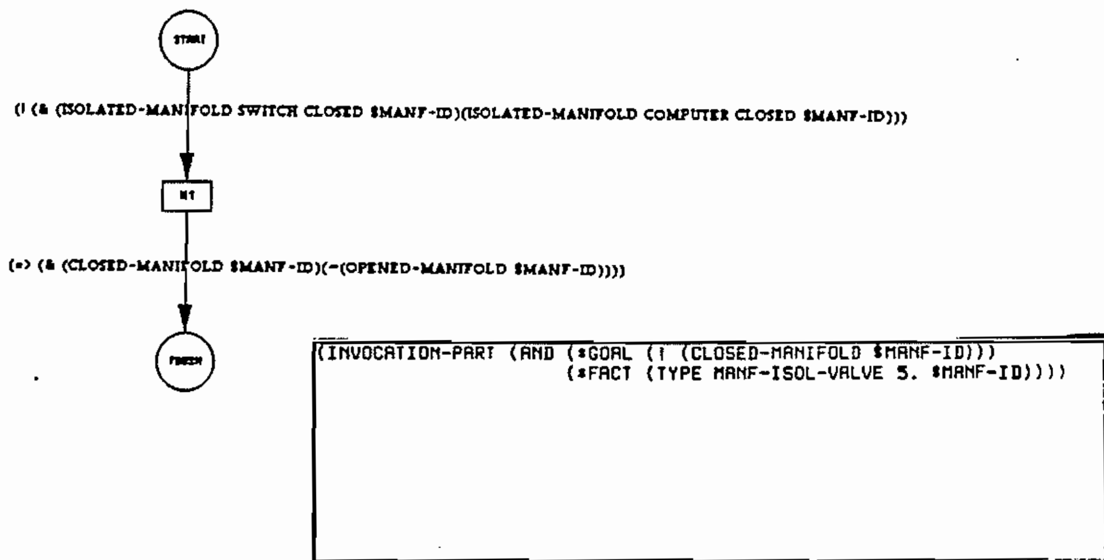


Figure 4.7: KAs for Closing a Manifold.

others they are not. When the diagnostic procedure is run more than once, it will *not* try to reachieve a closed manifold if that manifold had been closed and not reopened on the previous run. Thus, by encoding all of its knowledge in a perpetual, nonmonotonic data base, the system is able to remember and use its knowledge effectively.

Continuing with our execution of the `JET-FAIL-ON` KA, if the goal to close the manifold in question actually succeeds, the system will then move on to the next node and choose a new outgoing arc to traverse. One possible choice might be the arc labeled `(?(¬ (high-usage $rcs-id)))` – i.e., we establish the goal to determine whether there is *not* high usage (see Figure 4.8).

How does our system handle a goal of this form? First of all, it will check to see if there are any data base facts or KAs that match this goal precisely. In other words, because we can have negated facts in the system data base, it is possible that a fact of form `(¬ (high-usage rcs.1))` *is* present in the data base. Similarly, there may be a KA with an invocation part that indicates it is useful for precisely a goal of the form `(?(¬ (high-usage $rcs-id)))`. If a matching data base fact or a successful matching KA are found, then the goal will be satisfied in this way. However, if no such fact or matching KA is present, the system will try to achieve the goal using any other means at its disposal.

For goals composed of negated predicates, a metalevel KA is available that tries to achieve the goal using the rule of "negation as failure." In other words, for a goal of form $(!(\neg p))$ or $(?(\neg p))$, the metalevel KA will try to achieve $(!p)$ (or $(?p)$) and, if it fails to do so, will assume that the original negated goal has succeeded. Other metalevel KAs also exist for achieving a conjunct of goals (in our current system, this metalevel KA tries to achieve each of the conjuncts, successively, till all succeed or one fails), as well as a disjunct of goals (this metalevel KA tries to achieve each of the disjuncts, successively, till one succeeds or all of them fail). Of course, one might imagine other ways to achieve negated goals, conjuncts of goals, or disjuncts of goals. These new methods may easily be added to the system as new metalevel KAs. For example, one such KA might achieve a conjunct of goals by trying to achieve all of them in parallel.

Returning to the goal `(?(¬ (high-usage $rcs-id)))`, our current system will actually use the negation-as-failure metalevel KA. This KA will set up a goal of form `(?(high-usage $rcs-id))`. This particular goal will then be achieved by a KA that

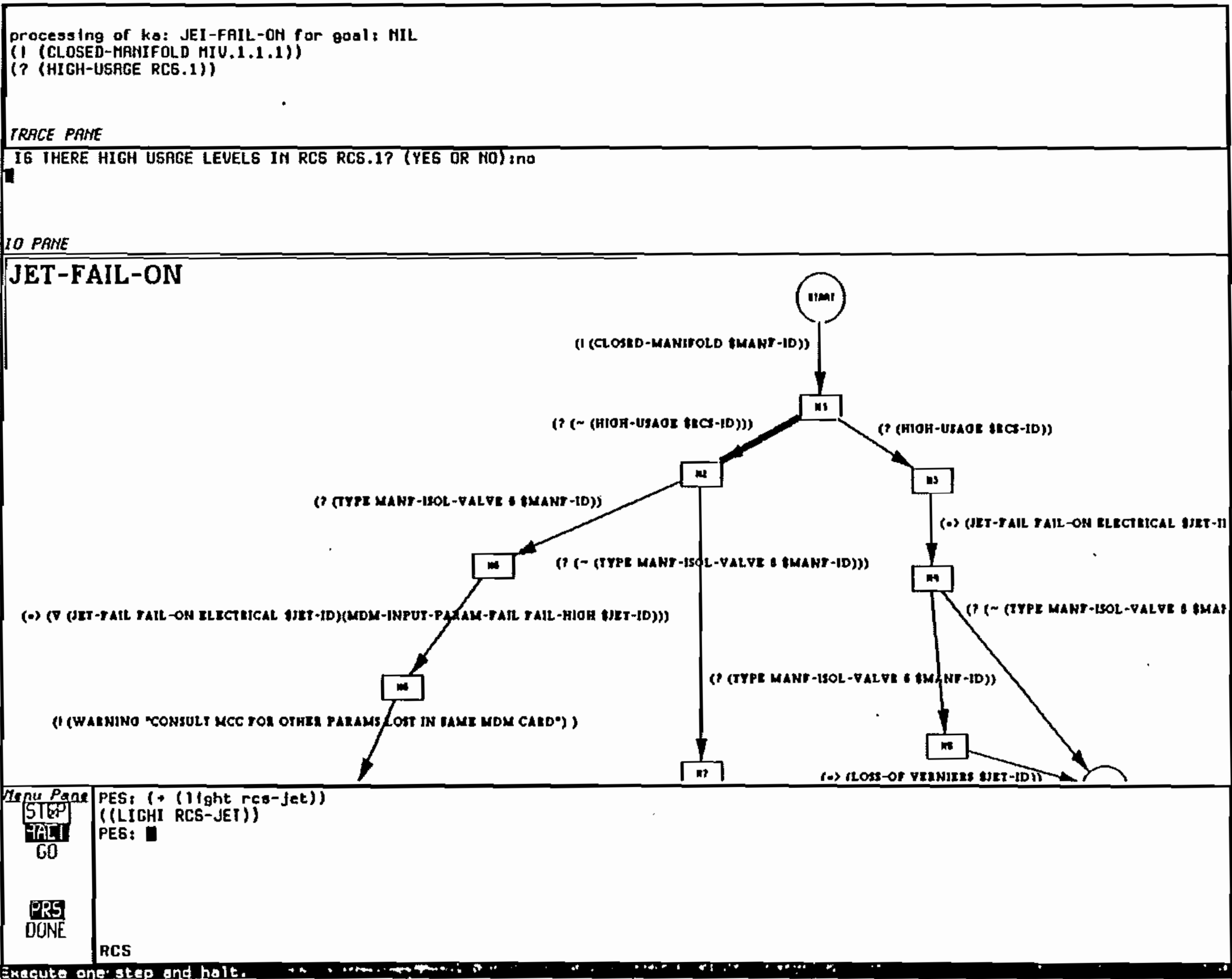


Figure 4.8: HIGH-USAGE Step.

asks the user the question "Are there high usage levels in RCS.1? (yes or no):" (once again, see Figure 4.8). If the user responds "no," the `HIGH-USAGE` KA will fail, and the original metalevel KA will decide that the goal `(?(¬ (high-usage $rcs-id)))` succeeded.

The `JET-FAIL-ON` KA might next proceed to ensuing goals of the form

```
(?  (¬ (type manf-isol-valve 5 $manf-id))),
(?  (type rcs f $rcs-id)),
(?  (¬ (orbiter ov102))),
```

which are all handled in routine ways (using data base facts, negation as failure, etc.). The next two arcs along this path are labeled with large conjunctive goals (see Figure 4.9). In fact, both of these arcs are conjuncts of facts in the data base.

To handle a conjunctive goal of this form, the system will first test to see if all of the conjuncts are facts. If this is true, it will achieve the goal via unification. (This is precisely what is done for this case.) In other cases, the system will first try to match the conjunct exactly against the invocation parts of KAs (to see if there is a KA that achieves precisely this conjunctive goal). If this too fails, it will resort to the conjunctive metalevel KA described above.

Returning to the example, the conjunction of facts depicted in Figure 4.9 is being used to find two particular manifolds in the system. The unification is set up much as for the example given earlier in our description of the RCS data base. In this particular instance, we are trying to find the two manifolds that meet the description: "RCS OXID and FU MANF" (see Figure 4.4). In the context of the entire procedure, a human would know that what is meant is the particular oxidant and fuel manifolds corresponding to the currently faulty manifold, but a machine is not so smart. The unification of facts must thus use the identity of the faulty manifold in its search for the corresponding fuel and oxidant manifolds. This "correspondence" is very much tied in to the structure of the RCS itself.

To conclude this particular run of the `JET-FAIL-ON` KA, if the pressure in one of the two fuel and oxidant manifolds was found to be less than 130 units, the system will diagnose the failure as an electrical failure of a particular jet (see Figure 4.10). If, on the other hand, the pressure in both is greater than 130, the diagnosis will be an MDM input parameter failure, and various manifolds and settings will be readjusted (see other path in Figure 4.10).

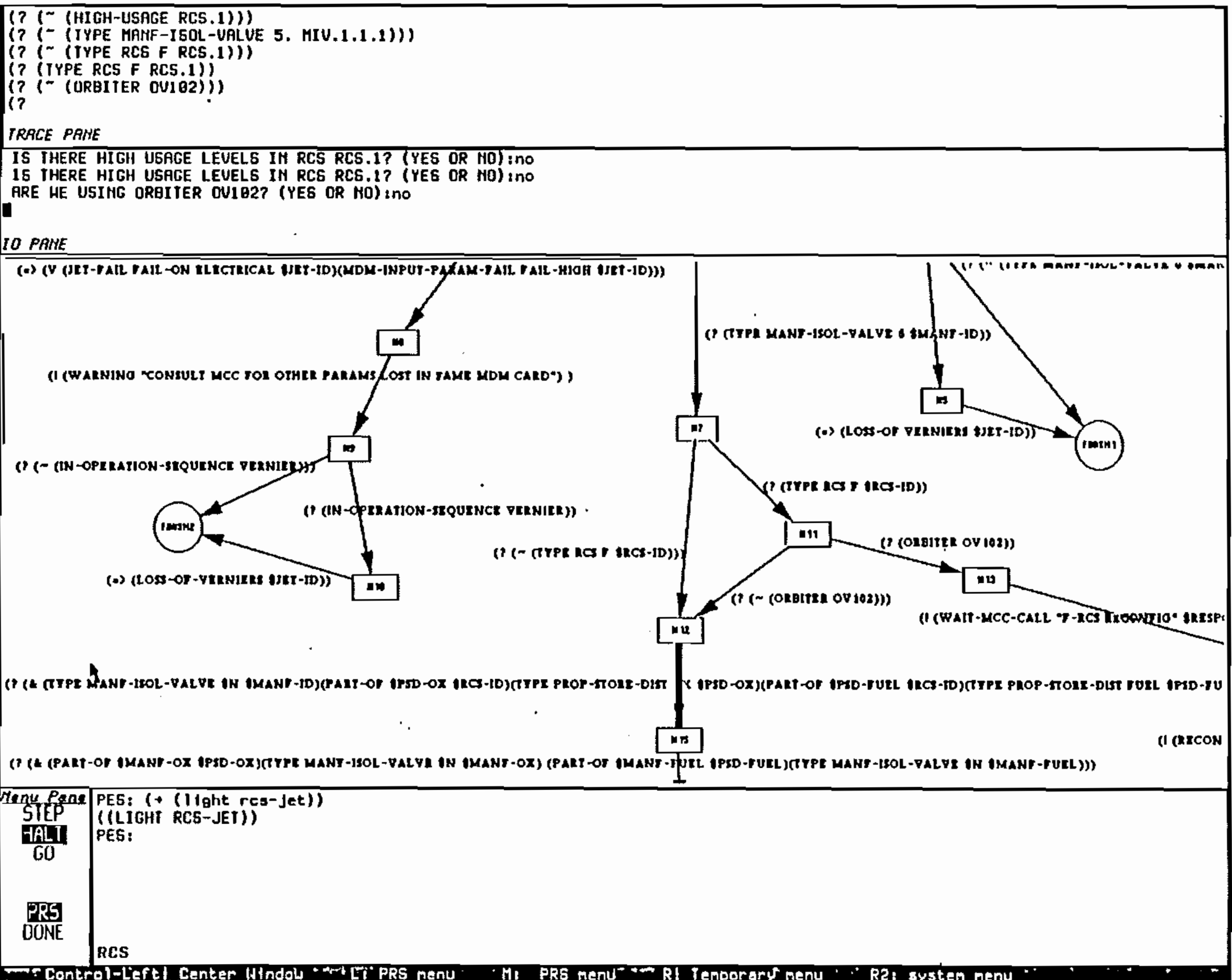


Figure 4.9: Unification of a Large Conjunct of Facts.

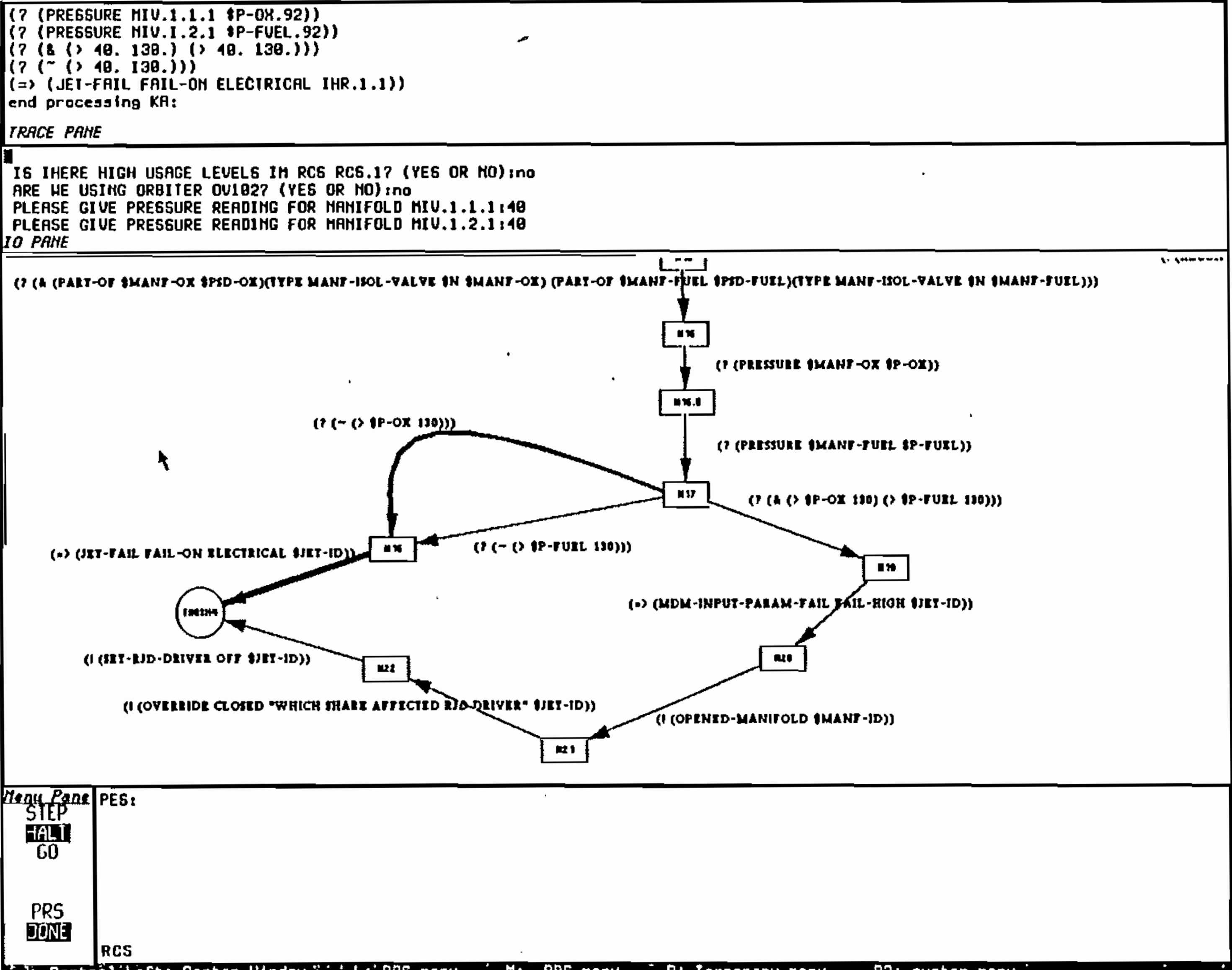


Figure 4.10: End of the JET-FAIL-ON KA.

From these few examples, we can see that the PES data base plays a large role in the diagnosis process. As another illustration of the use of the PES data base, consider what happens if a particular action results in some reconfiguration of the components of the RCS. If such an action were undertaken and overseen by PES, the new structure of the RCS (e.g., the way its components are interconnected) would be encoded in the PES data base and remembered. These facts may later be quite relevant when performing other tasks on the system. Moreover, if a new configuration is nonstandard in any way, an astronaut might forget the particular details of this configuration and not perform the malfunction procedures correctly or effectively.

For example, in Malfunction Procedure 1.4 (RCS LEAK ISOL) Step 6 (see Figure 4.3), we have the instruction "If Aft RCS, I'CNECT [interconnect] from OMS ... then open all MANFs. Prior to deorbit TIG return to straight RCS feed." Astronauts would have to remember that they had reconfigured the system for this particular case and later, upon deorbit, return to straight RCS feed. It is easy to see that PES would probably perform more adequately than a human in these circumstances. The new nonstandard configuration would be stored in the data base. A fact-invoked KA could then be used to trigger return to straight feed in the precise situation where the system is in this particular configuration and deorbit is about to begin.

Of course, there are many cases where we would expect our system to perform less knowledgeably than a human. For example, this might be true in the case where the malfunction procedures had actually failed to yield any particular results and an astronaut was forced to reason about the system "from first principles." Unless extensive knowledge was encoded into an elaborate system of deductive procedures and the content of the PES data base description was greatly enhanced, the PES system would be less effective than a human in this type of reasoning.

One aid to the astronaut in such a reasoning process, however, is the procedural nature of KAs and their graphical representation. Procedures are presented as meaningful entities rather than as sets of disjoint and seemingly unconnected rules. Because the purpose of each procedure and each step of a procedure is described abstractly (as goal descriptions), an astronaut might easily see another way of achieving a particular goal that could be used, or why a particular diagnostic failed. The graphical presentation of procedures also makes them easy to understand and their execution easy to follow. Thus, graphically represented KAs represent a powerful *explanation* facility for any form of procedural reasoning.

Chapter 5

Reasoning about Procedures

One of the crucial features of procedural expert systems is that the procedural representation has a well-defined declarative semantics. In this chapter, we discuss how this semantics enables us to reason about procedures and the behaviors they generate in quite general ways.

5.1 Metalevel Reasoning

When more than one KA responds to a goal or when a data-driven KA interrupts a goal-directed KA that is being executed, we need some means for deciding which of all the applicable alternatives we should execute next. When reasoning about a static world, one can often get by with relatively simple schemes. For example, Prolog uses depth-first search and considers alternatives in their lexical order. Concurrent evaluation may also be possible. In dynamic worlds, however, more powerful reasoning abilities are required: changes effected in the state of the world by one course of action may preclude backtracking to others, while interference among actions can make parallel exploration of all alternatives impossible.

Reasoning about the appropriateness of *sequences* of actions is particularly difficult in rule-based formalisms. Because the rules making up a procedure are ostensibly independent pieces of knowledge, there is no sensible way to reason about the procedure as a whole. The problem is that such information does not apply to the rules as

independent individuals, but to the procedure as a whole; thus, it cannot be attached sensibly to any one rule.

For a rule-based system, the best one can do is to attach to each rule information about its execution (in the given context). Unfortunately, such information is often not very useful in determining which *procedure* is the most suitable. For example, to ascertain whether or not a patient has a certain disease, one option may be to perform a series of quite expensive diagnostic tests, while another may require surgical examination. The individual steps leading up to the surgery may well be cheaper than each of the diagnostic tests and, under a rule-based system, one would be led to the point of incision before discovering the true cost of the chosen procedure.

Procedural expert systems present no such difficulties: because KAs represent entire procedures, we can reason directly about the procedures as single entities. One approach is to have the interpreter make a careful choice as to which KA to process at any given stage of execution. This could be achieved by giving applicable KAs priority levels or importance measures (such as those used in AM [20]), then having the interpreter select for execution whichever KA has the highest priority or greatest importance.

However, the importance or utility of a KA is often context-dependent and qualitative in nature. This kind of information is difficult to represent using simple numerical priorities. It is therefore better to represent the knowledge about selection of KAs in some logical form, allowing the system to *reason* about what is best to do next. Indeed, such knowledge is an essential part of an expert's understanding of a problem domain. We shall call such knowledge *metalevel*knowledge, because the entities it describes and manipulates are the *object-level* KAs representing the physics of the problem domain.

Much of this metalevel knowledge is also procedural in nature (for example, the KA interpreter itself can be viewed as a metalevel procedure). It is thus desirable that this metalevel knowledge be represented in the same form as the object-level knowledge [11,19]. As Hayes says [16] : "We need to be able to describe processing strategies in a language at least as rich as that in which we describe the external domains and, for good engineering, it should be the same language." Therefore, we allow *metalevel KAs* to describe and manipulate object-level KAs, much in the same way that object-level KAs describe and manipulate entities in the problem domain. In fact, the system interpreter does not even distinguish between these two kinds of KAs.

We also impose no restriction upon the number of metalevels; for example, we allow metametalevel KAs to operate on metalevel KAs. In fact, we even allow metalevel KAs to reflect upon themselves, and we mix meta- and object-level KAs quite freely (making sure, of course, that each KA is independently valid).

Still, we need to provide these metalevel KAs with information upon which they can base an assessment of the relative utilities of a set of KAs. Such information would include estimated costs (in time, space, and dollars), criticality (e.g., emergency procedures), and the probability of success in attaining given goals. These [metalevel] facts could be entered into the data base along with all the other facts about the problem domain. However, since they are known at the time the KAs to which they refer are created, it is convenient to attach these facts directly to the KAs themselves. We call such a collection of facts the *information part* of a KA.

There are a number of problems that have to be solved before such a scheme can be usefully implemented. Most of these involve issues of efficiency and the need to determine how much expressive power is required at the metalevel. Others involve issues of consistency. For example, if the metalevel were to have unfettered access to the data base, it could add and delete arbitrary facts. Clearly, this could be catastrophic. What restrictions should we therefore impose on the metalevel to ensure that these changes would be consistent with the object-level theory (as represented by object-level KAs)?

5.2 Reasoning about Complex Goals

One of the important features of the system is that it can *reason* about how to achieve complex composite goals. For example, a conjunctive goal such as

```
(!((p a b) ∧ (q a b)))
```

may be set by the user or may appear as a subgoal labeling the arc of some KA.

Often, there will be no KA in the system that directly unifies with such a composite goal. However, because our notion of goal has a well-defined semantics, the system can reason about how it can achieve composite goals by trying to achieve the simpler component goals. For example, the system could reason that a given conjunctive goal could be realized by achieving all the component goals, one after the other.

The rules for how composite goals can be decomposed into simpler ones follows directly from the semantics given to goal descriptions. We will use the notation $\langle P \rangle\ (\sigma)$ to mean that *every* successful behaviour associated with the KA P satisfies the temporal assertion σ. $\langle P \rangle_F$ denotes failed behaviors. The symbols ";" and "|" represent sequential composition and [nondeterministic] choice, respectively.

Some typical proof rules are as follows:

Conjunctive Testing

$$\frac{\langle P_1 \rangle (?p) \wedge \langle P_1 \rangle (\#q) \wedge \langle P_2 \rangle (?q)}{\langle P_1\ ;\ P_2 \rangle\ (?(p \wedge q))}$$

Conjunctive Achievement

$$\frac{\langle P_1 \rangle (!p) \wedge \langle P_2 \rangle (!q) \wedge \langle P_2 \rangle (\#p)}{\langle P_1 ; P_2 \rangle (!(p \wedge q))}$$

Disjunctive Testing

$$\frac{\langle P_1 \rangle (?p) \wedge \langle P_2 \rangle (?q) \wedge \langle P_1 \rangle_F (\#q) \wedge \langle P_2 \rangle_F (\#p)}{\langle P_1 \mid P_2 \rangle (?(p \vee q))}$$

Disjunctive Achievement

$$\frac{\langle P_1 \rangle (!p) \wedge \langle P_2 \rangle (!q)}{\langle P_1 \mid P_2 \rangle (!(p \vee q))}$$

The first rule states that, to *test* for a condition $(p \wedge q)$, one way is first to test for p using a KA that does not modify q, then subsequently to test for q. The second states that to *achieve* a condition $(p \wedge q)$, one way is to first achieve p and then to achieve q without affecting p. The disjunctive rules simply say that to test for or to achieve $(p \vee q)$ we simply need try each of the disjuncts, though in testing for $(p \vee q)$ we must, in addition, be careful that any *failed* attempt to test one of the disjuncts does not affect the other.

Note that these proof rules are not the only ones, nor are they the strongest, that could be used. For example, in the rule for conjunctive achievement, we need not require that p be unaffected by $\langle P_2 \rangle$; all we need do is regress the goal $!p$ through $\langle P_2 \rangle$

and set this as the goal of $\langle P_1 \rangle$. However, since, in most real-world cases, it is difficult to regress conditions through complex sequences of actions, the rules given above prove to be most practical.

These rules are represented in the current system by metalevel KAs. Thus, if no KAs respond directly to an extant composite goal, the metalevel KAs corresponding to the appropriate decomposition rules will be invoked to break down the complex goal into a sequence of simpler goals.

Representing decomposition rules in the form of metalevel KAs provides the system with enormous flexibility. It allows users to add or delete such rules as they find appropriate. Moreover, in this way it is even possible to add domain-specific decomposition rules to the system.

Other rules can also be represented by metalevel KAs. For example, the current system allows the user to specify that the closed-world assumption [24] applies to various state predicates, and implements this default assumption by means of a metalevel KA.

Chapter 6

Conclusions

We have described a reactive system for reasoning about and performing complex tasks in dynamic environments, and have devised a theoretically sound scheme for representing and reasoning about the kind of procedural knowledge typical of dynamic problem domains. A declarative semantics for the representation has been constructed that allows a user to specify *facts* about behaviors independently of context. We have also defined an operational semantics that shows *how* these facts can be used by a system to achieve desired operational goals. Possession of both a declarative and an operational semantics is an essential precondition of a system endowed with all the desirable properties of expert systems, including explanatory capability, reasoning ability, evolutionary potential, and verifiability. The system also includes powerful metalevel reasoning capabilities, using the same formalism for representing this knowledge as for object-level knowledge. We have constructed a practical implementation of a system based on this representation, and shown how it can be applied to certain problems in the automation of space operations.

Much more work remains to be done. We need to consider planning [23] and consistency maintenance [5]. We should also investigate concurrency, and extend the model to deal with it. Some work in this direction is described by us elsewhere [13].

Acknowledgments

There have been a number of people involved in implementing and testing the system described herein, including Pierre Bessiere, Marcel Schoppers, Joshua Singer, and Mabry Tyson. We are most grateful for their contributions. We also wish to thank Oscar Firschein for his critical reading of this report and many helpful suggestions.

Appendix A

Sample Knowledge Base for the RCS

In this section we present a sample of the data base facts and KAs used to represent some of the malfunction-handling procedures for the RCS system. They represent a first attempt at formalizing the domain; considerable work with mission controllers and other experts is needed before a realistic formalization can be developed.

A.1 Glossary of Identifier Prefixes

Individual elements in the system are represented by unique identifiers. These are of the form ⟨*word*⟩.⟨*number*⟩.⟨*number*⟩..., and are named in such a way to given some intuition about the type of object and its location.

```
RCS RCS
HEP Helium Pressurization system
PSD Propellant Storage and Distribution
THR Thruster
HET Helium Tank
HEV Helium Pressure Valve
REG Regulator
```

```
CHK Check Valve
REL Relief Valve
OXT Oxygen Tank
FUT Fuel Tank
TIV Tank Isolation Valve
MIV Manifold Isolation Valve
BIV Bipropellant Valve
XFV Crossfeed Valve
```

A.2 RCS State Description (Initial Data base)

The following are some of the facts stored within the RCS data base. They represent the basic structure of a portion of the RCS in its standard configuration (see Figure 4.2).

A.2.1 Top Level Reactant Control Systems

```
(TYPE RCS F RCS.1)
(TYPE RCS L RCS.2)
(TYPE RCS R RCS.3)
```

A.2.2 Basic Components of Forward RCS

```
(TYPE HE-PRESSURIZATION OX   HEP.1.1)
(TYPE HE-PRESSURIZATION FUEL HEP.1.2)
(PART-OF HEP.1.1 RCS.1)
(PART-OF HEP.1.2 RCS.1)

(TYPE PROP-STORE-DIST OX   PSD.1.1)
(TYPE PROP-STORE-DIST FUEL PSD.1.2)
(PART-OF PSD.1.1 RCS.1)
(PART-OF PSD.1.2 RCS.1)
```

```
(TYPE THRUSTER PRIMARY 1 L THR.1.1)
(TYPE THRUSTER PRIMARY 1 U THR.1.2)
(TYPE THRUSTER PRIMARY 1 D THR.1.3)
(TYPE THRUSTER PRIMARY 1 F THR.1.4)
(TYPE THRUSTER PRIMARY 2 R THR.1.5)
(TYPE THRUSTER PRIMARY 2 U THR.1.6)
(TYPE THRUSTER PRIMARY 2 D THR.1.7)
(TYPE THRUSTER PRIMARY 2 F THR.1.8)
(TYPE THRUSTER PRIMARY 3 L THR.1.9)
(TYPE THRUSTER PRIMARY 3 U THR.1.10)
(TYPE THRUSTER PRIMARY 3 D THR.1.11)
(TYPE THRUSTER PRIMARY 3 F THR.1.12)
(TYPE THRUSTER VERNIER 4 R THR.1.13)
(TYPE THRUSTER VERNIER 4 D THR.1.14)
(TYPE THRUSTER VERNIER 5 L THR.1.15)
(TYPE THRUSTER VERNIER 5 R THR.1.16)

(PART-OF THR.1.1 RCS.1)
(PART-OF THR.1.2 RCS.1)
(PART-OF THR.1.3 RCS.1)
(PART-OF THR.1.4 RCS.1)
(PART-OF THR.1.5 RCS.1)
(PART-OF THR.1.6 RCS.1)
(PART-OF THR.1.7 RCS.1)
(PART-OF THR.1.8 RCS.1)
(PART-OF THR.1.9 RCS.1)
(PART-OF THR.1.10 RCS.1)
(PART-OF THR.1.11 RCS.1)
(PART-OF THR.1.12 RCS.1)
(PART-OF THR.1.13 RCS.1)
(PART-OF THR.1.14 RCS.1)
(PART-OF THR.1.15 RCS.1)
(PART-OF THR.1.16 RCS.1)
```

A.2.3 Helium Pressurization System Of Forward RCS

```
(TYPE HE-TANK HET.1.1.1)
(PART-OF HET.1.1.1 HEP.1.1)

(TYPE HE-TANK HET.1.2.1)
(PART-OF HET.1.2.1 HEP.1.2)

(TYPE HE-PRESS-VALVE A HEV.1.1.1)
(TYPE HE-PRESS-VALVE B HEV.1.1.2)
(PART-OF HEV.1.1.1 HEP.1.1)
(PART-OF HEV.1.1.2 HEP.1.1)

(TYPE HE-PRESS-VALVE A HEV.1.2.1)
(TYPE HE-PRESS-VALVE B HEV.1.2.2)
(PART-OF HEV.1.2.1 HEP.1.2)
(PART-OF HEV.1.2.2 HEP.1.2)

(TYPE REGULATOR A 1 REG.1.1.1)
(TYPE REGULATOR A 2 REG.1.1.2)
(TYPE REGULATOR B 1 REG.1.1.3)
(TYPE REGULATOR B 2 REG.1.1.4)
(PART-OF REG.1.1.1 HEP.1.1)
(PART-OF REG.1.1.2 HEP.1.1)
(PART-OF REG.1.1.3 HEP.1.1)
(PART-OF REG.1.1.4 HEP.1.1)

(TYPE REGULATOR A 1 REG.1.2.1)
(TYPE REGULATOR A 2 REG.1.2.2)
(TYPE REGULATOR B 1 REG.1.2.3)
(TYPE REGULATOR B 2 REG.1.2.4)
(PART-OF REG.1.2.1 HEP.1.2)
(PART-OF REG.1.2.2 HEP.1.2)
(PART-OF REG.1.2.3 HEP.1.2)
(PART-OF REG.1.2.4 HEP.1.2)
```

```
(TYPE CHECK 1 CHK.1.1.1)
(TYPE CHECK 2 CHK.1.1.2)
(TYPE CHECK 3 CHK.1.1.3)
(TYPE CHECK 4 CHK.1.1.4)
(PART-OF CHK.1.1.1 HEP.1.1)
(PART-OF CHK.1.1.2 HEP.1.1)
(PART-OF CHK.1.1.3 HEP.1.1)
(PART-OF CHK.1.1.4 HEP.1.1)

(TYPE CHECK 1 CHK.1.2.1)
(TYPE CHECK 2 CHK.1.2.2)
(TYPE CHECK 3 CHK.1.2.3)
(TYPE CHECK 4 CHK.1.2.4)
(PART-OF CHK.1.2.1 HEP.1.2)
(PART-OF CHK.1.2.2 HEP.1.2)
(PART-OF CHK.1.2.3 HEP.1.2)
(PART-OF CHK.1.2.4 HEP.1.2)

(TYPE RELIEF REL.1.1.1)
(PART-OF REL.1.1.1 HEP.1.1)

(TYPE RELIEF REL.1.2.1)
(PART-OF REL.1.2.1 HEP.1.2)
```

A.2.4 Propellant Distribution System Of Forward RCS

```
(TYPE OX-TANK OXT.1.1.1)
(PART-OF OXT.1.1.1 PSD.1.1)

(TYPE FUEL-TANK FUT.1.2.1)
(PART-OF FUT.1.2.1 PSD.1.2)

(TYPE TANK-ISOL-VALVE 1/2   TIV.1.1.1)
```

```
(TYPE TANK-ISOL-VALVE 3/4/5 TIV.1.1.2)
(PART-OF TIV.1.1.1 PSD.1.1)
(PART-OF TIV.1.1.2 PSD.1.1)

(TYPE TANK-ISOL-VALVE 1/2   TIV.1.2.1)
(TYPE TANK-ISOL-VALVE 3/4/5 TIV.1.2.2)
(PART-OF TIV.1.2.1 PSD.1.2)
(PART-OF TIV.1.2.2 PSD.1.2)

(TYPE MANF-ISOL-VALVE 1 MIV.1.1.1)
(TYPE MANF-ISOL-VALVE 2 MIV.1.1.2)
(TYPE MANF-ISOL-VALVE 3 MIV.1.1.3)
(TYPE MANF-ISOL-VALVE 4 MIV.1.1.4)
(TYPE MANF-ISOL-VALVE 5 MIV.1.1.5)
(PART-OF MIV.1.1.1 PSD.1.1)
(PART-OF MIV.1.1.2 PSD.1.1)
(PART-OF MIV.1.1.3 PSD.1.1)
(PART-OF MIV.1.1.4 PSD.1.1)
(PART-OF MIV.1.1.5 PSD.1.1)

(TYPE MANF-ISOL-VALVE 1 MIV.1.2.1)
(TYPE MANF-ISOL-VALVE 2 MIV.1.2.2)
(TYPE MANF-ISOL-VALVE 3 MIV.1.2.3)
(TYPE MANF-ISOL-VALVE 4 MIV.1.2.4)
(TYPE MANF-ISOL-VALVE 5 MIV.1.2.5)
(PART-OF MIV.1.2.1 PSD.1.2)
(PART-OF MIV.1.2.2 PSD.1.2)
(PART-OF MIV.1.2.3 PSD.1.2)
(PART-OF MIV.1.2.4 PSD.1.2)
(PART-OF MIV.1.2.5 PSD.1.2)
```

A.2.5 Thruster System Of Forward RCS

```
(TYPE BIPROP-VALVE OX   BIV.1.1.1)
(TYPE BIPROP-VALVE FUEL BIV.1.1.2)
(PART-OF BIV.1.1.1 THR.1.1)
(PART-OF BIV.1.1.2 THR.1.1)

(TYPE BIPROP-VALVE OX   BIV.1.2.1)
(TYPE BIPROP-VALVE FUEL BIV.1.2.2)
(PART-OF BIV.1.2.1 THR.1.2)
(PART-OF BIV.1.2.2 THR.1.2)

(TYPE BIPROP-VALVE OX   BIV.1.3.1)
(TYPE BIPROP-VALVE FUEL BIV.1.3.2)
(PART-OF BIV.1.3.1 THR.1.3)
(PART-OF BIV.1.3.2 THR.1.3)

(TYPE BIPROP-VALVE OX   BIV.1.4.1)
(TYPE BIPROP-VALVE FUEL BIV.1.4.2)
(PART-OF BIV.1.4.1 THR.1.4)
(PART-OF BIV.1.4.2 THR.1.4)

(TYPE BIPROP-VALVE OX   BIV.1.5.1)
(TYPE BIPROP-VALVE FUEL BIV.1.5.2)
(PART-OF BIV.1.5.1 THR.1.5)
(PART-OF BIV.1.5.2 THR.1.5)

(TYPE BIPROP-VALVE OX   BIV.1.6.1)
(TYPE BIPROP-VALVE FUEL BIV.1.6.2)
(PART-OF BIV.1.6.1 THR.1.6)
(PART-OF BIV.1.6.2 THR.1.6)

(TYPE BIPROP-VALVE OX   BIV.1.7.1)
(TYPE BIPROP-VALVE FUEL BIV.1.7.2)
(PART-OF BIV.1.7.1 THR.1.7)
```

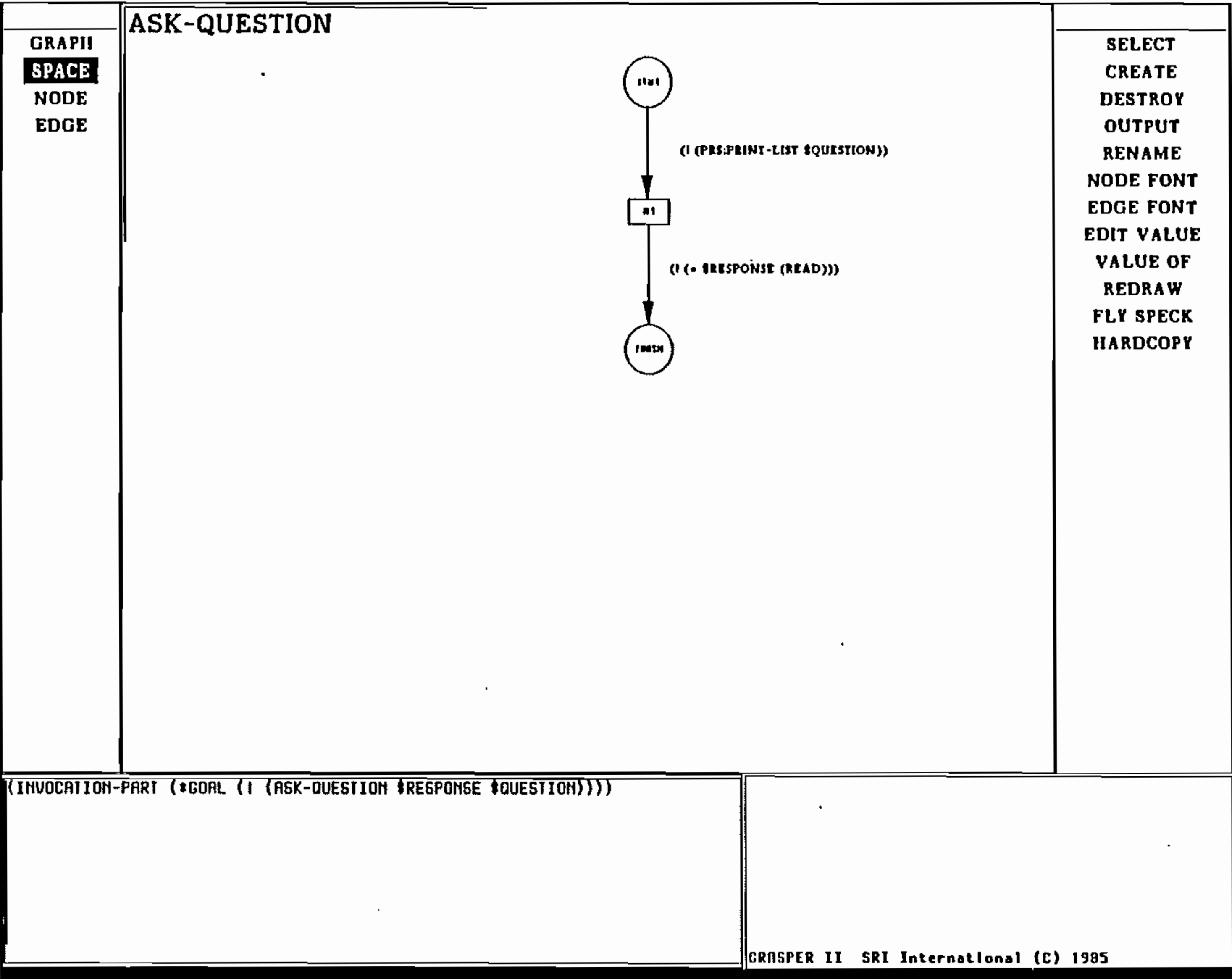
```
(PART-OF BIV.1.7.2 THR.1.7)

(TYPE BIPROP-VALVE OX   BIV.1.8.1)
(TYPE BIPROP-VALVE FUEL BIV.1.8.2)
(PART-OF BIV.1.8.1 THR.1.8)
(PART-OF BIV.1.8.2 THR.1.8)

(TYPE BIPROP-VALVE OX   BIV.1.9.1)
(TYPE BIPROP-VALVE FUEL BIV.1.9.2)
(PART-OF BIV.1.9.1 THR.1.9)
(PART-OF BIV.1.9.2 THR.1.9)

(TYPE BIPROP-VALVE OX   BIV.1.10.1)
(TYPE BIPROP-VALVE FUEL BIV.1.10.2)
(PART-OF BIV.1.10.1 THR.1.10)
(PART-OF BIV.1.10.2 THR.1.10)

(TYPE BIPROP-VALVE OX   BIV.1.11.1)
(TYPE BIPROP-VALVE FUEL BIV.1.11.2)
(PART-OF BIV.1.11.1 THR.1.11)
(PART-OF BIV.1.11.2 THR.1.11)

(TYPE BIPROP-VALVE OX   BIV.1.12.1)
(TYPE BIPROP-VALVE FUEL BIV.1.12.2)
(PART-OF BIV.1.12.1 THR.1.12)
(PART-OF BIV.1.12.2 THR.1.12)

(TYPE BIPROP-VALVE OX   BIV.1.13.1)
(TYPE BIPROP-VALVE FUEL BIV.1.13.2)
(PART-OF BIV.1.13.1 THR.1.13)
(PART-OF BIV.1.13.2 THR.1.13)

(TYPE BIPROP-VALVE OX   BIV.1.14.1)
(TYPE BIPROP-VALVE FUEL BIV.1.14.2)
```

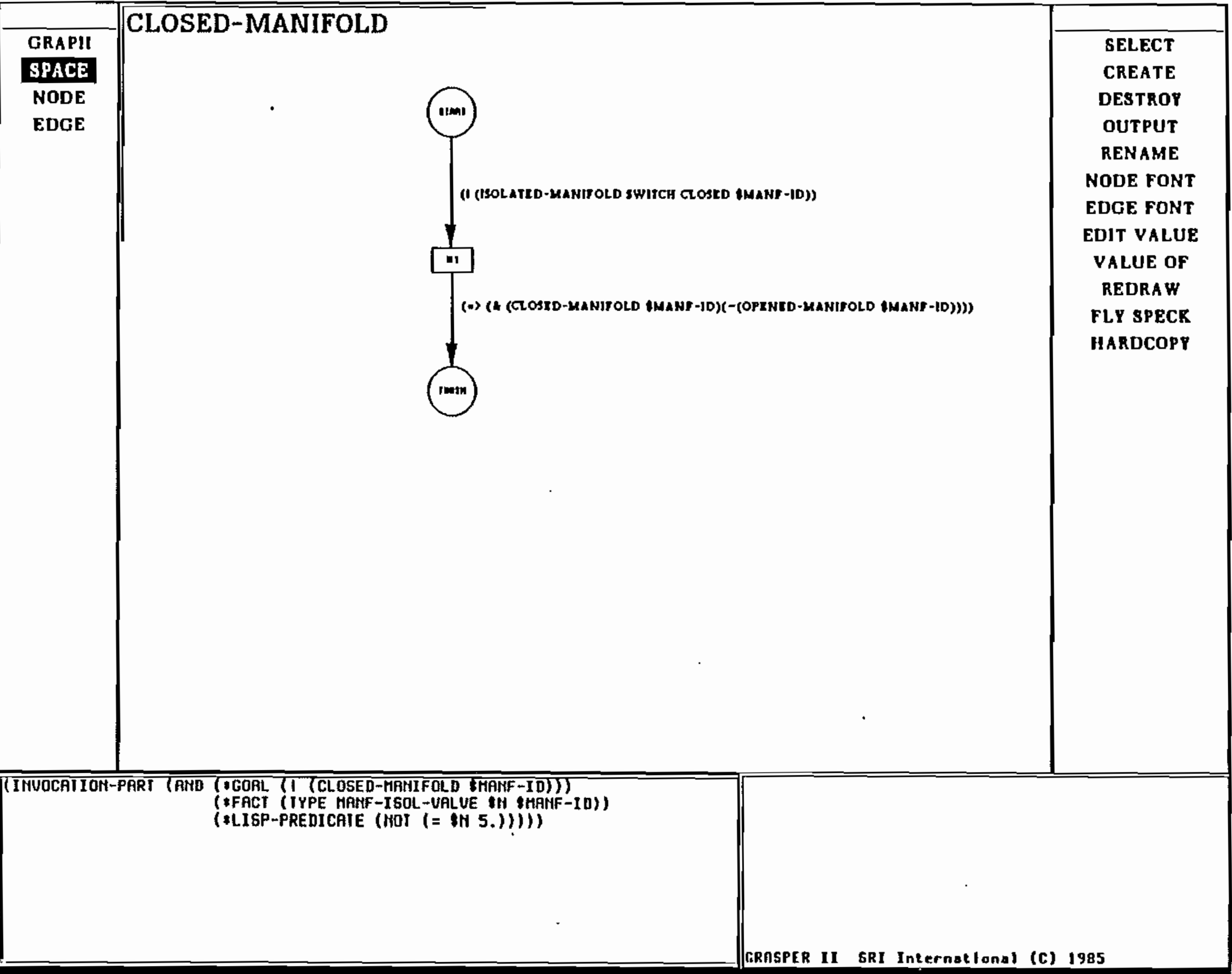
```
(PART-OF BIV.1.14.1 THR.1.14)
(PART-OF BIV.1.14.2 THR.1.14)

(TYPE BIPROP-VALVE OX   BIV.1.15.1)
(TYPE BIPROP-VALVE FUEL BIV.1.15.2)
(PART-OF BIV.1.15.1 THR.1.15)
(PART-OF BIV.1.15.2 THR.1.15)

(TYPE BIPROP-VALVE OX   BIV.1.16.1)
(TYPE BIPROP-VALVE FUEL BIV.1.16.2)
(PART-OF BIV.1.16.1 THR.1.16)
(PART-OF BIV.1.16.2 THR.1.16)
```

The data base also includes similar facts regarding the left and right aft RCSs.

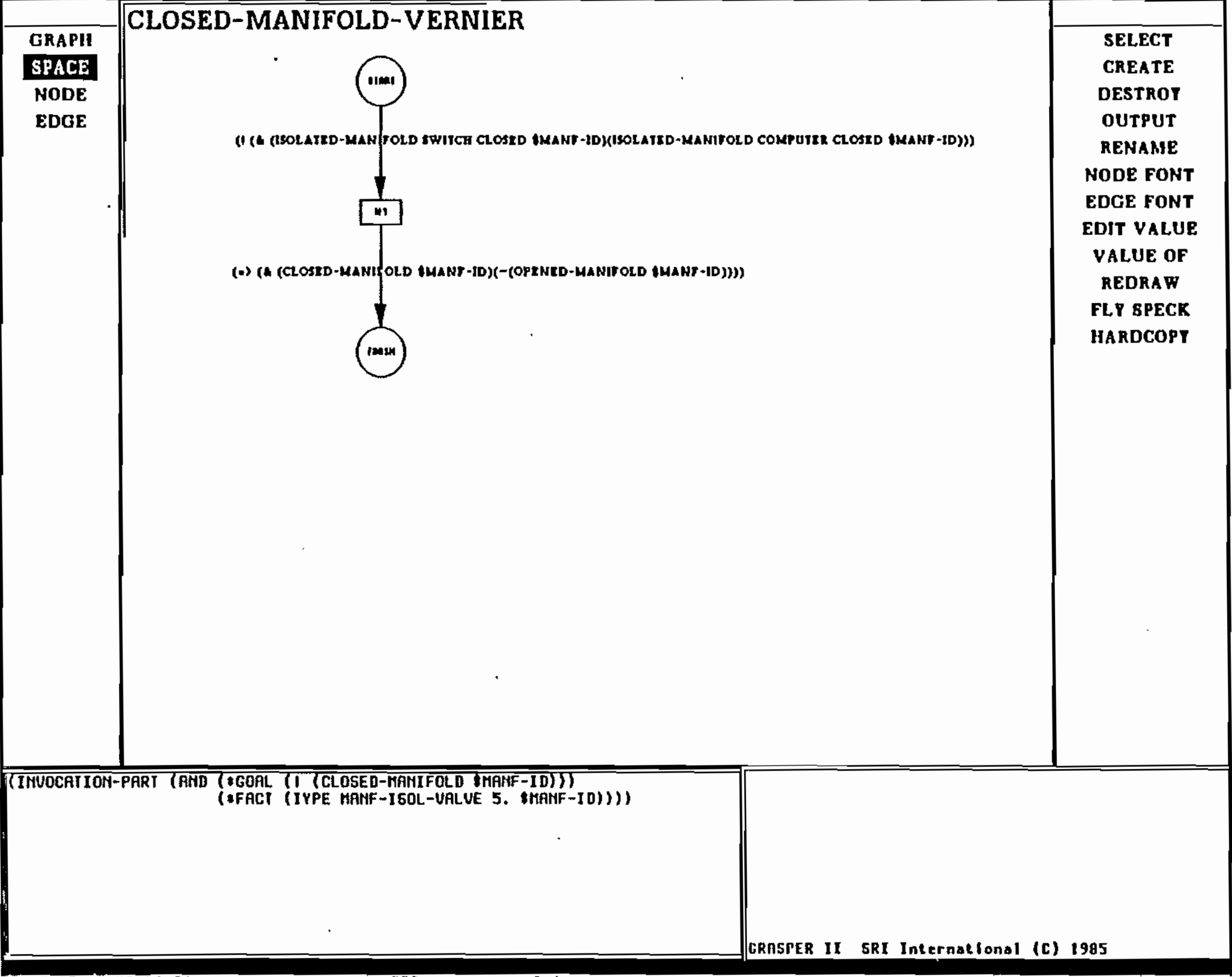
A.3 Knowledge Areas

This section contains sample processes representing some of the RCS malfunction handling procedures (see Figures 1.1, 1.2 and 1.3). Note that the syntax varies slightly from that given in the main body of the report in that the metalevel predicates `goal` and `fact` are prefixed here with a * sign. The pictures of the KAs are actual snapshots of the user interface to the system.

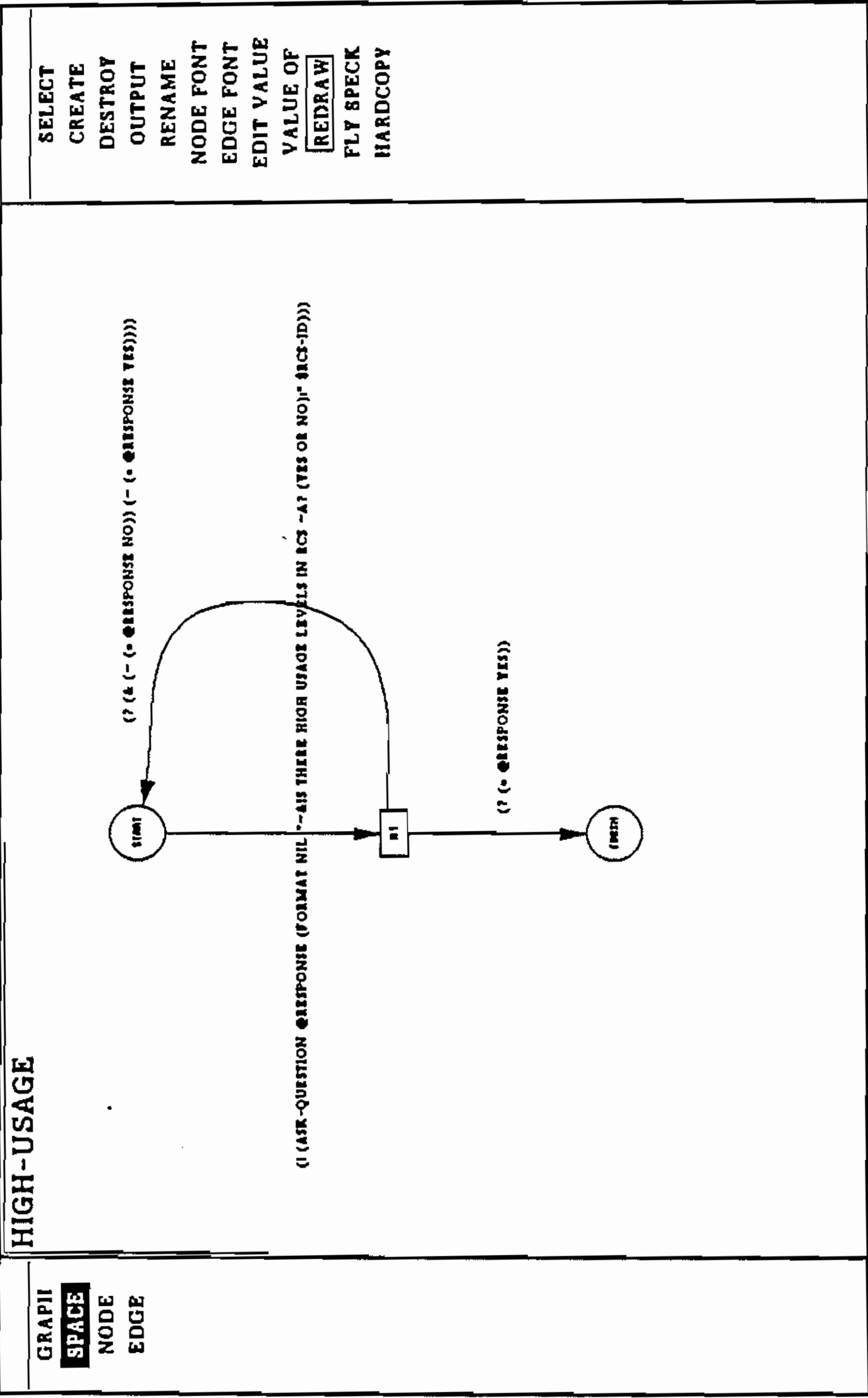

GRAPH
SPACE
NODE
EDGE
ASK-QUESTION
(! (PRS:PRINT-LIST $QUESTION))
(! (= $RESPONSE (READ)))
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(INVOCATION-PART (*GOAL (! (ASK-QUESTION $RESPONSE $QUESTION))))
GRASPER II SRI International (C) 1985
12/17/85 09:59:51 Singer
PES:
Ty!

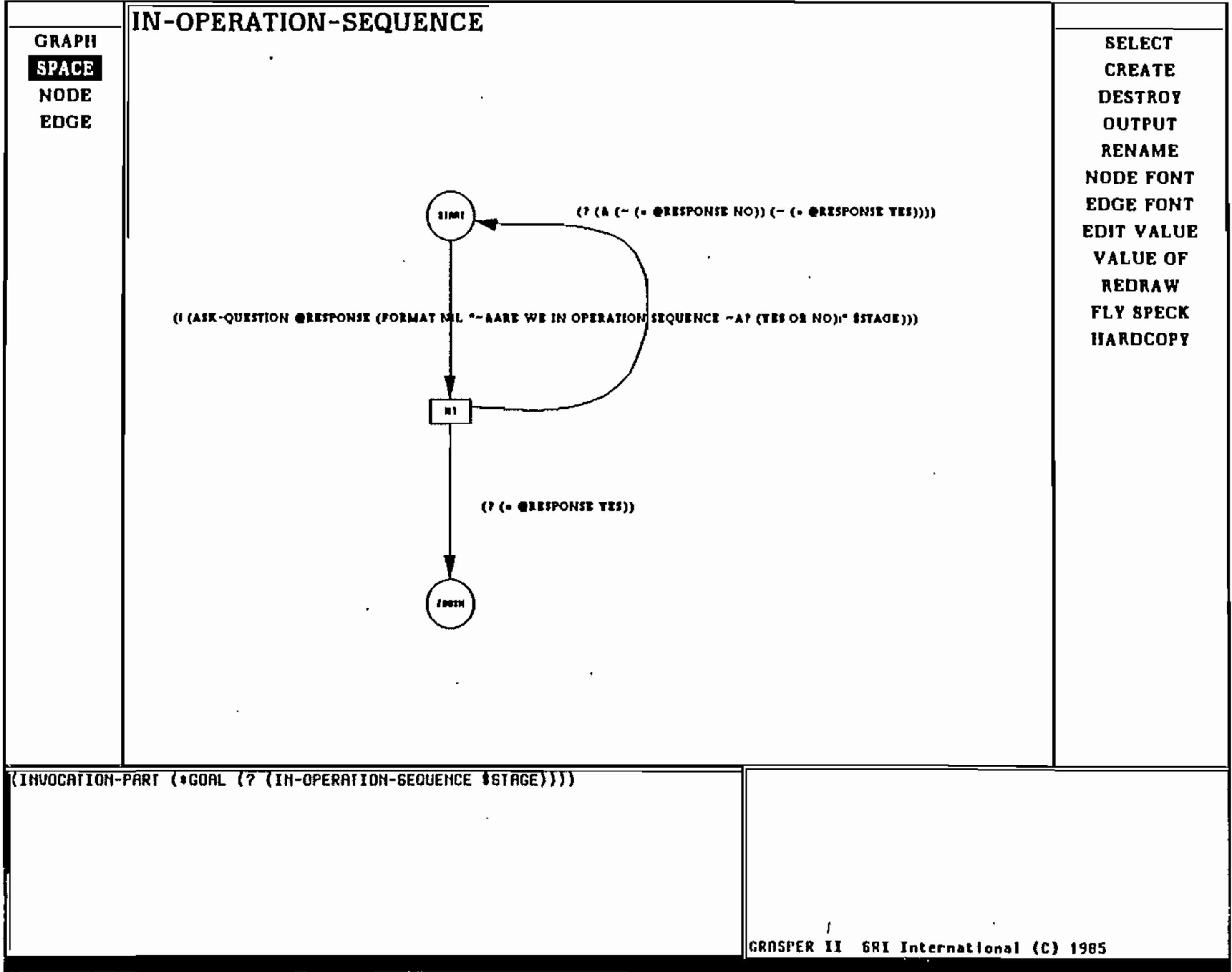


GRAPH
SPACE
NODE
EDGE
CLOSED-MANIFOLD
(! (ISOLATED-MANIFOLD SWITCH CLOSED $MANF-ID))
(=> (& (CLOSED-MANIFOLD $MANF-ID)(~(OPENED-MANIFOLD $MANF-ID))))
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(INVOCATION-PART (AND (*GOAL (! (CLOSED-MANIFOLD $MANF-ID)))
(*FACT (TYPE MANF-ISOL-VALVE $N $MANF-ID))
(*LISP-PREDICATE (NOT (= $N 5.)))))
GRASPER II SRI International (C) 1985
12/17/85 10:01:14 Singer
PES:
Ty1



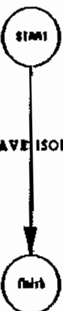
CLOSED-MANIFOLD-VERNIER
GRAPH
SPACE
NODE
EDGE
(! (& (ISOLATED-MANIFOLD SWITCH CLOSED $MANF-ID)(ISOLATED-MANIFOLD COMPUTER CLOSED $MANF-ID)))
(=> (& (CLOSED-MANIFOLD $MANF-ID)(~(OPENED-MANIFOLD $MANF-ID))))
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(INVOCATION-PART (AND (*GOAL (! (CLOSED-MANIFOLD $MANF-ID)))
(*FACT (TYPE MANF-ISOL-VALVE 5. $MANF-ID))))
GRASPER II SRI International (C) 1985
12/17/85 10:02:10 Singer
PES:
lyl


GRAPH
SPACE
NODE
EDGE
HIGH-USAGE
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(? (& (~ (~ (= @RESPONSE NO)) (~ (= @RESPONSE YES))))
(! (ASK-QUESTION @RESPONSE (FORMAT NIL "~&IS THERE HIGH USAGE LEVELS IN RCS ~A? (YES OR NO)" $RCS-ID)))
(? (= @RESPONSE YES))
(INVOCATION-PART (*GOAL (? (HIGH-USAGE $RCS-ID))))
GRASPER II SRI International (C) 1985
Redraw the current space.
12/17/85 10:10:46 Singer
PES:


IN-OPERATION-SEQUENCE
GRAPH
SPACE
NODE
EDGE
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(? (& (~ (= @RESPONSE NO)) (~ (= @RESPONSE YES))))
(! (ASK-QUESTION @RESPONSE (FORMAT NIL "~&ARE WE IN OPERATION SEQUENCE ~A? (YES OR NO):" $STAGE)))
(? (= @RESPONSE YES))
(INVOCATION-PART (*GOAL (? (IN-OPERATION-SEQUENCE $STAGE))))
GRASPER II SRI International (C) 1985
12/17/85 10:10:40 Singer
PES:
lyl

ISOLATED-MANIFOLD

GRAPH
SPACE
NODE
EDGE



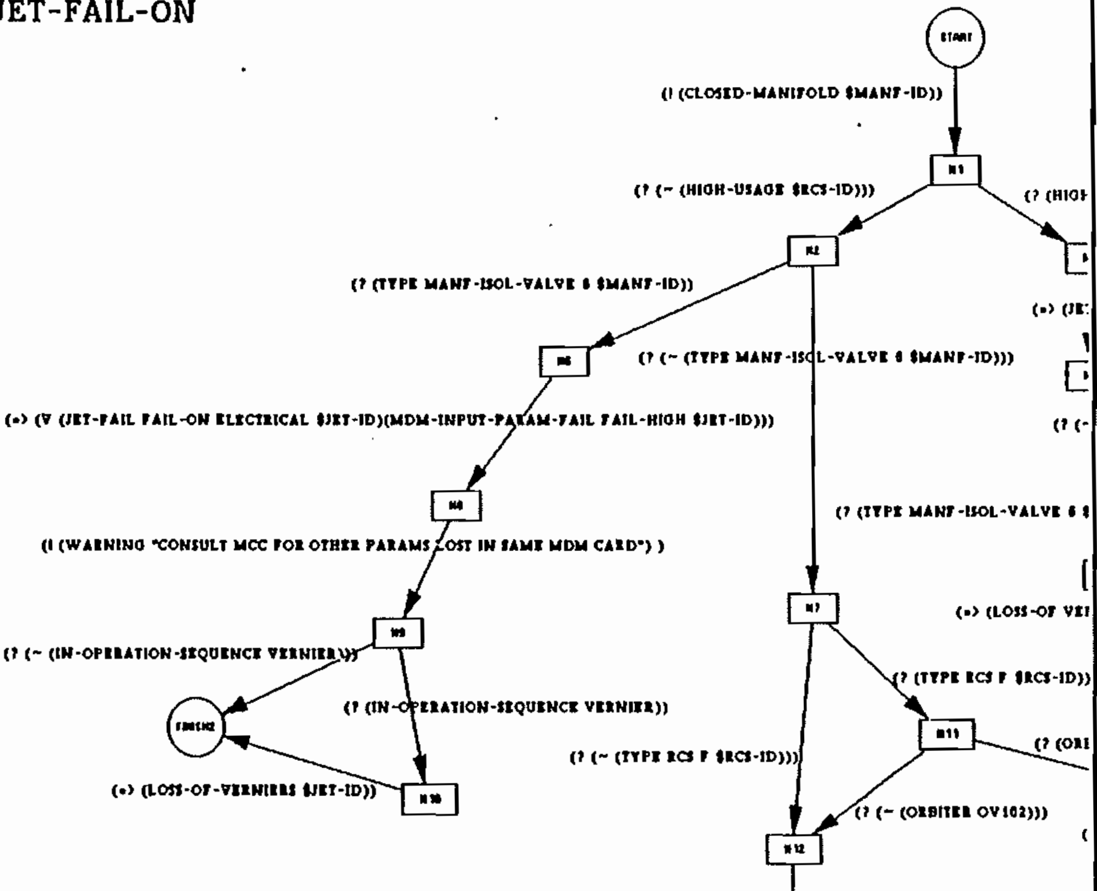
```
(! (PES:PRINT-LIST (FORMAT NIL "YOU HAVE ISOLATED MANIFOLD NUMBER ~A IN THE ~A POSITION VIA ~A." $MANF-ID $POSITION $MEANS)))
```

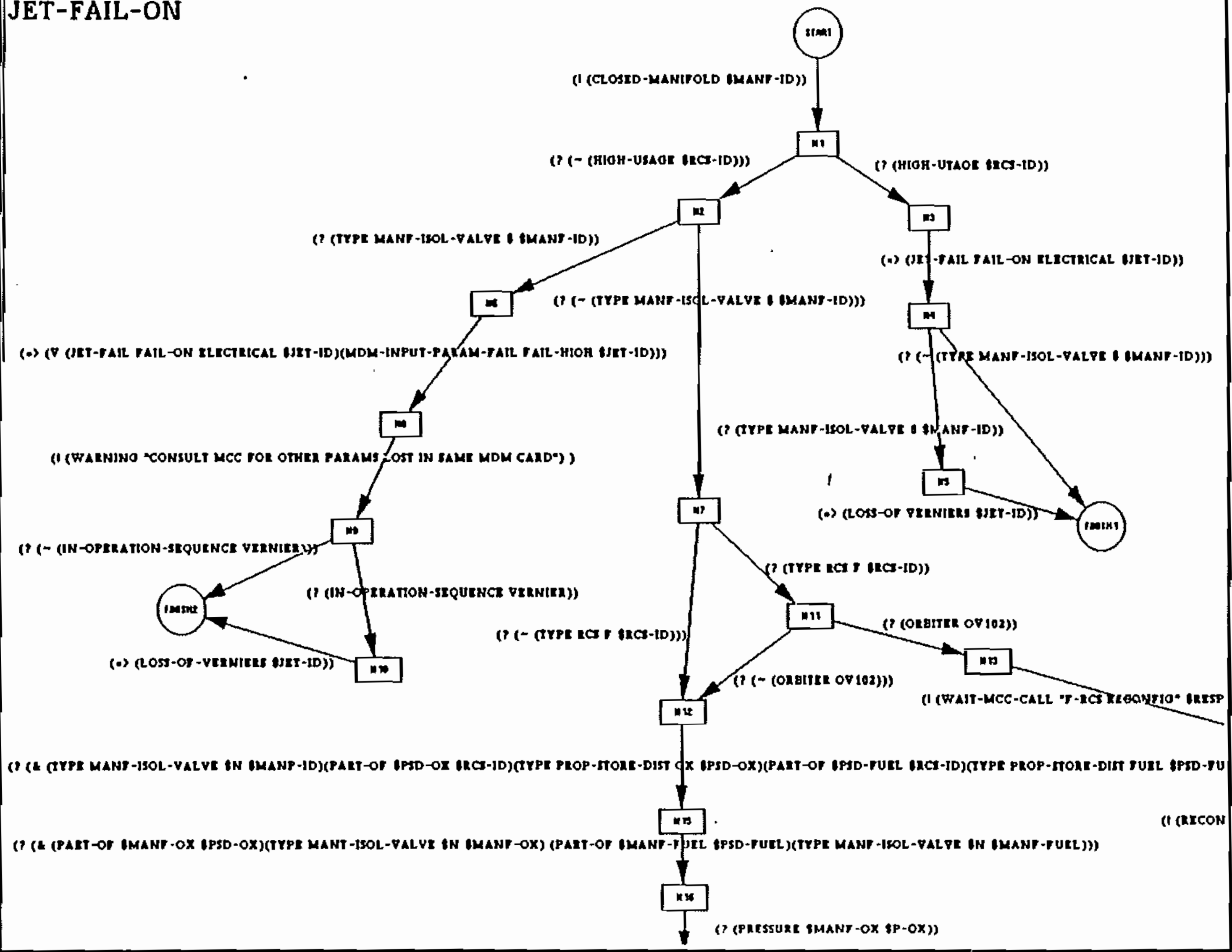
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY

```
(INVOCATION-PART (*GOAL (! (ISOLATED-MANIFOLD $MEANS $POSITION $MANF-ID))))
```

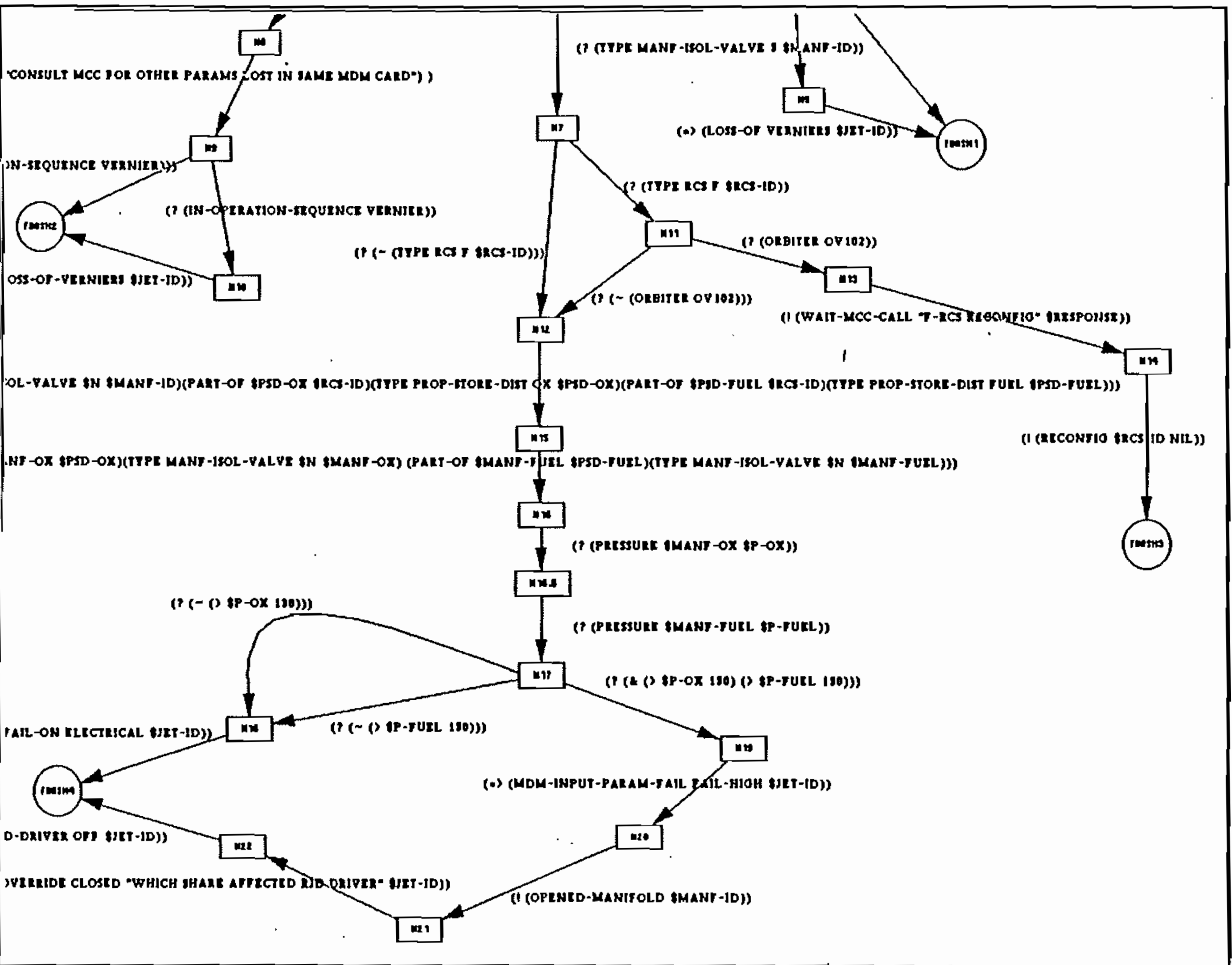
GRASPER II SRI International (C) 1985

12/17/85 10:26:16 Singer PES: lyl

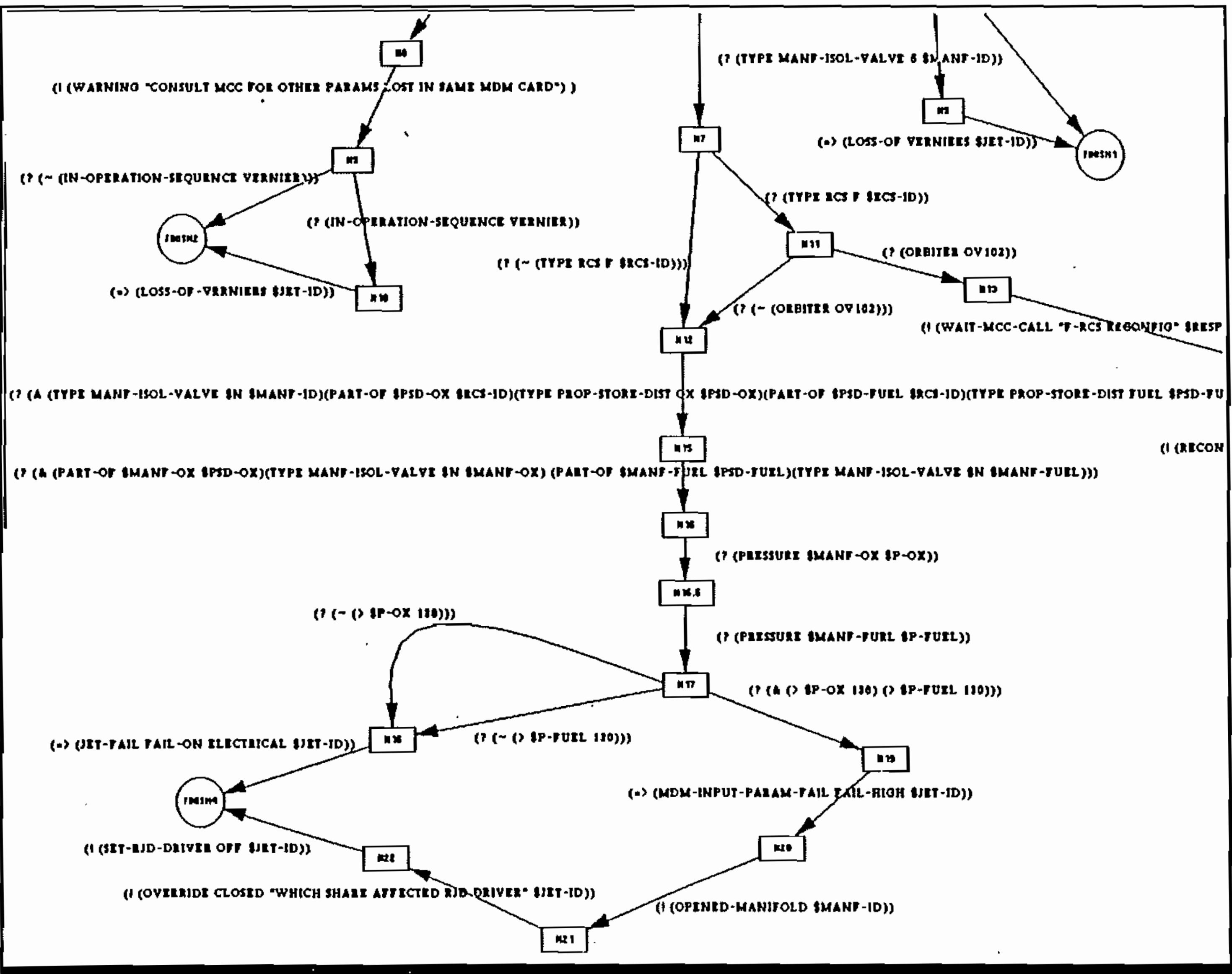

GRAPH
SPACE
NODE
EDGE
JET-FAIL-ON
(! (CLOSED-MANIFOLD $MANF-ID))
(? (~ (HIGH-USAGE $RCS-ID)))
(? (TYPE MANF-ISOL-VALVE & $MANF-ID))
(? (~ (TYPE MANF-ISOL-VALVE & $MANF-ID)))
(=> (V (JET-FAIL FAIL-ON ELECTRICAL $JET-ID)(MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID)))
(! (WARNING "CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
(? (~ (IN-OPERATION-SEQUENCE VERNIER)))
(? (IN-OPERATION-SEQUENCE VERNIER))
(=> (LOSS-OF-VERNIERS $JET-ID))
(? (TYPE RCS F $RCS-ID))
(? (~ (TYPE RCS F $RCS-ID)))
(? (~ (ORBITER OV102)))
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(INVOCATION-PART (AND (*FACT (LIGHT RCS-JET))
(*FACT (ALARM BACKUP-CW))
(*FACT (FAULT $RCS-ID RCS $JET-ID JET))
(*FACT (JETFAIL-INDICATOR ON $MANF-ID))))
GRASPER II SRI International (C) 1985
12/17/85 13:35:21 Singer
PES:
Iyt

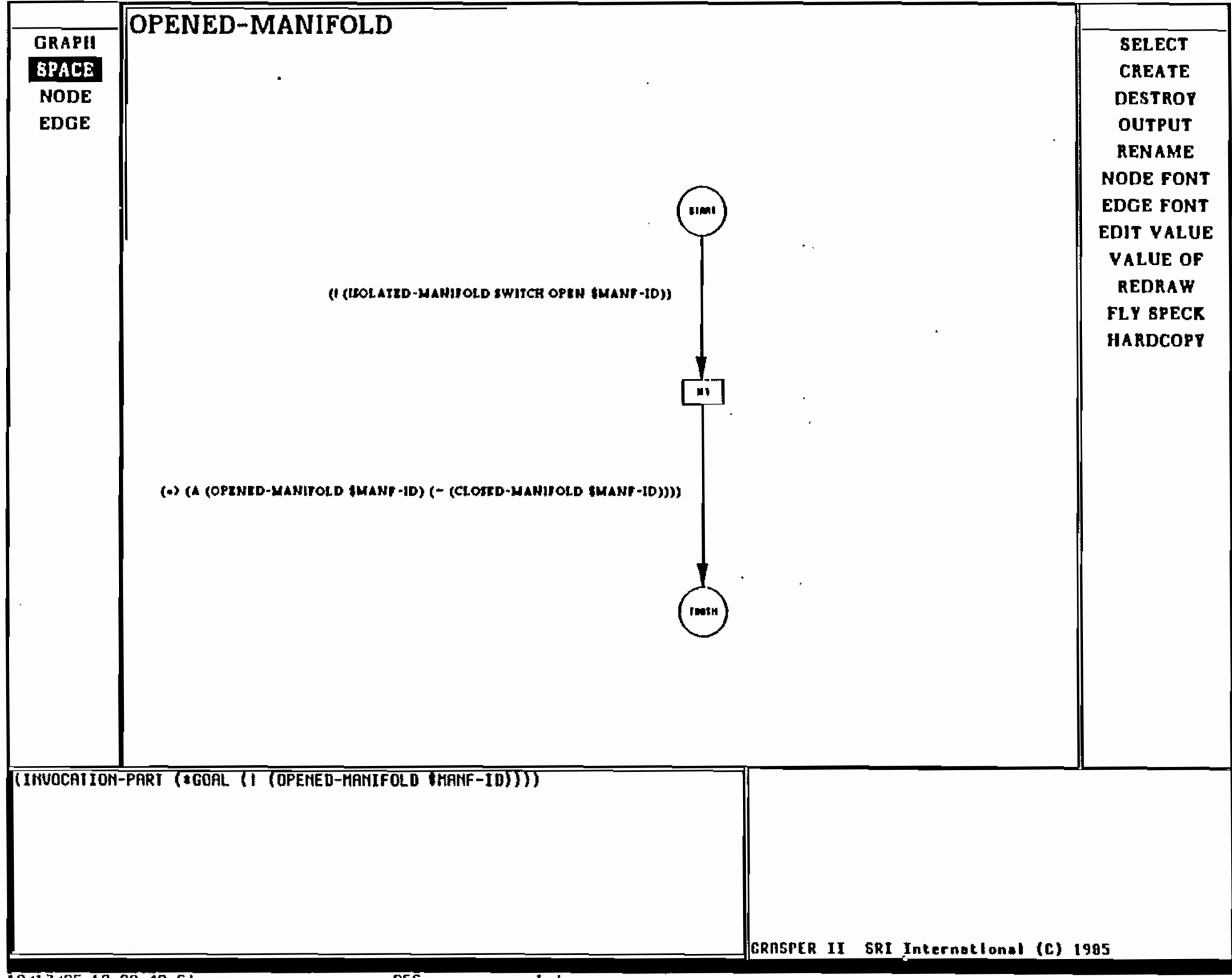

JET-FAIL-ON
START
(! (CLOSED-MANIFOLD $MANF-ID))
N1
(? (~ (HIGH-USAGE $RCS-ID)))
(? (HIGH-UTAOE $RCS-ID))
N2
N3
(? (TYPE MANF-ISOL-VALVE $ $MANF-ID))
(=) (JET-FAIL FAIL-ON ELECTRICAL $JET-ID))
(? (~ (TYPE MANF-ISOL-VALVE $ $MANF-ID)))
N4
N6
(=) (V (JET-FAIL FAIL-ON ELECTRICAL $JET-ID)(MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID)))
(? (~ (TYPE MANF-ISOL-VALVE $ $MANF-ID)))
N8
(? (TYPE MANF-ISOL-VALVE $ $MANF-ID))
(! (WARNING "CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
N5
N7
(=) (LOSS-OF VERNIERS $JET-ID))
FINISH1
N9
(? (~ (IN-OPERATION-SEQUENCE VERNIER)))
(? (TYPE RCS F $RCS-ID))
(? (IN-OPERATION-SEQUENCE VERNIER))
FINISH2
N11
(? (ORBITER OV102))
(? (~ (TYPE RCS F $RCS-ID)))
N13
(=) (LOSS-OF-VERNIERS $JET-ID))
N10
(? (~ (ORBITER OV102)))
(! (WAIT-MCC-CALL "F-RCS RECONFIG" $RESP
N12
(? (& (TYPE MANF-ISOL-VALVE $N $MANF-ID)(PART-OF $PSD-OX $RCS-ID)(TYPE PROP-STORE-DIST OX $PSD-OX)(PART-OF $PSD-FUEL $RCS-ID)(TYPE PROP-STORE-DIST FUEL $PSD-FU
N15
(! (RECON
(? (& (PART-OF $MANF-OX $PSD-OX)(TYPE MANT-ISOL-VALVE $N $MANF-OX) (PART-OF $MANF-FUEL $PSD-FUEL)(TYPE MANF-ISOL-VALVE $N $MANF-FUEL)))
N16
(? (PRESSURE $MANF-OX $P-OX))
12/17/85 12:12:04 Singer
PES:
(no window)

JET-FAIL-ON CONTINUED

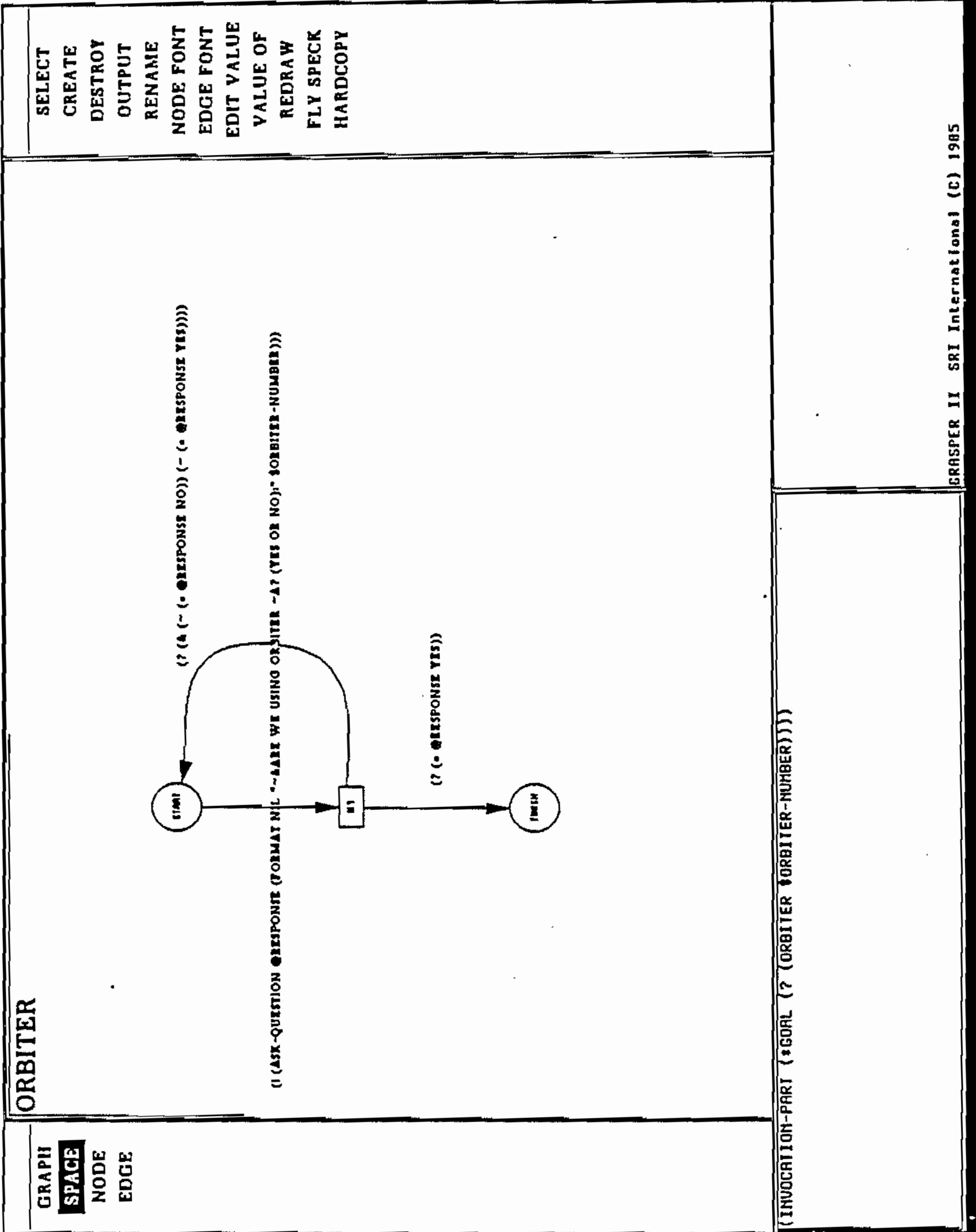


JET-FAIL-ON CONTINUED

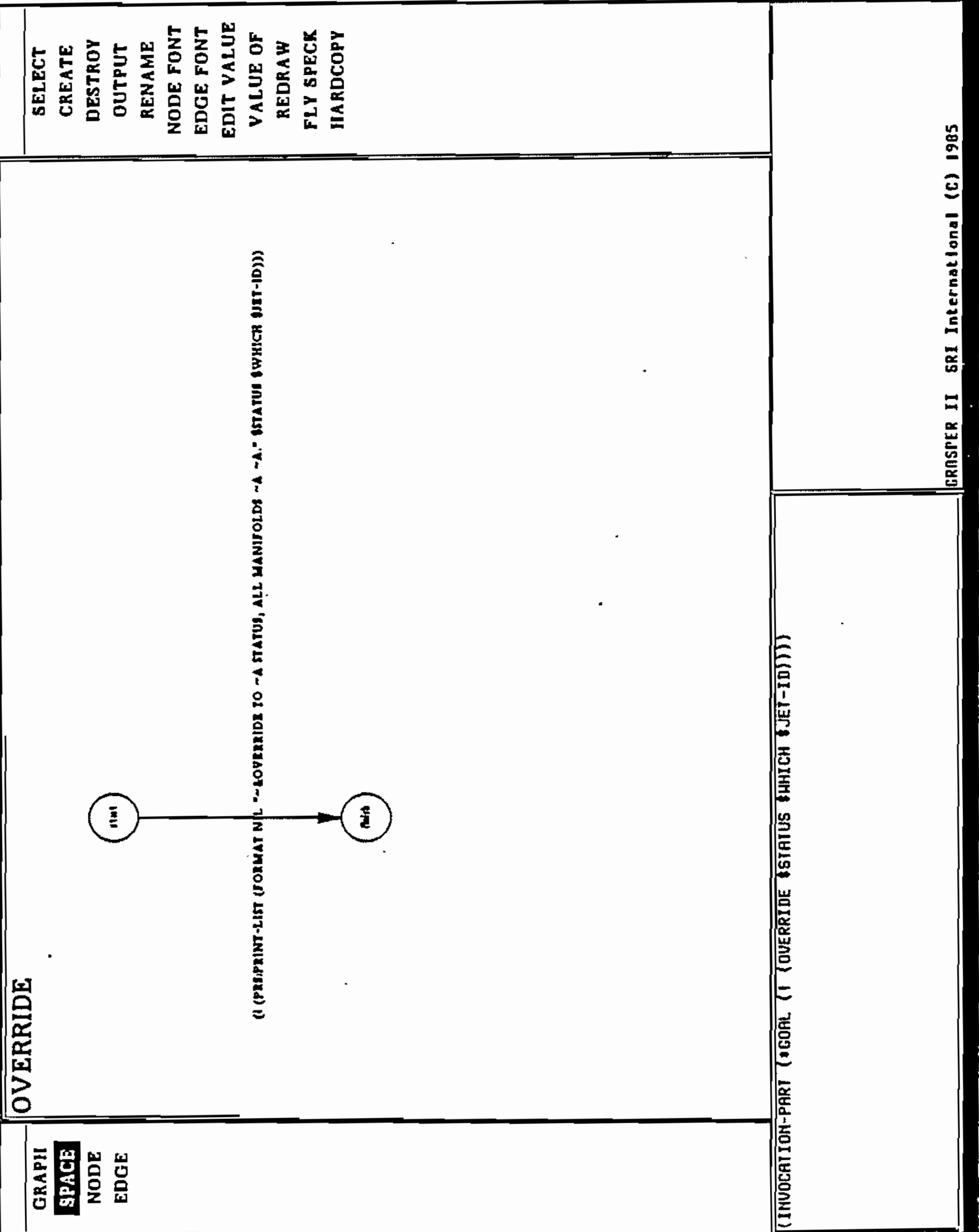


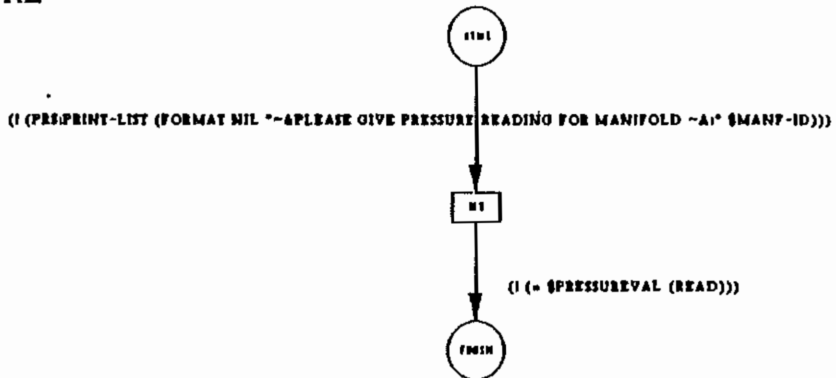


OPENED-MANIFOLD
GRAPH
SPACE
NODE
EDGE
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(! (ISOLATED-MANIFOLD SWITCH OPEN $MANF-ID))
(=) (A (OPENED-MANIFOLD $MANF-ID) (~ (CLOSED-MANIFOLD $MANF-ID))))
(INVOCATION-PART (*GOAL (! (OPENED-MANIFOLD $MANF-ID))))
GRASPER II SRI International (C) 1985
12/17/85 10:28:49 Singer
PES:
1y1

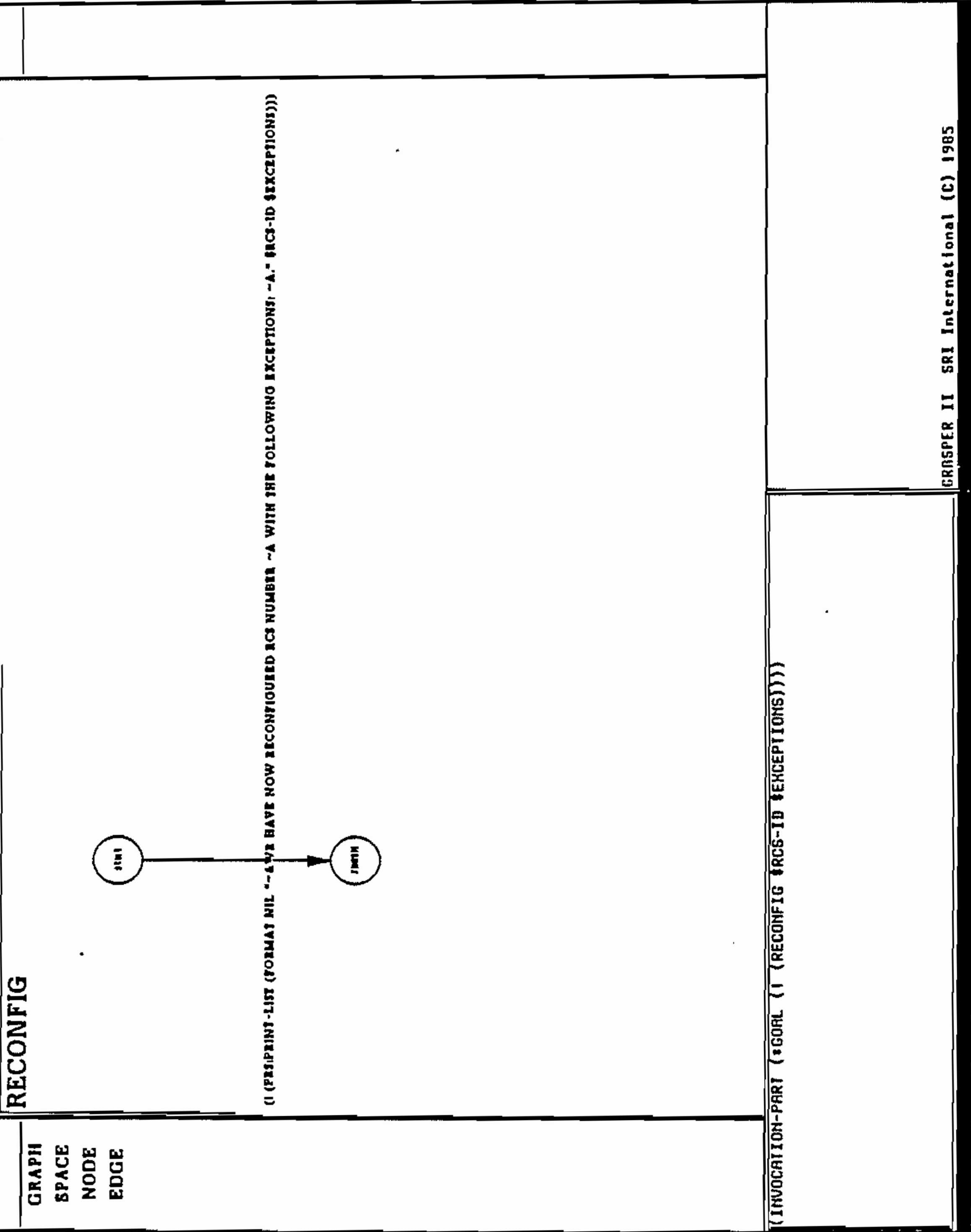


ORBITER
GRAPH
SPACE
NODE
EDGE
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(? (& (~ (= @RESPONSE NO)) (~ (= @RESPONSE YES))))
(! (ASK-QUESTION @RESPONSE (FORMAT NIL "~&ARE WE USING ORBITER ~A? (YES OR NO):" $ORBITER-NUMBER)))
(? (= @RESPONSE YES))
(INVOCATION-PART (*GOAL (? (ORBITER $ORBITER-NUMBER))))
GRASPER II SRI International (C) 1985
12/17/85 10:31:44 Singer
PES:


GRAPH
SPACE
NODE
EDGE
OVERRIDE
(I (PRS:PRINT-LIST (FORMAT NIL "~&OVERRIDE TO ~A STATUS, ALL MANIFOLDS ~A ~A." $STATUS $WHICH $JET-ID)))
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(INVOCATION-PART (*GOAL (I (OVERRIDE $STATUS $WHICH $JET-ID))))
GRASPER II SRI International (C) 1985
12/17/85 10:32:53 Singer
PES:
Tyl



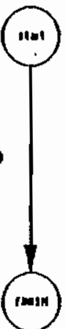
PRESSURE
GRAPH
SPACE
NODE
EDGE
(I (PRS:PRINT-LIST (FORMAT NIL "~&PLEASE GIVE PRESSURE READING FOR MANIFOLD ~A:" $MANF-ID)))
(I (= $PRESSUREVAL (READ)))
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(INVOCATION-PART (*GOAL (? (PRESSURE $MANF-ID $PRESSUREVAL))))
GRASPER II SRI International (C) 1985
12/17/85 10:35:02 Singer
PES:
Ty1



GRAPH
SPACE
NODE
EDGE
RECONFIG
(! (PRS:PRINT-LIST (FORMAT NIL "~&WE HAVE NOW RECONFIGURED RCS NUMBER ~A WITH THE FOLLOWING EXCEPTIONS: ~A." $RCS-ID $EXCEPTIONS)))
(INVOCATION-PART (*GOAL (! (RECONFIG $RCS-ID $EXCEPTIONS))))
12/17/85 10:37:59 Singer
PES:
GRASPER II SRI International (C) 1985

GRAPH
SPACE
NODE
EDGE

SET-RJD-DRIVER



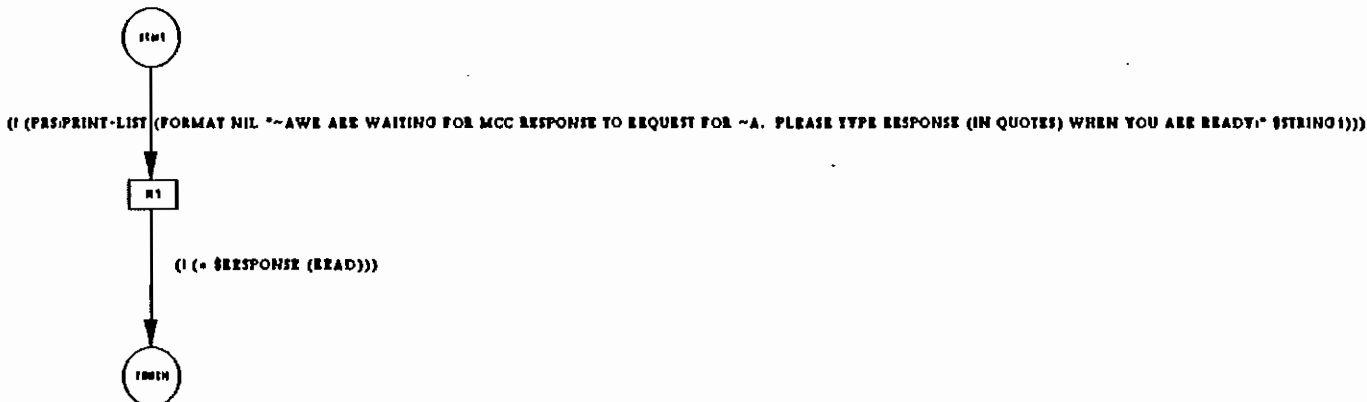
(! (PRS:PRINT-LIST (FORMAT NIL "~ATURN RJD DRIVE ~A TO ~A." $JET-ID $STATUS)))

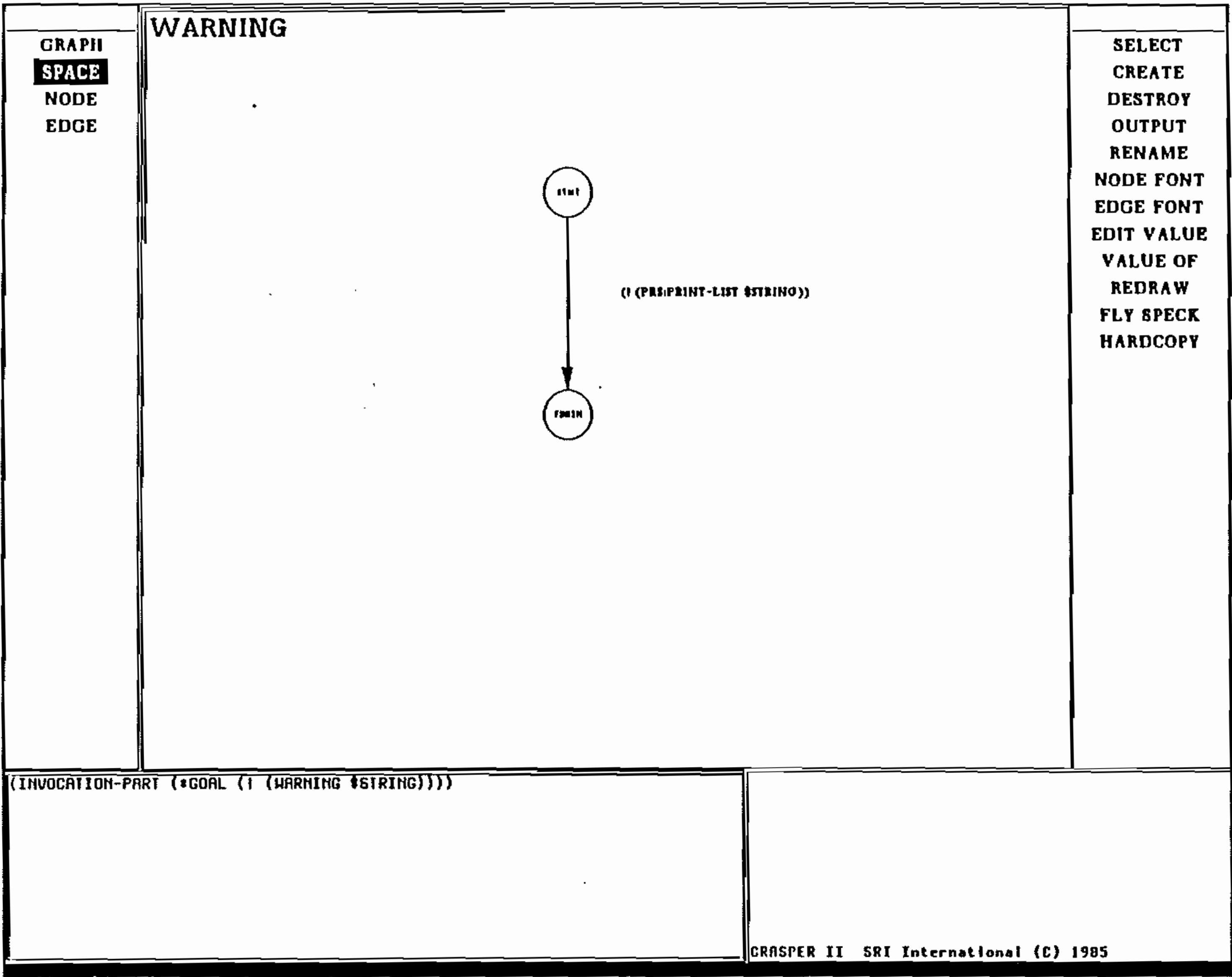
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY

(INVOCATION-PART (*GOAL (! (SET-RJD-DRIVER $STATUS $JET-ID))))

GRASPER II SRI International (C) 1985

12/17/85 15:19:58 Singer PES: lyt


WAIT-MCC-CALL
(! (PRS:PRINT-LIST (FORMAT NIL "~AWE ARE WAITING FOR MCC RESPONSE TO REQUEST FOR ~A. PLEASE TYPE RESPONSE (IN QUOTES) WHEN YOU ARE READY:" $STRING1)))
(! (= $RESPONSE (READ)))
GRAPH
SPACE
NODE
EDGE
CREATE
DESTROY
EDIT VALUE
VALUE OF
RENAME
MOVE NAME
(INVOCATION-PART (*GOAL (! (WAIT-MCC-CALL $STRING1 $RESPONSE))))
GRASPER II SRI International (C) 1985
12/17/85 10:49:05 Singer
PES:
!y!


GRAPH
SPACE
NODE
EDGE
WARNING
SELECT
CREATE
DESTROY
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
HARDCOPY
(! (PRS:PRINT-LIST $STRING))
(INVOCATION-PART (*GOAL (! (WARNING $STRING))))
GRASPER II SRI International (C) 1985
12/17/85 10:56:11 Singer
PES:

References

[1] J. S. Aikins. Prototypical knowledge for expert systems. *Artificial Intelligence*, 20:163–210, 1983.

[2] J. F. Allen. *A General Model of Action and Time*. Technical Report 97, University of Rochester, Rochester, New York, 1981.

[3] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1982.

[4] R. Davis. *Applications Of Metalevel Knowledge To The Construction, Maintenance, And Use Of Large Knowledge Bases*. Technical Report STAN-CS-76-552, HPP-76-7, Stanford University, Stanford, California, 1976.

[5] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12, 1979.

[6] W. Feurzeig, J. Frederiksen, J. White, and P. Horwitz. Designing an expert system for training automotive electrical troubleshooting. In J. J. Richardson, editor, *Proceedings of the Joint Services Workshop on Artificial Intelligence in Maintenance*, Air Force Systems Command, Human Resources Laboratory, Brooks Air Force Base, Texas, 1984.

[7] R. E. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28:904–920, 1985.

[8] R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[9] C. Forgy and J. McDermott. OPS, a domain-independent production system language. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 933–939, 1977.

[10] M. R. Genesereth and D. E. Smith. *Metalevel Architecture.* HPP Memo-81-6, Stanford University, Stanford, California, 1982.

[11] M. P. Georgeff. Procedural control in production systems. *Artificial Intelligence*, 18:175–201, 1982.

[12] M. P. Georgeff. *Reasoning about Process.* Technical Note, Artificial Intelligence Center, SRI International, Menlo Park, California, forthcoming.

[13] M. P. Georgeff. A theory of action for multiagent planning. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, Austin, Texas, 1984.

[14] M. P. Georgeff and U. Bonollo. Procedural expert systems. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, Germany, 1983.

[15] M. P. Georgeff, A. L. Lansky, and P. Bessiere. A procedural logic. In *Proceedings of the Ninth International Conference on Artificial Intelligence*, Los Angeles, California, 1985.

[16] P. J. Hayes. In defense of logic. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 559–565, Cambridge, Massachusetts, 1977.

[17] F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28:921–932, 1985.

[18] G. G. Hendrix. Modeling simultaneous actions and continuous processes. *Artificial Intelligence*, 4:145–180, 1973.

[19] R. Kowalski. *Logic for Problem Solving.* North Holland, New York, New York, 1979.

[20] D. B. Lenat. Automated theory formation in mathematics. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 833–842, Cambridge, Massachusetts, 1977.

[21] J. T. Maylin and N. Lance. An expert system for fault management and automatic shutdown avoidance in a regenerative life support subsystem. In *Proceedings of*

the Instrument Society of America First Annual Workshop on Robotics and Expert Systems, Houston, Texas, 1985.

[22] D. McDermott. *A Temporal Logic for Reasoning about Plans and Processes.* Computer Science Research Report 196, Yale University, New Haven, Connecticut, 1981.

[23] N. J. Nilsson. *Problem Solving Methods in Artificial Intelligence.* McGraw Hill, New York, New York, 1971.

[24] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[25] J. J. Richardson, editor. *Artificial Intelligence in Maintenance: Proceedings of the Joint Services Workshop.* Air Force Systems Command, Air Force Human Resources Laboratory, Brooks Air Force Base, Texas, 1984.

[26] S. J. Rosenschein. Plan synthesis: a logical perspective. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 331–337, Vancouver, British Columbia, 1981.

[27] E. D. Sacerdoti. *A Structure for Plans and Behaviour.* Elsevier, North Holland, New York.

[28] M. Stefik. An examination of a frame-structured representation system. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 845–852, 1979.

[29] M. Stefik. Planning with constraints. *Artificial Intelligence*, 16:111–140, 1981.

[30] A. Tate. Goalstructure — capturing the intent of plans. In *Proceedings of the Sixth European Conference on Artificial Intelligence*, pages 273–276, 1984.

[31] S. Vere. *Planning In Time: Windows And Durations For Activities And Goals.* Jet Propulsion Laboratory, Pasadena, California.

[32] van Melle, W. *A Domain-Independent System that Aids in Constructing Knowledge-Based Consultation Programs.* Technical Report STAN-CS-80-814 and HPP Memo-80-1, Computer Science Department, Stanford University, Stanford, California, 1980.

[33] D. E. Wilkins. Domain independent planning: representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.