DEVELOPMENT OF AN EXPERT SYSTEM FOR REPRESENTING PROCEDURAL KNOWLEDGE

Final Report
Covering Period 15 May 1984 to 18 December 1985

December 18, 1985

By: Michael P. Georgeff, Program Director
Representation and Reasoning Group

Amy L. Lansky, Computer Scientist
Representation and Reasoning Group

Artificial Intelligence Center
Computer Science and Technology Division

Prepared for:

National Aeronautics and Space Administration
Ames Research Center
Space Technology Branch
Moffett Field, California 94035

Attention: Dr. Henry Lum

SRI Project 7268

Approved for Public Release: Distribution Unlimited.

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

Preparation of this report was supported by the National Aeronautics and Space Administration under Contract NSA2-11864.

Approved:

Stanley J. Rosenschein, Director
Artificial Intelligence Center

Donald L. Nielson, Vice President and Director
Computer Science and Technology Division

Contents

Chapter 1

Description of the Problem

1.1 Introduction

A high level of automation is of paramount importance in most space operations. It is critical for unmanned missions and greatly increases the effectiveness of manned missions. However, although many functions can be automated by using advanced engineering techniques, others require complex reasoning, sensing, and manipulatory capabilities that go beyond this technology.

Automation of fault diagnosis and malfunction handling is a case in point. The military have long been interested in this problem, and have developed automatic test equipment to aid in the maintenance of complex military hardware. These systems are all based on conventional software and engineering techniques. However, the effectiveness of such test equipment is severely limited [42]. The equipment is inflexible and unresponsive to the skill level of the technicians using it. The diagnostic procedures cannot be matched to the exigencies of the current situation nor can they cope with reconfiguration or modification of the items under test. The diagnosis cannot be guided by useful advice from technicians and, when a fault cannot be isolated, no explanation is given as to the cause of failure. Because these systems perform a prescribed sequence of tests, 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 test software is substantial and time to maturation is excessive.

Many space operations require even more complex reasoning abilities than those needed for the maintenance of military equipment. 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 extraterrestrial rovers.

Conventional automation techniques are unlikely to be effective in most of these applications. The systems need to be very flexible and responsive to changes in operating conditions. They must be able to interact with mission controllers, mission specialists, and astronauts in a way that uses their skills to maximum effect. And, because of the enormous cost and long duration of many space projects, these systems must be verifiable and capable of evolutionary change.

Significant advances in artificial intelligence (AI) have recently led to the development of powerful and flexible reasoning systems, known as *expert* or *knowledge-based systems*. These systems utilize the knowledge of experts to reason about problems in the domain of interest in much the same way as is done by the experts themselves. They have the ability to explain their reasoning to the expert or user, and can incrementally acquire new knowledge. They are flexible, responsive to changes of situation, and can modify their behavior under varying conditions.

The application of expert systems to space 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. However, most of these systems are not well suited to problem domains in which much of an expert's knowledge is procedural - that is, where the operation depends on performing various *sequences* of tests and actions. Yet this type of knowledge is crucial in each of the aforementioned applications. Furthermore, since all these applications involve operations that change the state of the world, a knowledge representation scheme is needed that adequately models the effects of action and change. No conventional expert system yet provides such a representation scheme.

In this report we describe a scheme for *explicitly* representing and reasoning about procedural knowledge while retaining the benefits of traditional expert systems. 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*** [18].

It is important to point out that this research confronts some fundamental problems that arise in domains in which knowledge of the world is influenced by events and actions. In particular, the use of procedural knowledge is critical in areas where reasoning over time is concerned, and where choice of action is strongly dependent on external events.

This report describes work done during the first phase of this project. The work so far has been directed at delineating the basic design of the system, experimenting with potentially useful applications within the context of NASA's space operations, and identifying some of the outstanding problems requiring further research.

We have made substantial progress in these areas. We have devised a powerful and theoretically sound scheme for representing and reasoning about procedural knowledge. 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. 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 [38] and consistency maintenance [8]. We should also investigate concurrency, and extend the model to deal with it. Some work in this direction is described by us elsewhere [19].

1.2 Fault Isolation and Diagnosis

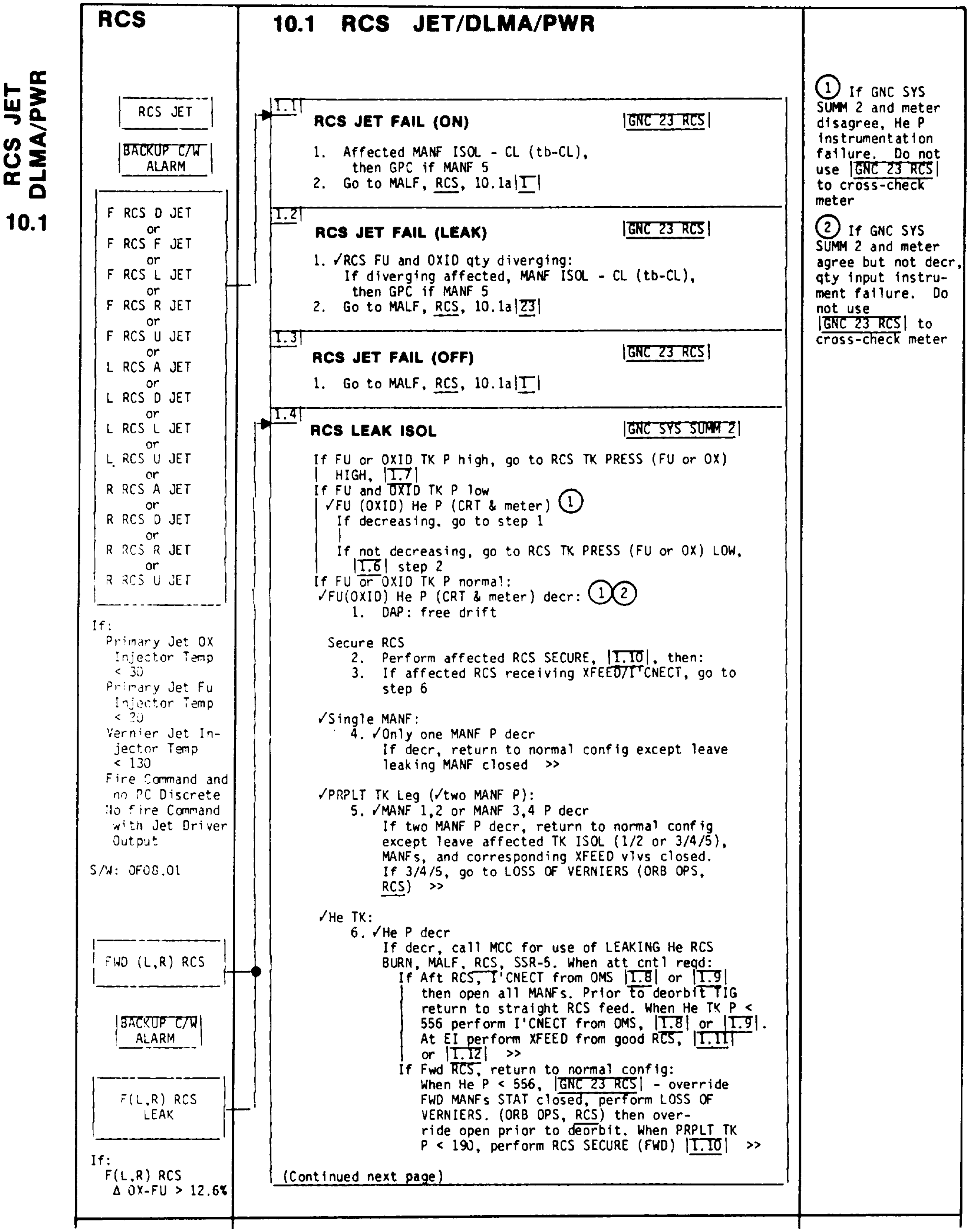
As mentioned in the introduction, there are many space operations that require complex reasoning capabilities, including command and control, monitoring and control of experiments, management of various subsystems, and the handling of system malfunctions. The last is of particular importance. On the 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 system depend entirely on the context of the particular maintenance procedure being executed: if the desired response is obtained,



RCS JET DLMA/PWR

10.1

RCS

|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%

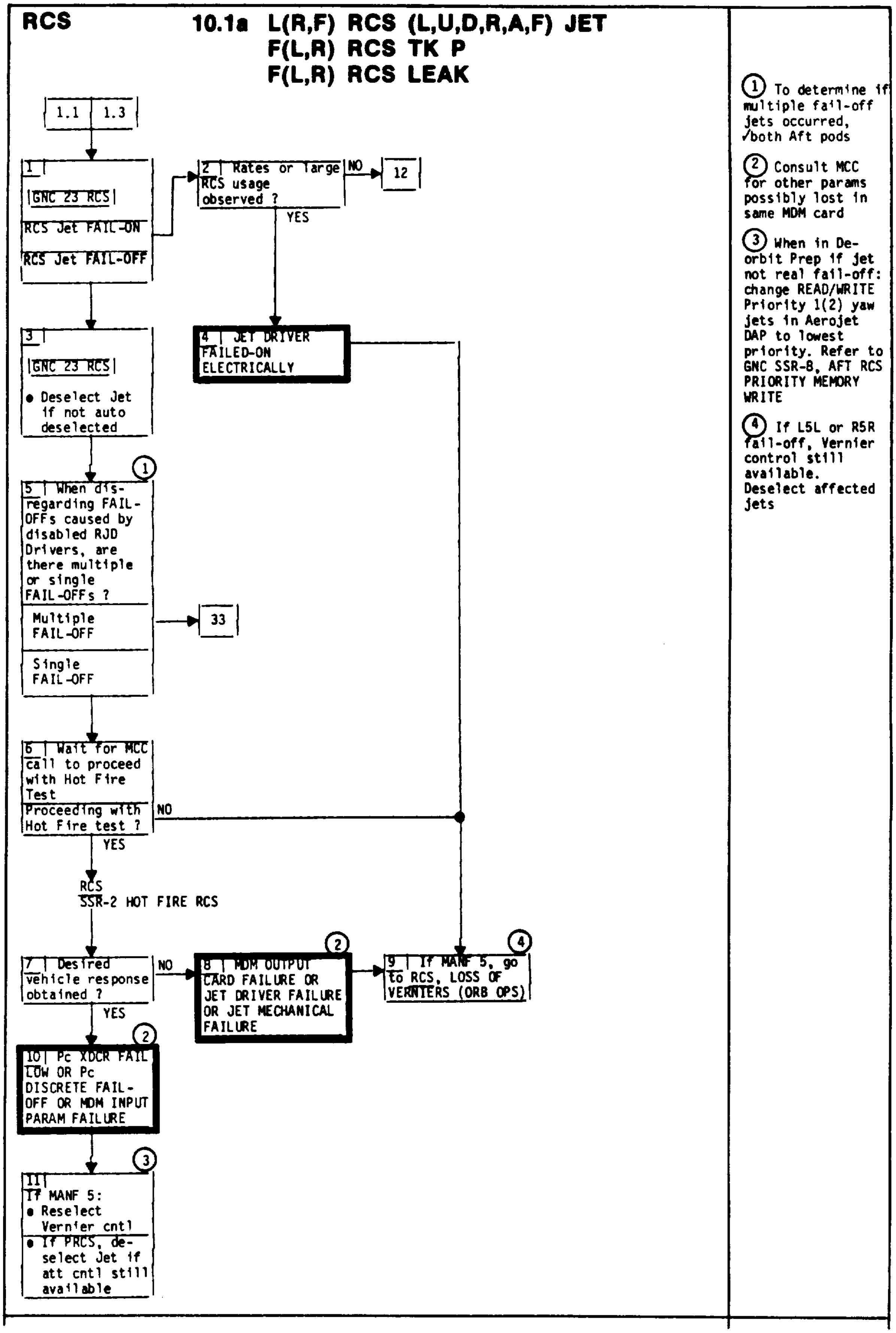
10.1 RCS JET/DLMA/PWR

```
|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 over-
              ride 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



RCS 10.1a L(R,F) RCS (L,U,D,R,A,F) JET
F(L,R) RCS TK P
F(L,R) RCS LEAK

1.1 | 1.3

1 | GNC 23 RCS | RCS Jet FAIL-ON | RCS Jet FAIL-OFF

2 | Rates or Targe RCS usage observed ? NO → 12; YES → 4

4 | JET DRIVER FAILED-ON ELECTRICALLY

3 | GNC 23 RCS | ● Deselect Jet if not auto deselected

5 | When disregarding FAIL-OFFs caused by disabled RJD Drivers, are there multiple or single FAIL-OFFs ? ① Multiple FAIL-OFF → 33 | Single FAIL-OFF

6 | Wait for MCC call to proceed with Hot Fire Test | Proceeding with Hot Fire test ? NO → 9; YES

RCS SSR-2 HOT FIRE RCS

7 | Desired vehicle response obtained ? NO → 8; YES → 10

8 | MDM OUTPUT CARD FAILURE OR JET DRIVER FAILURE OR JET MECHANICAL FAILURE ②

9 | If MANF 5, go to RCS, LOSS OF VERNIERS (ORB OPS) ④

10 | Pc XDCR FAIL LOW OR Pc DISCRETE FAIL-OFF OR MDM INPUT PARAM FAILURE ②

11 | If MANF 5: ● Reselect Vernier cntl | ● If PRCS, deselect Jet if att cntl still available ③

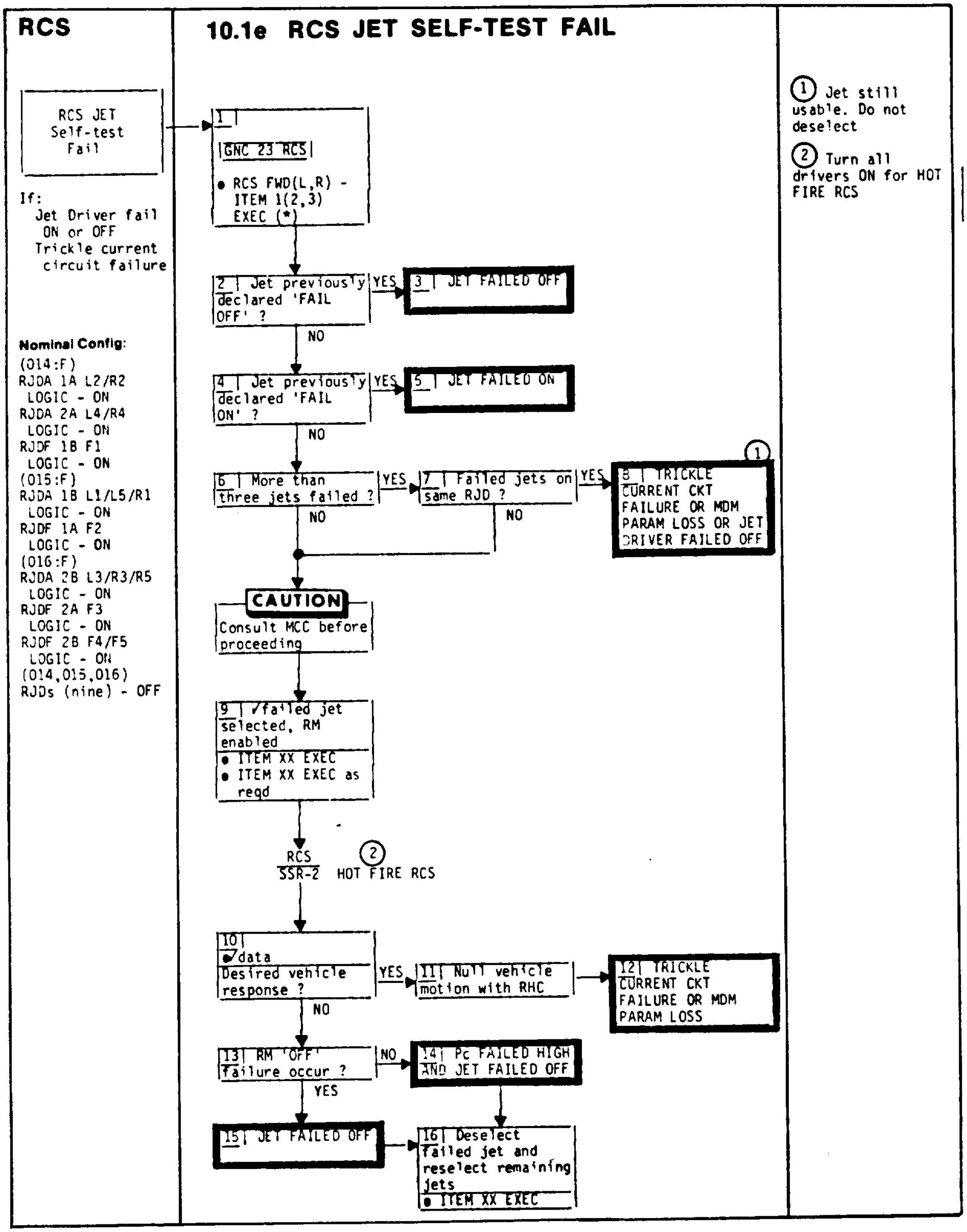
① To determine if multiple fail-off jets occurred, ✓both Aft pods

② Consult MCC for other params possibly lost in same MDM card

③ When in De-orbit Prep if jet not real fail-off: change READ/WRITE Priority 1(2) yaw jets in Aerojet DAP to lowest priority. Refer to GNC SSR-8, AFT RCS PRIORITY MEMORY WRITE

④ If L5L or R5R fail-off, Vernier control still available. Deselect affected jets

Figure 1.2: RCS Malfunction Procedure 10.1a



RCS

10.1e RCS JET SELF-TEST FAIL

RCS JET
Self-test
Fail

If:
Jet Driver fail
ON or OFF
Trickle current
circuit failure

Nominal Config:
(O14:F)
RJDA 1A L2/R2
LOGIC - ON
RJDA 2A L4/R4
LOGIC - ON
RJDF 1B F1
LOGIC - ON
(O15:F)
RJDA 1B L1/L5/R1
LOGIC - ON
RJDF 1A F2
LOGIC - ON
(O16:F)
RJDA 2B L3/R3/R5
LOGIC - ON
RJDF 2A F3
LOGIC - ON
RJDF 2B F4/F5
LOGIC - ON
(O14,O15,O16)
RJDs (nine) - OFF

(1) Jet still usable. Do not deselect

(2) Turn all drivers ON for HOT FIRE RCS

1
GNC 23 RCS
● RCS FWD(L,R) - ITEM 1(2,3) EXEC (*)
2 Jet previously declared 'FAIL OFF' ?
YES
3 JET FAILED OFF
NO
4 Jet previously declared 'FAIL ON' ?
YES
5 JET FAILED ON
NO
6 More than three jets failed ?
YES
7 Failed jets on same RJD ?
YES
1
8 TRICKLE CURRENT CKT FAILURE OR MDM PARAM LOSS OR JET DRIVER FAILED OFF
NO
NO
CAUTION
Consult MCC before proceeding
9 ✓failed jet selected, RM enabled
● ITEM XX EXEC
● ITEM XX EXEC as reqd
RCS
SSR-2
2
HOT FIRE RCS
10
●✓data
Desired vehicle response ?
YES
11 Null vehicle motion with RHC
12 TRICKLE CURRENT CKT FAILURE OR MDM PARAM LOSS
NO
13 RM 'OFF' failure occur ?
NO
14 Pc FAILED HIGH AND JET FAILED OFF
YES
15 JET FAILED OFF
16 Deselect failed jet and reselect remaining jets
● ITEM XX EXEC

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 system 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 [10]: 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 [6], is very important for effective practical reasoning [15].

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 basis 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)[1,13,24,47]. 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 [34]. It is interesting to note how they are forced to use procedures for the diagnosis, and how the representation of these procedures manifests most of 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, rather 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 [51], OPS [14], 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 necessary for reasoning about composite goals, is critical for verification, substantially eases the construction process, allows for the incremental addition of knowledge, and provides for more meaningful explanations of system reasoning. A more formal treatment of the semantics of the representation is given in Appendix A, and the basis for the formal model in Appendix B (see also [20,21]).

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

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

EFFECTS: (! (OPERABLE $MODULE))

BODY: .

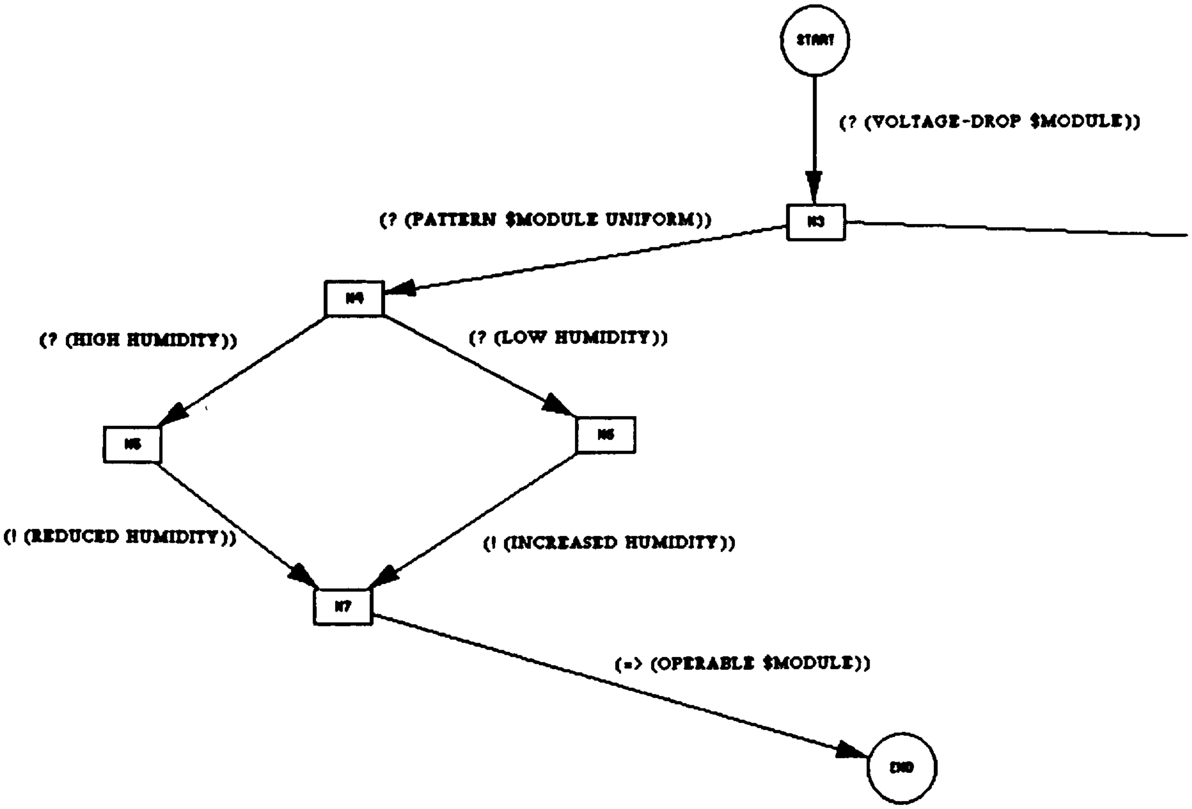


Figure 3.1: Portion of KA for Fuel Cell Malfunction

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.

The main point to note about this procedure 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

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

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

BODY: .

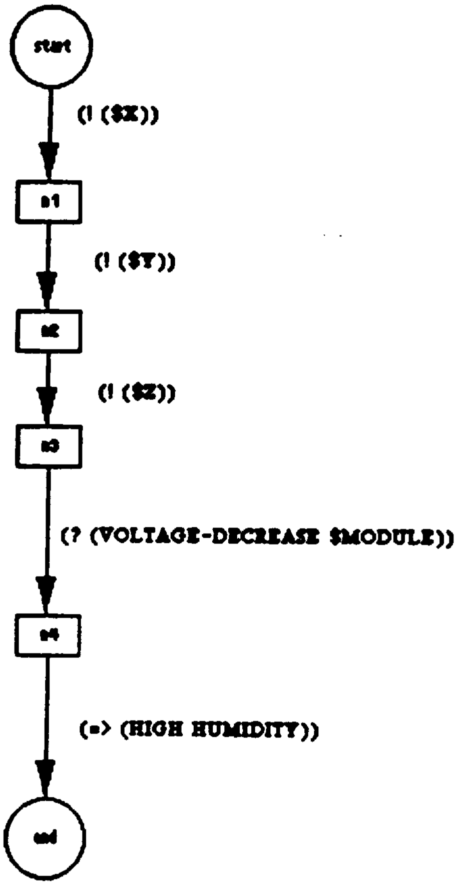


Figure 3.2: KA for Testing Humidity

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 particulary important because we often need the system to react in different ways to observed conditions, depending on the current operating mode.

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*.

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. Goals are represented by descriptions of the *behaviors* that are to be achieved.

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,

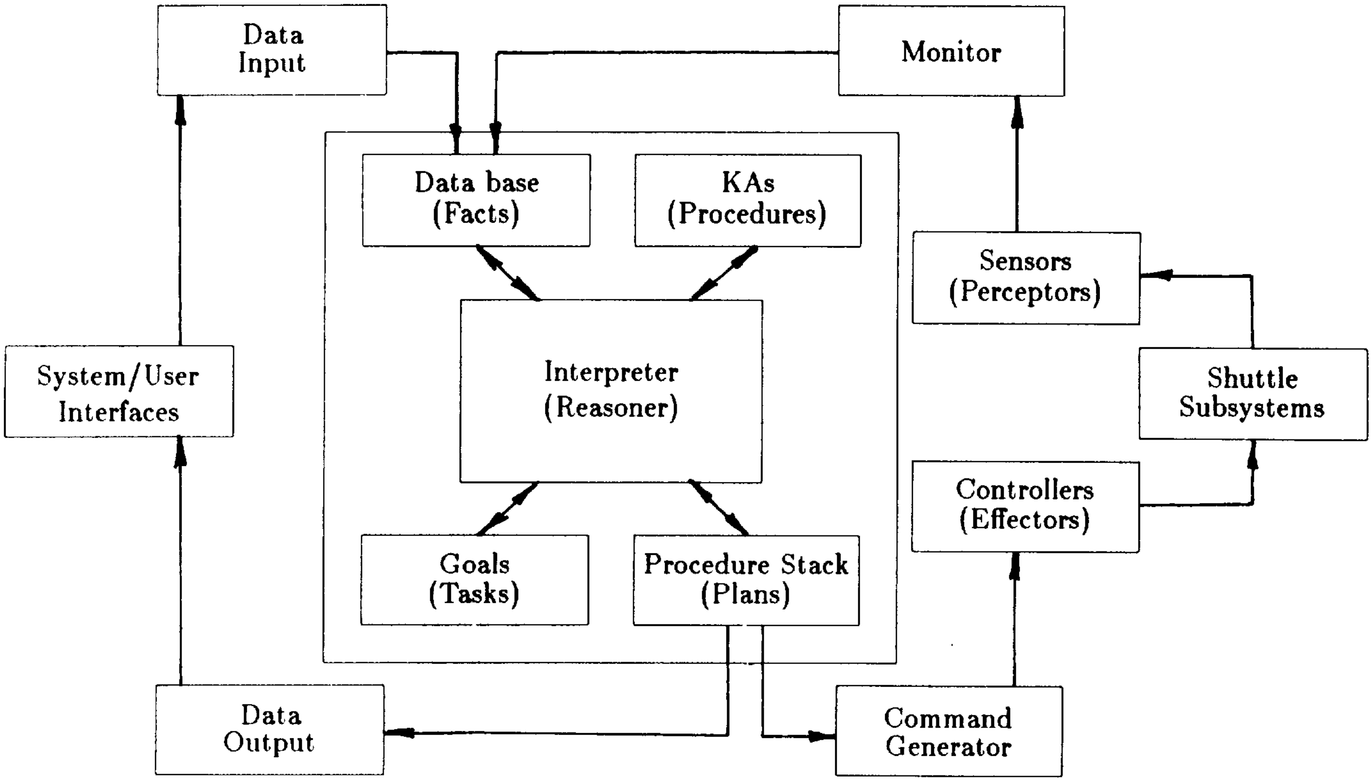


Figure 3.3: System Structure

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 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 below. 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.

```
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
```

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. 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.

Finally, it is important to note that a procedural expert system is not limited to representing procedural knowledge, but can also represent static knowledge about the problem domain. For example, the data base can be expected to include many facts about the domain, such as the fact that a particular fuel tank is part of a particular RCS, or that the current pressure in the tank is 130 psi.

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 - $\neg$ (negation), $\wedge$ (conjunction), and $\vee$ (disjunction) - are allowed and have their usual meaning. We use prefix notation (as in LISP) and both $\wedge$ and $\vee$ 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.

We intend that future extensions to the system will include a *structure editor*. This editor would enable the graphical representation of an application system (for instance, a schematic of the subsystem under test), 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 struture of the application system into predicate form - a task that we had to perform for the space shuttle application described in the next section.

A graphical representation of system schematics could also be used as a vehicle for run-time explanation and interaction with a PES user. For example, manifolds currently being opened, closed, or tested could be highlighted and their current characteristics displayed. There is even potential for user alteration of a system's characteristics through the manipulation of its graphical representation - the graphical interface would then play an active manipulative role, rather than a passive role reflecting current status.

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

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

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 transitted, 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 (! (= @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.

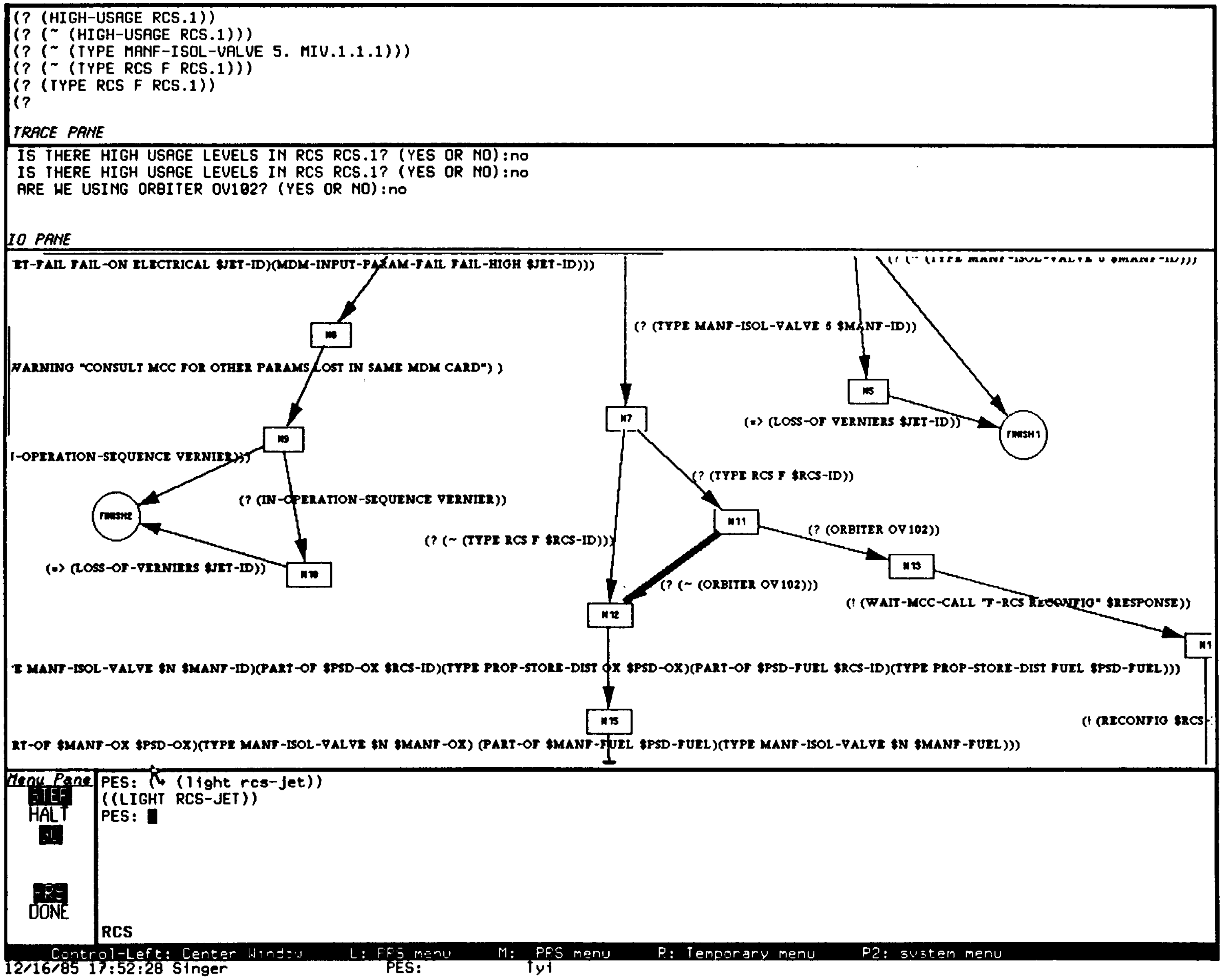
The second pane is a user run-time interaction window - the input/output pane. Here, messages that are "sent to" the user by a KA (for example, questions that ask the user for particular kinds of information) are printed out, and user responses are typed in and sent back to the KA. For instance, in the RCS system 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 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.



```
(? (HIGH-USAGE RCS.1))
(? (~ (HIGH-USAGE RCS.1)))
(? (~ (TYPE MANF-ISOL-VALVE 5. MIV.1.1.1)))
(? (~ (TYPE RCS F RCS.1)))
(? (TYPE RCS F RCS.1))
(?
```

```
IS THERE HIGH USAGE LEVELS IN RCS RCS.1? (YES OR NO):no
IS THERE HIGH USAGE LEVELS IN RCS RCS.1? (YES OR NO):no
ARE WE USING ORBITER OV102? (YES OR NO):no
```

```
PES: (light rcs-jet))
((LIGHT RCS-JET))
PES: █
```

TRACE PANE
IO PANE
RT-FAIL FAIL-ON ELECTRICAL $JET-ID)(MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID)))
N8
WARNING "CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
N9
I-OPERATION-SEQUENCE VERNIER)))
FINISH2
(? (IN-OPERATION-SEQUENCE VERNIER))
(=> (LOSS-OF-VERNIERS $JET-ID))
N10
(? (TYPE MANF-ISOL-VALVE 5 $MANF-ID))
N7
N5
(=> (LOSS-OF VERNIERS $JET-ID))
FINISH1
(? (TYPE RCS F $RCS-ID))
N11
(? (ORBITER OV102))
(? (~ (TYPE RCS F $RCS-ID)))
N13
(? (~ (ORBITER OV102)))
(! (WAIT-MCC-CALL "F-RCS RECONFIG" $RESPONSE))
N12
N1
E 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-FUEL)))
N15
(! (RECONFIG $RCS
RT-OF $MANF-OX $PSD-OX)(TYPE MANF-ISOL-VALVE $N $MANF-OX) (PART-OF $MANF-FUEL $PSD-FUEL)(TYPE MANF-ISOL-VALVE $N $MANF-FUEL)))
Menu Pane
STEP
HALT
GO
PES
DONE
RCS
Control-Left: Center Window L: PES menu M: PES menu R: Temporary menu P2: system menu
12/16/85 17:52:28 Singer PES: Tyi

Figure 4.1: PES User Interface

- **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.

- Setting up an additional window for the structure editor described earlier. This would be a context for visualizing changes to the physical characteristics of the domain.
- 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.

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

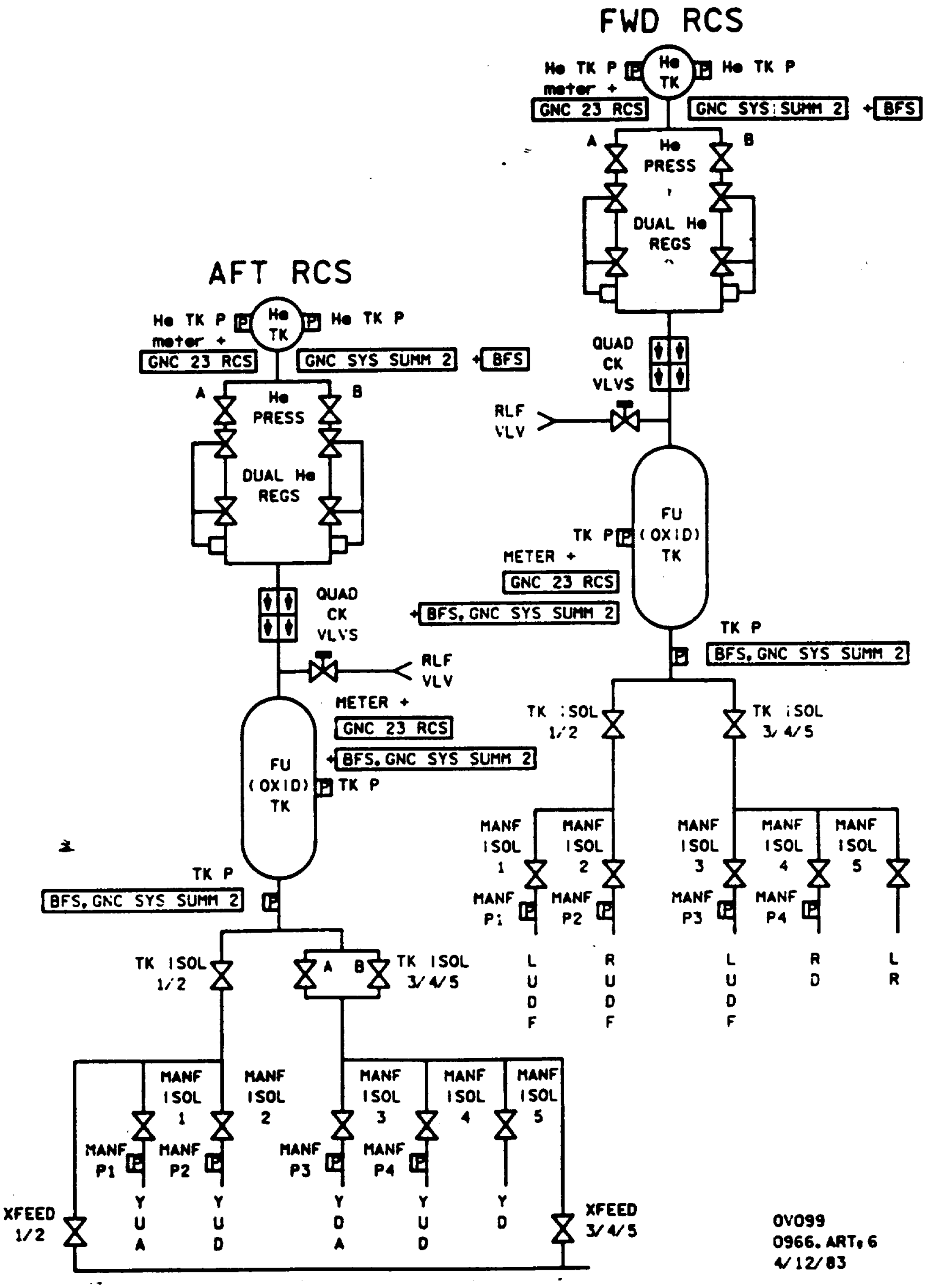
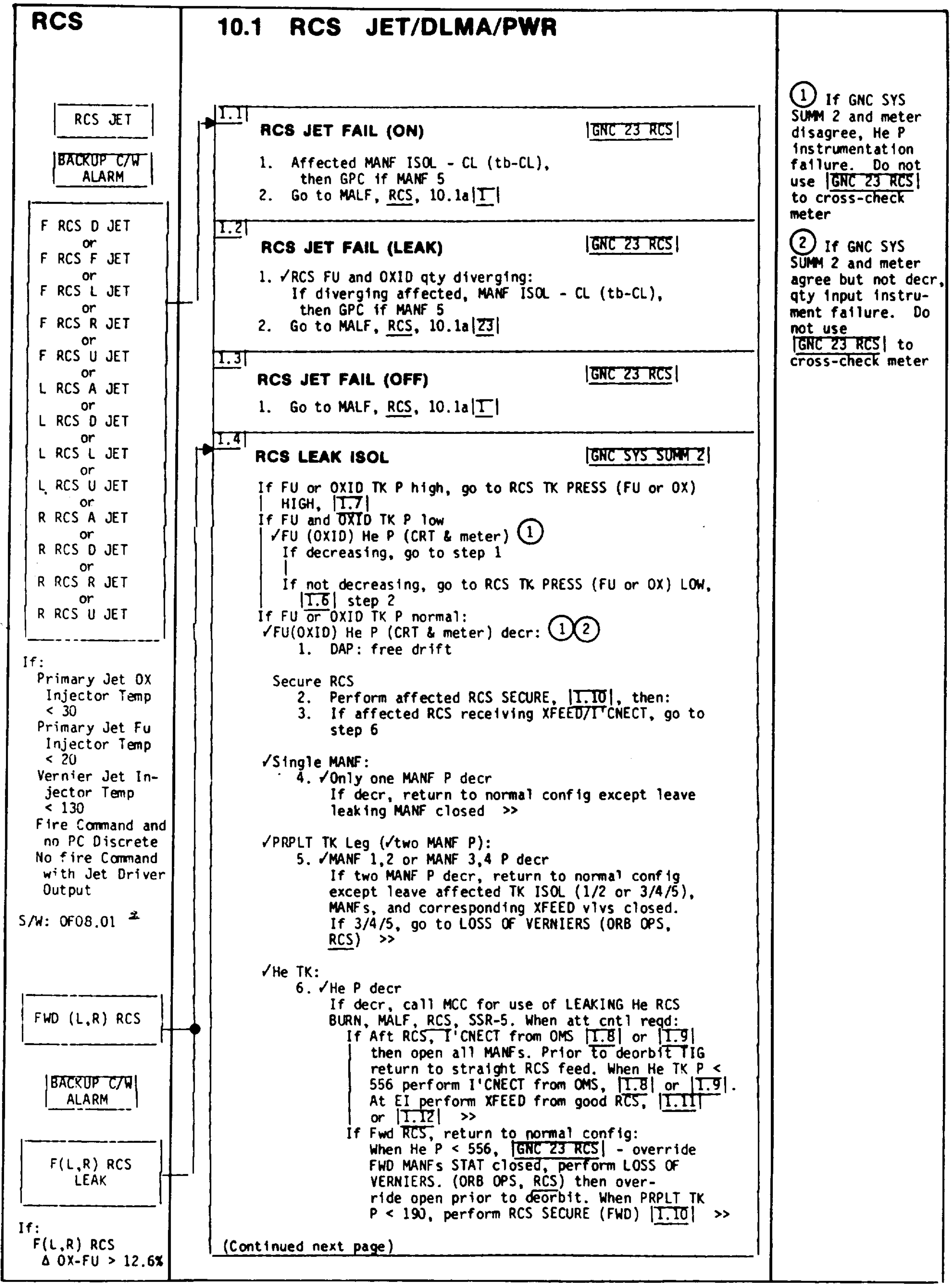


Figure 4.2: The RCS System



RCS

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%

10.1 RCS JET/DLMA/PWR

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 4.3: Some RCS Malfunction Procedures

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 system (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 system is given in Appendix C.)

```
(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 system?" 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 system. 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 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 application of the RCS system 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.

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

10.1 RCS JET/DLMA/PWR

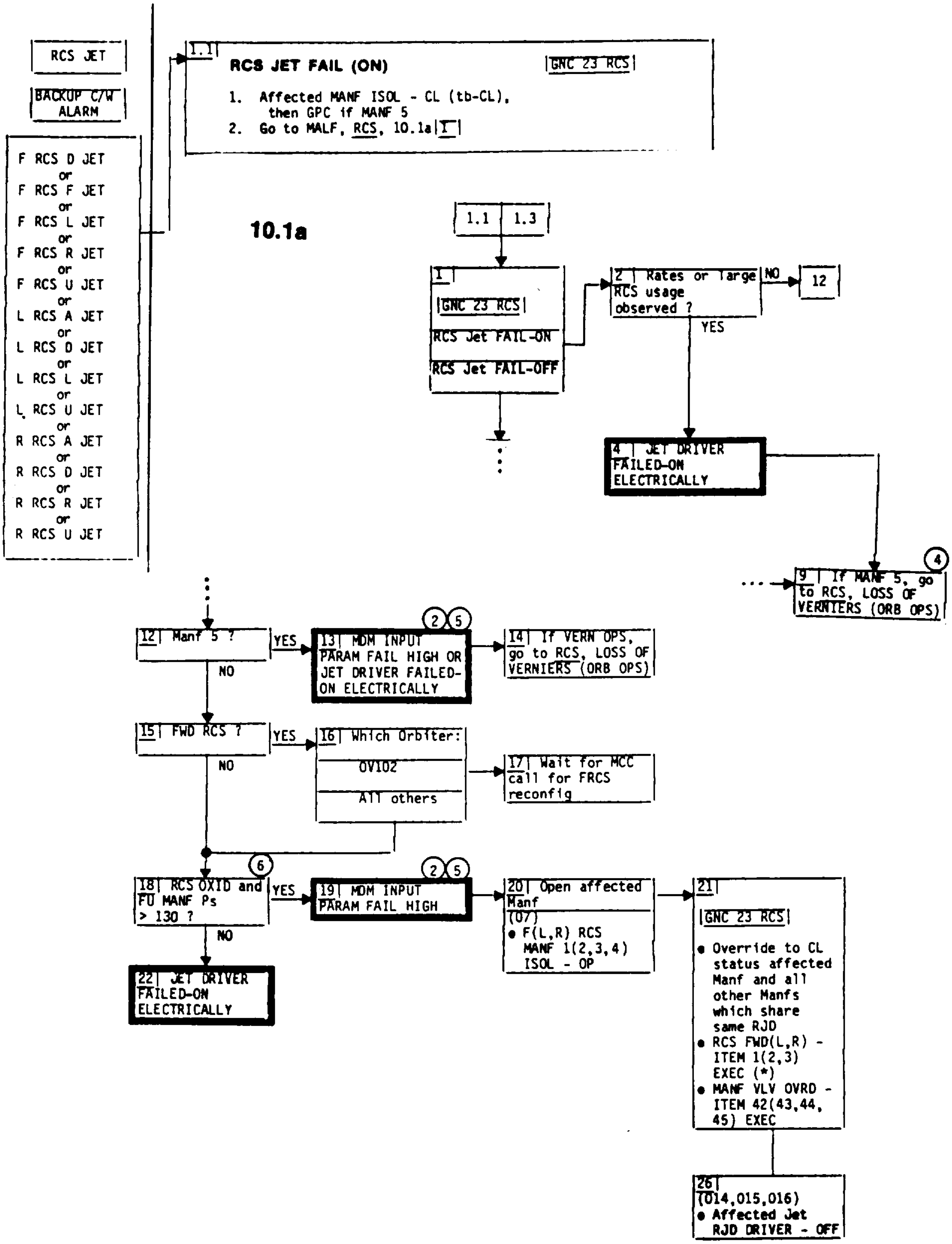
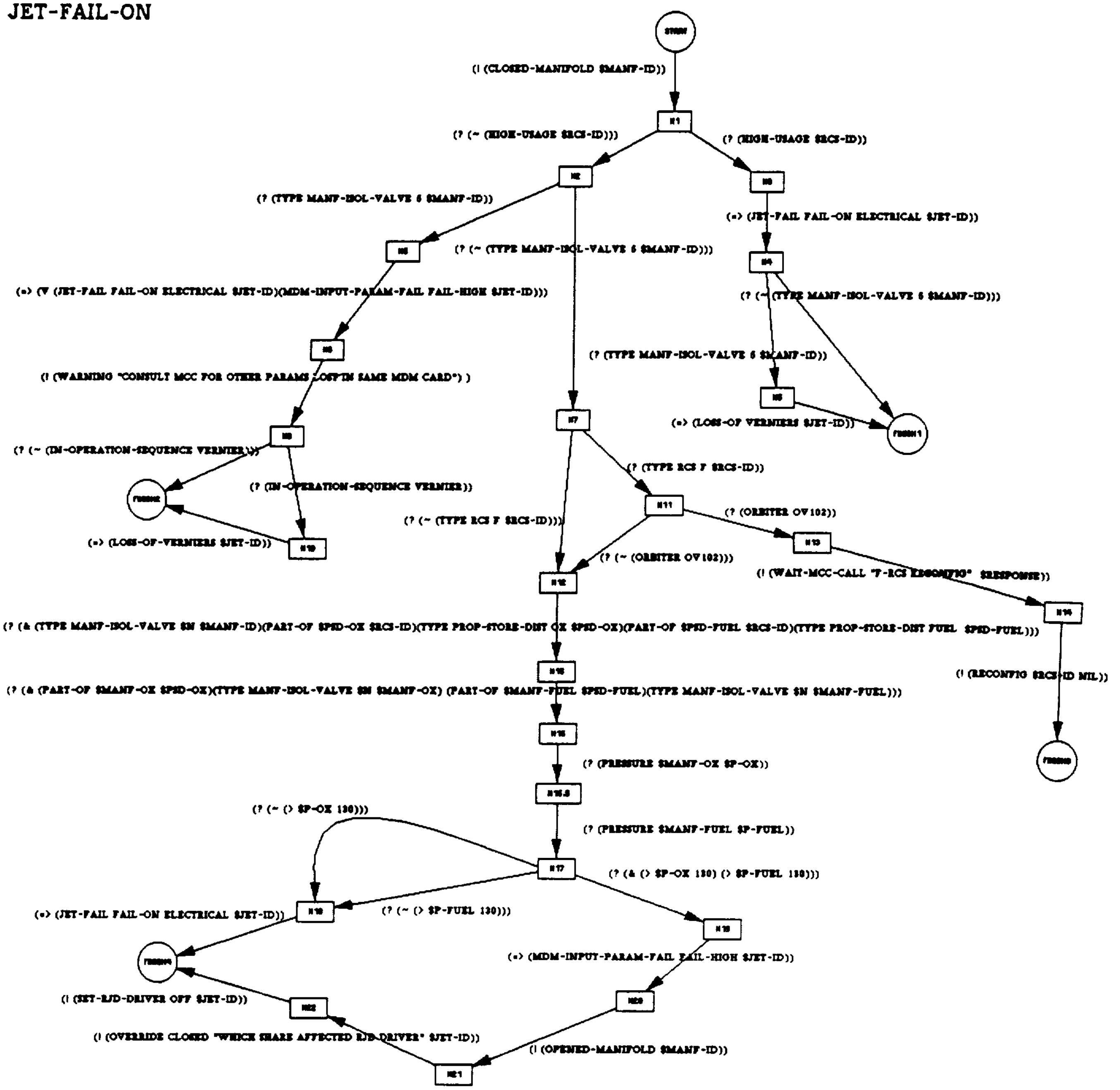

RCS JET
BACKUP C/W ALARM
F RCS D JET or F RCS F JET or F RCS L JET or F RCS R JET or F RCS U JET or L RCS A JET or L RCS D JET or L RCS L JET or L RCS U JET or R RCS A JET or R RCS D JET or R RCS R JET or R RCS U JET
1.1 RCS JET FAIL (ON) GNC 23 RCS
1. Affected MANF ISOL - CL (tb-CL), then GPC if MANF 5
2. Go to MALF, RCS, 10.1a I
10.1a
1.1 1.3
1 GNC 23 RCS
RCS Jet FAIL-ON
RCS Jet FAIL-OFF
2 Rates or Targe RCS usage observed ?
NO
12
YES
4 JET DRIVER FAILED-ON ELECTRICALLY
4
9 If MANF 5, go to RCS, LOSS OF VERNIERS (ORB OPS)
12 Manf 5 ?
YES
NO
2 5
13 MDM INPUT PARAM FAIL HIGH OR JET DRIVER FAILED-ON ELECTRICALLY
14 If VERN OPS, go to RCS, LOSS OF VERNIERS (ORB OPS)
15 FWD RCS ?
YES
NO
16 Which Orbiter:
OV102
All others
17 Wait for MCC call for FRCS reconfig
6
18 RCS OXID and FU MANF Ps > 130 ?
YES
NO
2 5
19 MDM INPUT PARAM FAIL HIGH
20 Open affected Manf
(O7)
• F(L,R) RCS MANF 1(2,3,4) ISOL - OP
21
GNC 23 RCS
• Override to CL status affected Manf and all other Manfs which share same RJD
• RCS FWD(L,R) - ITEM 1(2,3) EXEC (*)
• MANF VLV OVRD - ITEM 42(43,44, 45) EXEC
22 JET DRIVER FAILED-ON ELECTRICALLY
26
(O14,O15,O16)
• Affected Jet RJD DRIVER - OFF

Figure 4.4: RCS JET FAIL (ON) Malfunction Procedure

JET-FAIL-ON


START
(! (CLOSED-MANIFOLD $MANF-ID))
N1
(? (~ (HIGH-USAGE $RCS-ID)))
(? (HIGH-USAGE $RCS-ID))
(? (TYPE MANF-ISOL-VALVE 6 $MANF-ID))
(? (~ (TYPE MANF-ISOL-VALVE 6 $MANF-ID)))
(=> (JET-FAIL FAIL-ON ELECTRICAL $JET-ID))
(=> (V (JET-FAIL FAIL-ON ELECTRICAL $JET-ID)(MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID)))
(? (~ (TYPE MANF-ISOL-VALVE 6 $MANF-ID)))
(? (TYPE MANF-ISOL-VALVE 6 $MANF-ID))
(! (WARNING "CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
(=> (LOSS-OF VERNIERS $JET-ID))
FINISH1
(? (~ (IN-OPERATION-SEQUENCE VERNIER)))
(? (TYPE RCS F $RCS-ID))
(? (IN-OPERATION-SEQUENCE VERNIER))
N11
(? (ORBITER OV102))
(? (~ (TYPE RCS F $RCS-ID)))
(=> (LOSS-OF-VERNIERS $JET-ID))
(? (~ (ORBITER OV102)))
(! (WAIT-MCC-CALL "F-RCS RECONFIG" $RESPONSE))
(? (& (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-FUEL)))
(! (RECONFIG $RCS-ID NIL))
(? (& (PART-OF $MANF-OX $PSD-OX)(TYPE MANF-ISOL-VALVE $N $MANF-OX) (PART-OF $MANF-FUEL $PSD-FUEL)(TYPE MANF-ISOL-VALVE $N $MANF-FUEL)))
(? (PRESSURE $MANF-OX $P-OX))
(? (~ (> $P-OX 130)))
(? (PRESSURE $MANF-FUEL $P-FUEL))
N17
(? (& (> $P-OX 130) (> $P-FUEL 130)))
(=> (JET-FAIL FAIL-ON ELECTRICAL $JET-ID))
(? (~ (> $P-FUEL 130)))
(=> (MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID))
(! (SET-RJD-DRIVER OFF $JET-ID))
(! (OVERRIDE CLOSED "WHICH SHARE AFFECTED RJD-DRIVER" $JET-ID))
(! (OPENED-MANIFOLD $MANF-ID))
N21

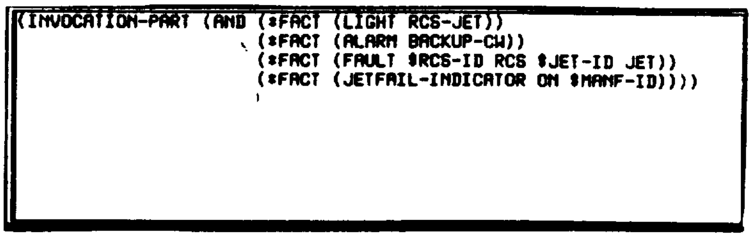

```
(INVOCATION-PART (AND (*FACT (LIGHT RCS-JET))
                      (*FACT (ALARM BACKUP-CW))
                      (*FACT (FAULT $RCS-ID RCS $JET-ID JET))
                      (*FACT (JETFAIL-INDICATOR ON $MANF-ID))))
```

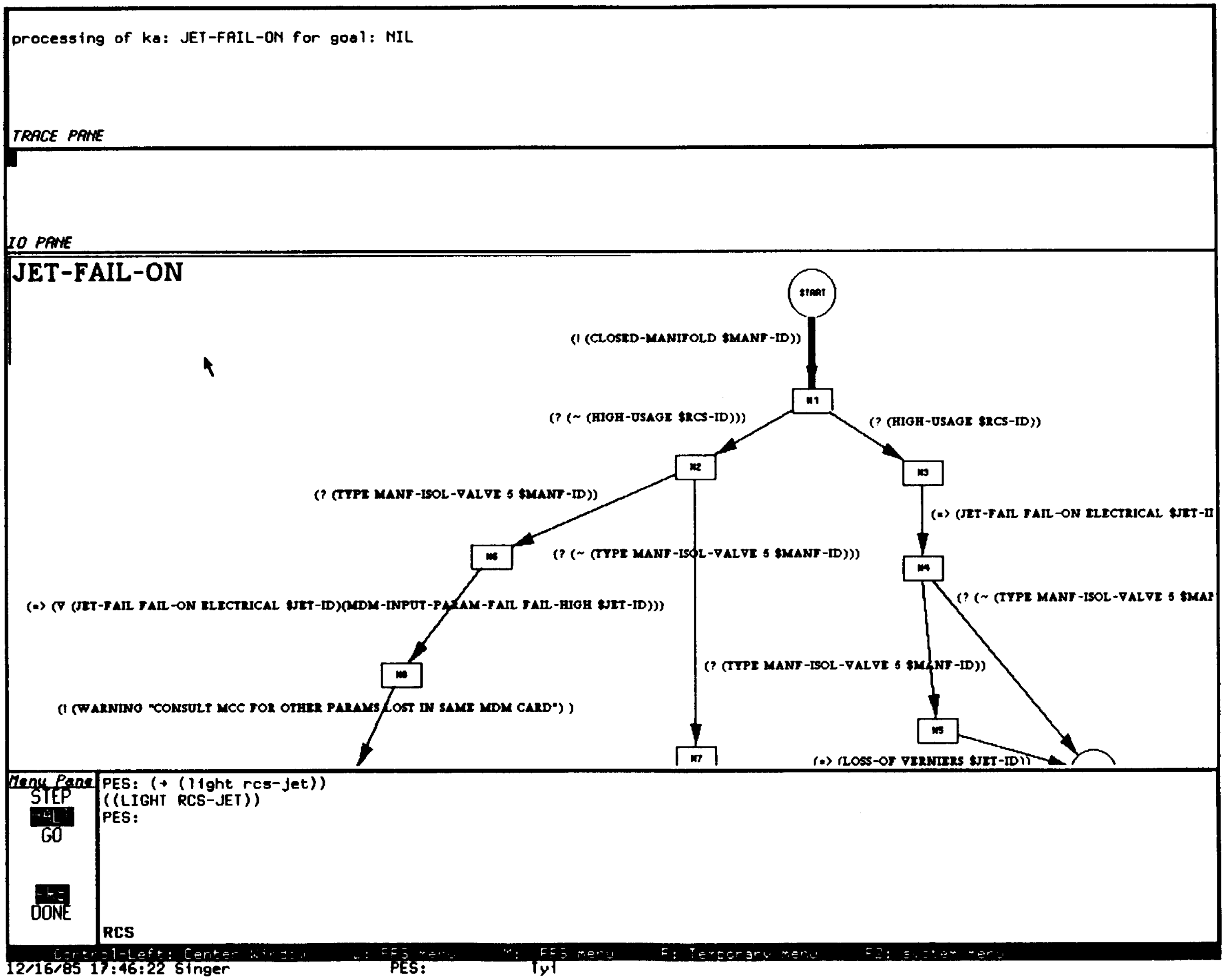
Figure 4.5: JET-FAIL-ON KA

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 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.



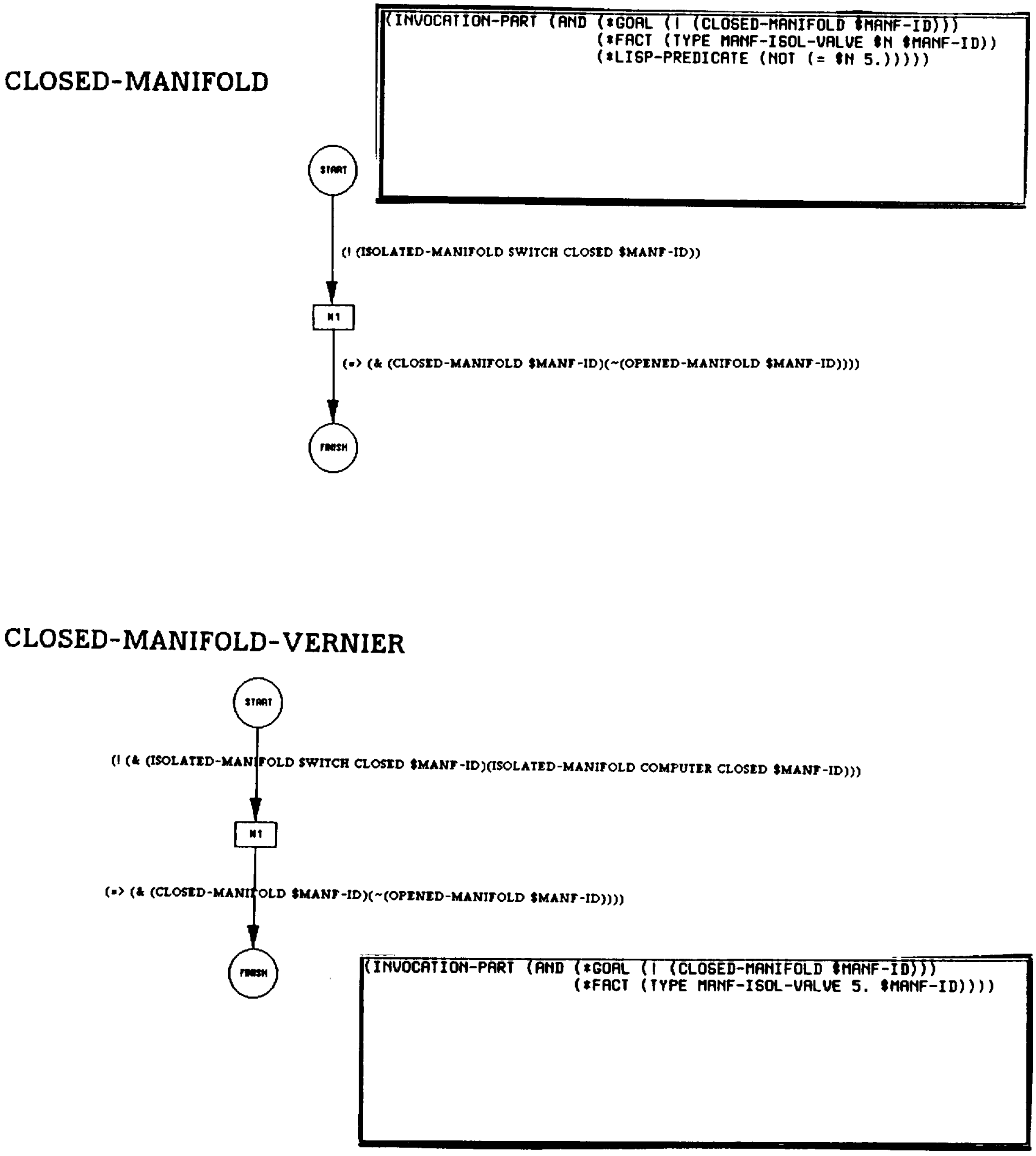
processing of ka: JET-FAIL-ON for goal: NIL

TRACE PANE
IO PANE
JET-FAIL-ON
START
(! (CLOSED-MANIFOLD $MANF-ID))
N1
(? (~ (HIGH-USAGE $RCS-ID)))
(? (HIGH-USAGE $RCS-ID))
N2
N3
(? (TYPE MANF-ISOL-VALVE 5 $MANF-ID))
(=> (JET-FAIL FAIL-ON ELECTRICAL $JET-II
N6
(? (~ (TYPE MANF-ISOL-VALVE 5 $MANF-ID)))
N4
(? (~ (TYPE MANF-ISOL-VALVE 5 $MAI
(=> (V (JET-FAIL FAIL-ON ELECTRICAL $JET-ID)(MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID)))
N8
(? (TYPE MANF-ISOL-VALVE 5 $MANF-ID))
(! (WARNING "CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
N5
N7
(=> (LOSS-OF VERNIERS $JET-ID))
Menu Pane
STEP
GO
DONE
Control-Left: Center Window L: PPS menu M: PPS menu R: Temporary menu R2: system menu

```
PES: (+ (light rcs-jet))
((LIGHT RCS-JET))
PES:

RCS
```

```
12/16/85 17:46:22 Singer                PES:          Tyi
```

Figure 4.6: The CLOSED-MANIFOLD Step.



CLOSED-MANIFOLD
START
(! (ISOLATED-MANIFOLD SWITCH CLOSED $MANF-ID))
N1
(=> (& (CLOSED-MANIFOLD $MANF-ID)(~(OPENED-MANIFOLD $MANF-ID))))
FINISH
CLOSED-MANIFOLD-VERNIER
START
(! (& (ISOLATED-MANIFOLD SWITCH CLOSED $MANF-ID)(ISOLATED-MANIFOLD COMPUTER CLOSED $MANF-ID)))
N1
(=> (& (CLOSED-MANIFOLD $MANF-ID)(~(OPENED-MANIFOLD $MANF-ID))))
FINISH

```
(INVOCATION-PART (AND (*GOAL (! (CLOSED-MANIFOLD $MANF-ID)))
                      (*FACT (TYPE MANF-ISOL-VALVE $N $MANF-ID))
                      (*LISP-PREDICATE (NOT (= $N 5.)))))
```

```
(INVOCATION-PART (AND (*GOAL (! (CLOSED-MANIFOLD $MANF-ID)))
                      (*FACT (TYPE MANF-ISOL-VALVE 5. $MANF-ID))))
```

Figure 4.7: KAs for Closing a Manifold.

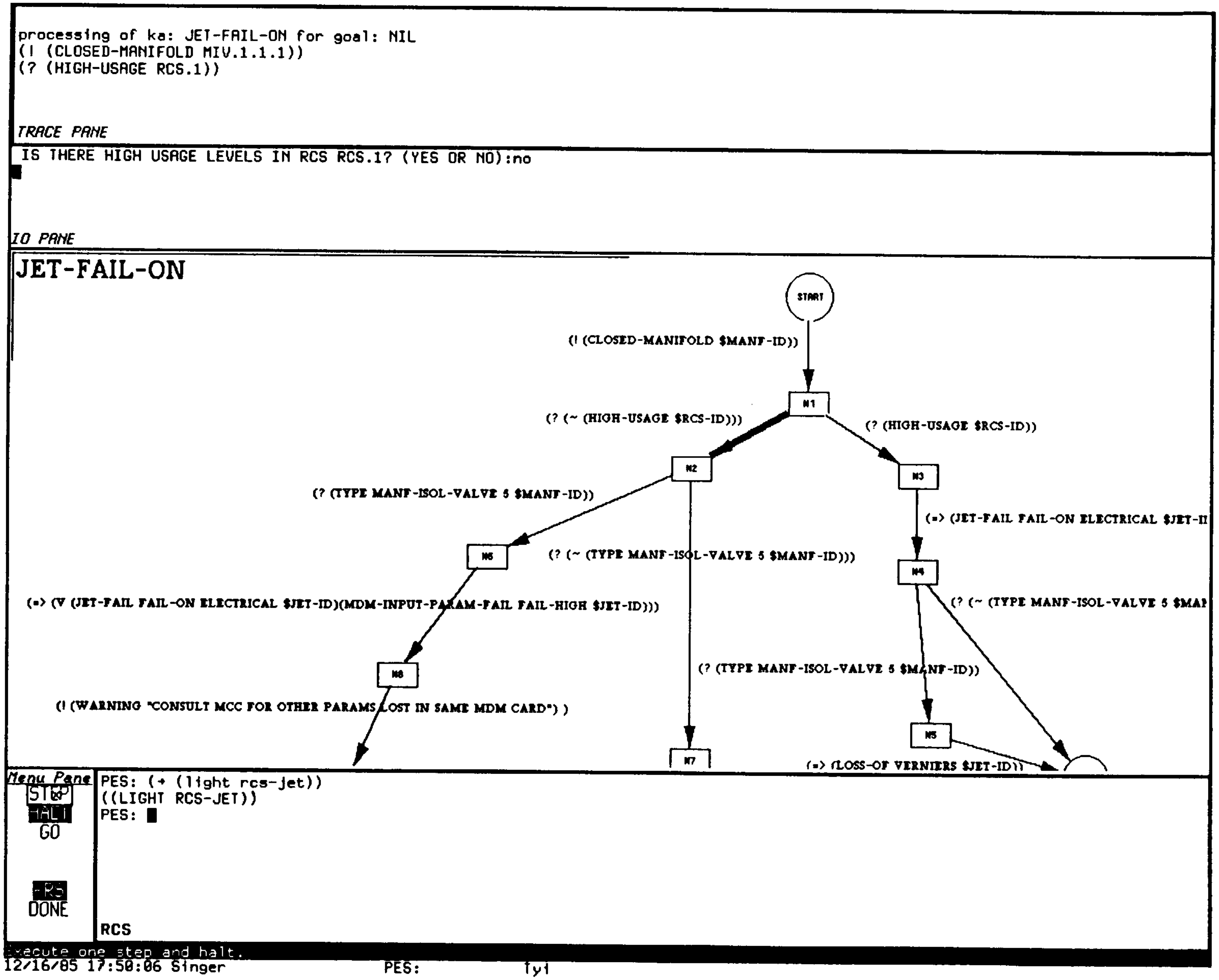
It is precisely the lack of this kind of goal semantics and reasoning ability that caused a recent space shuttle flight to abort. Although the shuttle system knew that a particular manifold was closed, it found itself unable to proceed when an instruction of the form "close the manifold" was given to 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 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



```
processing of ka: JET-FAIL-ON for goal: NIL
(! (CLOSED-MANIFOLD MIV.1.1.1))
(? (HIGH-USAGE RCS.1))
```

```
IS THERE HIGH USAGE LEVELS IN RCS RCS.1? (YES OR NO):no
```

IO PANE
JET-FAIL-ON
START
(! (CLOSED-MANIFOLD $MANF-ID))
N1
(? (~ (HIGH-USAGE $RCS-ID)))
(? (HIGH-USAGE $RCS-ID))
N2
N3
(? (TYPE MANF-ISOL-VALVE 5 $MANF-ID))
(=> (JET-FAIL FAIL-ON ELECTRICAL $JET-II
N6
(? (~ (TYPE MANF-ISOL-VALVE 5 $MANF-ID)))
N4
(=> (V (JET-FAIL FAIL-ON ELECTRICAL $JET-ID)(MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID)))
(? (~ (TYPE MANF-ISOL-VALVE 5 $MAI
(? (TYPE MANF-ISOL-VALVE 5 $MANF-ID))
N8
(! (WARNING "CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
N5
N7
(=> (LOSS-OF VERNIERS $JET-ID))
Menu Pane
STEP
HALT
GO
RS
DONE

```
PES: (+ (light rcs-jet))
((LIGHT RCS-JET))
PES: █
```

```
Execute one step and halt.
12/16/85 17:50:06 Singer                PES:          Tyi
```

Figure 4.8: HIGH-USAGE Step.

form (!(¬*p*)) or (?(¬*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 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.

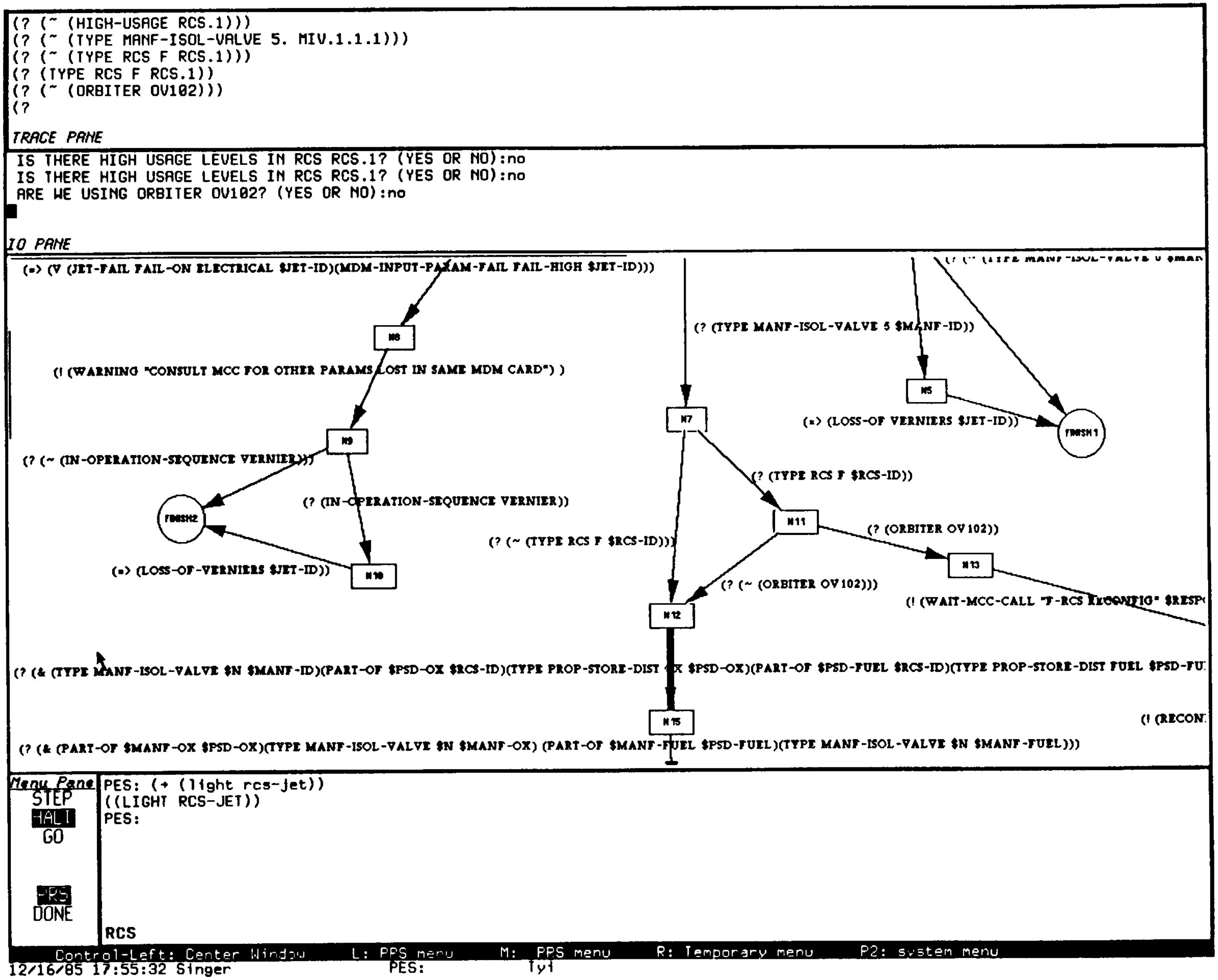
The JET-FAIL-ON KA might next proceed to ensuing goals of the form

```
(?  (¬ (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



```
(? (~ (HIGH-USAGE RCS.1)))
(? (~ (TYPE MANF-ISOL-VALVE 5. MIV.1.1.1)))
(? (~ (TYPE RCS F RCS.1)))
(? (TYPE RCS F RCS.1))
(? (~ (ORBITER OV102)))
(?
```

TRACE PANE

```
IS THERE HIGH USAGE LEVELS IN RCS RCS.1? (YES OR NO):no
IS THERE HIGH USAGE LEVELS IN RCS RCS.1? (YES OR NO):no
ARE WE USING ORBITER OV102? (YES OR NO):no
```

IO PANE

(=> (V (JET-FAIL FAIL-ON ELECTRICAL $JET-ID)(MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID)))
N8
(! (WARNING "CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
N9
(? (~ (IN-OPERATION-SEQUENCE VERNIER)))
(? (IN-OPERATION-SEQUENCE VERNIER))
FINISH2
(=> (LOSS-OF-VERNIERS $JET-ID))
N10
(? (TYPE MANF-ISOL-VALVE 5 $MANF-ID))
(? (~ (TYPE MANF-ISOL-VALVE 5 $MAN
N5
N7
(=> (LOSS-OF VERNIERS $JET-ID))
FINISH1
(? (TYPE RCS F $RCS-ID))
N11
(? (ORBITER OV102))
(? (~ (TYPE RCS F $RCS-ID)))
N13
(? (~ (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 MANF-ISOL-VALVE $N $MANF-OX) (PART-OF $MANF-FUEL $PSD-FUEL)(TYPE MANF-ISOL-VALVE $N $MANF-FUEL)))
Menu Pane
STEP
HALT
GO
PPS
DONE

```
PES: (+ (light rcs-jet))
((LIGHT RCS-JET))
PES:



RCS
```

Control-Left: Center Window   L: PPS menu   M: PPS menu   R: Temporary menu   P2: system menu
12/16/85 17:55:32 Singer   PES: Tyi

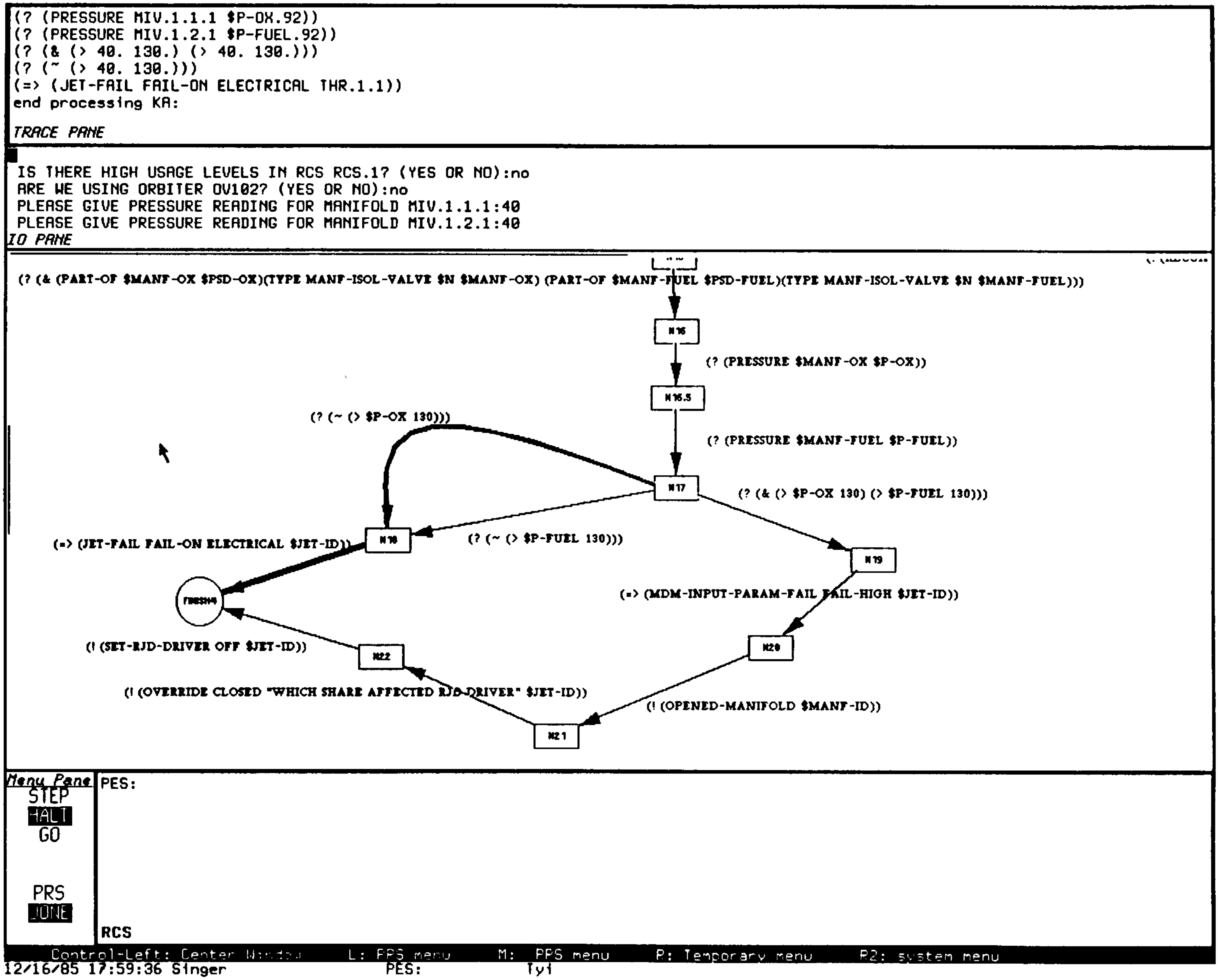
Figure 4.9: Unification of a Large Conjunct of Facts.

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 "correspondance" is very much tied in to the structure of the RCS system 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).

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 system. If such an action were undertaken and overseen by PES, the new structure of the RCS system (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.



```
(? (PRESSURE MIV.1.1.1 $P-OX.92))
(? (PRESSURE MIV.1.2.1 $P-FUEL.92))
(? (& (> 40. 130.) (> 40. 130.)))
(? (~ (> 40. 130.)))
(=> (JET-FAIL FAIL-ON ELECTRICAL THR.1.1))
end processing KA:
```

```
IS THERE HIGH USAGE LEVELS IN RCS RCS.1? (YES OR NO):no
ARE WE USING ORBITER OV102? (YES OR NO):no
PLEASE GIVE PRESSURE READING FOR MANIFOLD MIV.1.1.1:40
PLEASE GIVE PRESSURE READING FOR MANIFOLD MIV.1.2.1:40
```

IO PANE
(? (& (PART-OF $MANF-OX $PSD-OX)(TYPE MANF-ISOL-VALVE $N $MANF-OX) (PART-OF $MANF-FUEL $PSD-FUEL)(TYPE MANF-ISOL-VALVE $N $MANF-FUEL)))
N16
(? (PRESSURE $MANF-OX $P-OX))
N16.5
(? (PRESSURE $MANF-FUEL $P-FUEL))
N17
(? (~ (> $P-OX 130)))
(? (& (> $P-OX 130) (> $P-FUEL 130)))
(? (~ (> $P-FUEL 130)))
N18
N19
(=> (JET-FAIL FAIL-ON ELECTRICAL $JET-ID))
(=> (MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID))
FINISHED
N20
(! (SET-RJD-DRIVER OFF $JET-ID))
N22
(! (OVERRIDE CLOSED "WHICH SHARE AFFECTED RJD-DRIVER" $JET-ID))
(! (OPENED-MANIFOLD $MANF-ID))
N21
Menu Pane
STEP
HALT
GO
PRS
DONE
PES:
RCS
Control-Left: Center Window
L: PPS menu
M: PPS menu
P: Temporary menu
R2: system menu
12/16/85 17:59:36 Singer
PES:
Tyi

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

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.

Our experiences so far have found the efficiency of PES to be quite adequate to applications like the RCS system. While the system may not be suited for complicated numerical calculations, it seems to be a fine medium for malfunction diagnosis or other higher-level reasoning tasks. A more rigorous test of the system should, of course, be undertaken with an application consisting of many more KAs.

Chapter 5

Theoretical Considerations

In this chapter we discuss some of the theoretical results arising from our work.

5.1 Declarative Semantics

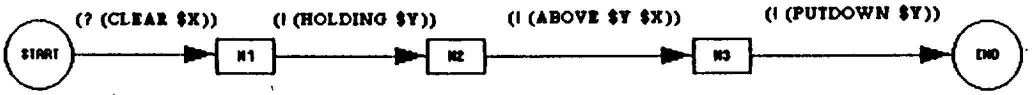
One of the most important features of procedural expert systems is that they have a declarative semantics. The body of a KA is intended to represent true statements about the problem domain under consideration. Unlike statements in standard logic, which can be viewed as describing a single state of the world, statements in the body of a KA are temporal facts about *sequences* of world states.

For example, consider the following simple KA:

INVOCATION: (GOAL (! (ON $Y $X)))

EFFECTS: (! (ON $Y $X))

BODY: .



The intended meaning of this is that, *for all* $x and $y, if `clear $x` holds initially, and after some time `(holding $y)` is true, and then `(above $y $x)` becomes true, and finally `(putdown $y)` becomes true, *then it follows that* `(on $y $x)` will be [finally] true. Note that nothing is said about *how* the various conditions on world state (such as `(holding $y)`) will be brought about – the KA simply states a declarative fact about the world, describing how one block ($y) may end up on top of another ($x).

The declarative semantics of KAs is an essential precondition if the system is to possess all the desirable properties of expert systems, including 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. The system can reason about composite goals and can determine, for example, that a given conjunctive goal could be realized by achieving all the component goals, one after the other. System evolvability is ensured because each KA 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 KA can be examined irrespective of the definition of any other KA. Thus, once all KAs have been independently validated, we can be certain that *no* situation could possibly arise that would cause these KAs to be executed incorrectly.

KAs also have an operational semantics that enables them to be used to achieve desired goals. For example, the blocks-world KA given above not only states a fact about the world, but also describes a way of achieving a state in which block $y is on block $x if that is one of the system's current goals. That is, if you want to reach a world in which, *for some* $x and $y, `(on $x $y)` holds, do some test to determine whether $x is clear, and if it is, try to achieve each of the following goals (`(holding $y)`, `(above $y $x)`, and `(putdown $y)`) in the order given. The test for determining `(clear $x)` may involve a lot of other actions and tests – this is fine, provided the test leaves the condition `(clear $x)` true at the beginning if it was true at the end. Similarly, attempts to achieve any of the other goals in the body of the KA could involve further tests and actions.

In other words, we can view a KA either as describing a procedure to achieve something (its operational semantics), or as a statement about the way the world is (its declarative 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 [23], and the combination of an operational and a declarative semantics is one of the major reasons for the success of Prolog [27]. However, Prolog and most knowledge representation languages (e.g., predicate calculus, rule- or frame-based languages [1,47]) 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.

The declarative semantics outlined above is very informal. It is important that this be formalized, and the soundness and completeness of the system investigated. This is not a simple problem, especially given that actions and tests can have side effects. Indeed, this is a very difficult problem even for conventional programming languages [33]. The first steps to formalize the system are described in Appendix A.

In providing a declarative semantics for procedural expert systems, we also need a means for describing the primitive tests and actions performed by sensors and effectors. Currently, the performance of tests and the execution of actions are 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. If the goal is not achieved, then the corresponding arc in the calling KA will not be traversed (unless, of course, some other means is found to achieve the desired goal).

Neither is there any 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. Currently, the system cannot handle this type of knowledge. The semantics and implementation of such a capability is not straightforward, especially if one wishes to allow for concurrency, and would require further research.

5.2 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 [29]), 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 [16,27]. As Hayes says [23] : "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)?

5.3 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 [41] applies to various state predicates, and implements this default assumption by means of a metalevel KA.

Chapter 6

Personnel and Publications

In addition to the work described herein, this project included, as separate components, a study of automation and robotics technology for the proposed space station and a research and development plan for AI-based technology. For completeness, the personnel and publications listed below include all components of the project.

6.1 Personnel

A number of researchers worked on the part of the project described herein. Michael Georgeff, Amy Lansky, and Pierre Bessiere developed most of the theory and a substantial part of the implementation. Mabry Tyson and Joshua Singer were also involved in implementation, and significantly upgraded the performance of the system. Marcel Schoppers helped test the system on various applications. Michael Georgeff was the Principal Investigator on this part of the project.

The other parts of the project involved the following personnel: Oscar Firschein, Michael Georgeff, William Park, Peter Cheeseman, Jacob Goldberg, Peter Neumann, William Kautz, Karl Levitt, Raphael Rom, and Andrew Poggio. Oscar Firschein was the Principal Investigator on these parts of the project.

6.2 Major Publications

Georgeff, M.P., "A Theory of Action for Multiagent Planning," *Proceedings of the National Conference on Artificial Intelligence*, Austin, Texas (1984).

Georgeff, M.P., A. Lansky, and P. Bessiere, "A Procedural Logic," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, California, 1985.

Georgeff, M.P., "Reasoning about Procedural Knowledge," *Proceedings of the AIAA/-ACM/IEEE Computers in Aerospace Conference*, Long Beach, California, 1985.

Georgeff, M.P., "A Theory of Process," *Proceedings of the Workshop on Distributed Artificial Intelligence*, Sea Ranch, California, 1985.

Georgeff, M.P., "An Expert System for Representing Procedural Knowledge," Mid-term Report, SRI International, Menlo Park, California, 1985.

Georgeff, M.P., and O. Firschein, "Expert Systems for Space Station Automation," *IEEE Control Systems*, Vol. 5, pp 3-8, 1985.

Firschein, O. and M.P. Georgeff, "Review of Automation Technologies Applicable to the National Space Program," *AIAA/NASA Symposium on Automation, Robotics, and Advanced Computing for the National Space Program*, Washington, D.C., 1985.

Park, W., and O. Firschein, "“Space Station Automation: The Role of Robotics and Artificial Intelligence," *SPIE Workshop, Space Station Automation*, Cambridge, Massachusetts, 1985.

Park, W., "Space Station Applications for Teleoperation and Robotics," *First Annual Workshop on Robotics and Expert Systems*, Houston, Texas, 1985.

Firschein, O., M.P. Georgeff, W. Park, P. Cheeseman, J. Goldberg, P. Neumann, W.H. Kautz, K. Levitt, R. Rom, and A. Poggio, "NASA Space Station Automation: AI-Based Technology Review," Final Report, Artificial Intelligence Center, SRI International, Menlo Park, California, 1984.

Firschein, O., M.P. Georgeff, W. Park, and P. Cheeseman, "A NASA Initiative in AI-Based Technology: R & D Plan," Final Report, Artificial Intelligence Center, SRI International, Menlo Park, California, 1984.

6.3 Major Presentations

Speaker at the Stanford Computer Science Colloquium, Stanford University, Stanford, California, July, 1984 (M. Georgeff).

Paper presented at the National Conference on Artificial Intelligence, Austin, Texas, August, 1984 (M. Georgeff).

Speaker at the Colloquium of the Center for the Study of Language and Information, Stanford University, Stanford, California, February, 1985 (M. Georgeff).

Speaker at the NASA Ames' Director's Colloquium, August, 1985 (M. Georgeff).

Paper presented at the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California, August, 1985 (M. Georgeff).

Paper presented at the AIAA/ACM/IEEE Computers in Aerospace Conference, Long Beach, California, October, 1985 (M. Georgeff).

Paper presented at the Workshop on Distributed Artificial Intelligence, Sea Ranch, California, December, 1985 (M. Georgeff).

Presentations on the Space Station Automation Study, Johnson Space Center, November 1984 (O. Firschein and W. Park).

Paper presented at the First Annual Workshop on Robotics and Expert Systems, Houston, Texas, June 1985 (W. Park).

Paper presented to the Congress of the US, Office of Technology Assessment, OTA Workshop on Automation and Robotics for the Space Station, July, 1985 (O. Firschein).

Paper presented at the AIAA/NASA Symposium on Automation, Robotics, and Advanced Computing, Washington, D.C., September 1985 (O. Firschein).

Appendix A

Semantics of the Procedural Representation

This appendix describes the semantics of the process representation used in procedural expert systems. Some of the terminology differs from that in the main body of this report; however, the material is self contained and the difference in terminology should present no difficulties. The aim of this work was to provide a formal semantics for a subset of the system being developed. A version of this material, entitled "A Procedural Logic," was published in the *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, held in Los Angeles, California, in August, 1985.

A.1 Introduction

Active intelligent systems need to be able to represent and reason about actions and how those actions can be combined to achieve given goals. Much of this knowledge is in the form of sequences of actions or "procedures" for accomplishing these goals. For example, knowledge about kicking a football, performing a certain dance movement, cooking a roast dinner, solving Rubik's cube, or diagnosing an engine malfunction, is primarily procedural in nature.

Within AI, there have been two approaches to the problem of action and practical reasoning, 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 ([2,25,32]). 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 [3,11,43,45,48,50,52,53]. 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 basis of planning, is relatively rare.

In attacking this problem, we first have to identify what it is that humans or other active systems do when performing a complex action. We postulate that such systems have some representation of a procedure for achieving given goals, or reacting to particular events, and that they can reason about and execute this procedure to achieve their aims. Just as we might view intelligent systems as having "beliefs" about the world, we consider these systems to have "procedures" for acting in the world. And, just as for theories of belief, the problem here is to provide abstract models for these "mental entities." We call these abstract models *processes*.

There are two aims to our work. One is to develop a theory suitable for building active intelligent agents. In that regard, the theory presented in this paper models only the simplest kind of agent – one with no preserved beliefs and with limited reasoning abilities. We define a declarative semantics for our formalism, as well as an operational semantics. Together these provide a suitable semantics for simple action sentences in natural language and a method of practical reasoning about how to accomplish given goals.

The other aim is to provide a basis for the design of improved programming languages – in particular, languages that allow users to represent their knowledge about the behavior of systems declaratively, are amenable to verification, and operationally are flexible and responsive to environmental changes. In this sense, our work can be viewed as the basis for executable specification languages.

It is important to point out that the theory presented here is not just another variant of the standard logics for describing dynamic behaviors. In particular, there is no existing logic (temporal, dynamic or interval-based) known to us that can both (1) express the same complexity of action as the formalism proposed here (which can handle sequencing, conditional selection, nondeterministic choice, iteration, and hierarchical abstraction), and (2) be used to ***automatically*** generate behaviors for achieving goals and to form plans. In this sense, the approach here offers the same kind of advantages as Prolog, but in a dynamic rather than static domain: it can be viewed as a logic describing properties of behaviors, or it can be used as a programming language for generating behaviors to achieve given goals.

Furthermore, the model we use is based on nondeterministic procedures. This nondeterminism is essential for providing the kind of flexibility exhibited by intelligent systems. The model also allows for action failures and tests with side effects, both of which are necessary for handling most real-world domains. Such a model would be very cumbersome to describe in any of the standard temporal or dynamic logics - indeed, we know of none that have attempted to do so.

The more recent work includes many capabilities not decribed in this paper, including a data base of preserved "beliefs" and more powerful reasoning abilities represented as metalevel processes.

A.2 Processes and Actions

Most previous work in representing actions has been based on state change models (e.g., [11,30,43]). However, existing models can describe only a limited class of actions and are too weak to be used in dealing with multiagent or dynamic worlds.

Some attempts have recently been made to provide a better underlying theory for actions. McDermott [32] considers an action or event to be a set of sequences of states, and describes a temporal logic for reasoning about such actions and events. Allen [2] also considers an action to be a set of sequences of states, and specifies an action by describing the relationships among the intervals over which the action's conditions and effects are assumed to hold. However, while it is possible to state arbitrary properties of actions and events, it is not obvious how one could use these logics to achieve, or form intentions to achieve, one's goals.[1]

Our notion of action is essentially the same as that of McDermott and Allen; namely, we consider actions to be sets of sequences of world states. However, in

[1] Allen [3] proposes a method of forming plans that is based on his representation of actions. However, he does not use the temporal logic directly, and actions are restricted to a particularly simple form (e.g., they do not include conditionals).

modeling intelligent agents, it is convenient to consider not only states of the external world, but also various "mental entities," such as beliefs, goals and intentions. In the same way, it is important to be able to model not only the actions that occur in the real world, but the internal mental "procedures" that agents use to generate their external behaviors. We will call these entities *processes* (see [19] and, for some early work based on similar ideas, [25]).

We assume that, at any given instant, the world is in a particular *world state*. A *process* is some abstract mechanism that can be executed to generate a sequence of world states, called a *behavior* of the process. The set of all behaviors of a process constitutes the *action* (or *action type*) generated by the process. In this paper we restrict our attention to sequential (nonconcurrent) processes.

Each process is modeled by a labeled transition network, with distinguished start and finish nodes. The nodes of the network are called *control points*, and are labeled with *state conditions*. These conditions can be viewed as representing constraints on possible world states. Each arc of the network is labeled by a *goal*, which can be considered to represent a particular type of *behavior* to be achieved.[2] Associated with each network is an *effect*, which is the goal that will be achieved if the process is successfully executed.

A process is *executed* in the following manner. At any moment during execution, the process is at a given control point c. An outgoing arc a may be traversed if (1) the current state of the world satisfies the state condition labeling c and (2) the goal labeling a is *successfully* achieved. If no outgoing arc from c can be traversed, process execution *fails*. Execution begins with control at the initial control point and *succeeds* if control reaches the final control point.

In some ways, a process may be viewed as just a convenient way of specifying actions. However, processes also allow us to make a distinction that is critical for practical reasoning - we can distinguish between behaviors that are *successful* executions of the process and those that are unsuccessful (or have *failed*). Since actions often fail to achieve their intended goals, it is important to be able to reason explicitly about the consequences of action failure. We thus need to be able to represent the behaviors that correspond to failed actions as well as successful ones. This is particulary important if the model is to be extended to handle multiagent and dynamic environments

[2] In Section 6 we show how a goal to achieve a given *state* can be represented as a type of behavior.

(e.g., see [26]). Similarly, in natural-language understanding, it is important to have a denotation for action sentences (such as "he was painting a picture") that allows for action failure, even in mid-performance ("he was painting a picture when killed by lightning").

The notion of action failure also allows us to represent tests on world states as actions, without the introduction of knowledge or belief structures (cf. [37]). To test whether a particular condition is true, one need simply perform an action that can only succeed when the condition is indeed true. (Of course, action failure cannot, in general, be equated with the falsity of the condition being tested.)

A.3 Process Descriptions

In this section we develop a formalism for describing processes and for reasoning about the behaviors they generate. Each process description consists of descriptions of its *effects* and of its *body*. The body is a network isomorphic to the network of the described process. The state conditions labeling the control points of the underlying process are modeled by expressions which have as their denotation world states; the goals labeling the arcs of the underlying process are modeled by expressions whose denotations are behaviors (sequences of world states). The description of process effects also denotes a set of behaviors.

A typical process description using the formalism is shown in Figure A.1. It describes a procedure for killing someone with a slingshot.

The process involves gathering stones, placing them in a pile, getting a slingshot, and then repeatedly taking up a stone and shooting it until the foe (**$person**) is hit on the head. In this particular domain, hitting someone on the head with a stone hurled by a slingshot always results in that person's death. The procedure is nondeterministic and allows agents to gather as many stones as they wish, limited only by their ability to continue gathering them. The procedure is not guaranteed to be successful – it may fail if any one of the actions labeling the arcs of the network fails. However, if there are only a finite number of gatherable stones, the procedure is guaranteed to terminate.

It is important to note how the process description captures *implicit* knowledge of the problem domain. This knowledge is of two kinds: one concerning the validity of the killing procedure, the other heuristic. For example, hitting a person on the

EFFECTS: (KILL $PERSON)

BODY: .

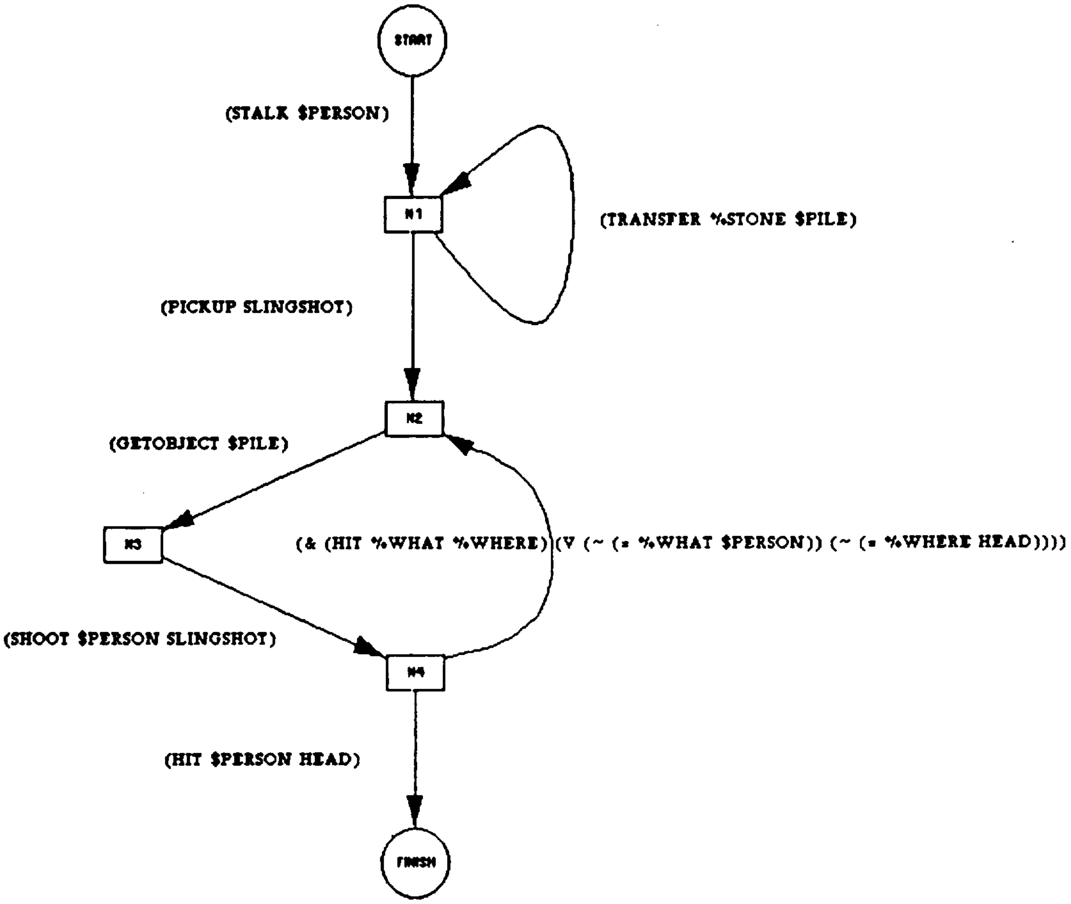


Figure A.1: David and Goliath

head with an object propelled from a slingshot will not always kill them (e.g., if it's a cotton ball), but will if it's a stone (in this particular domain). Thus, the validity of the conclusion depends critically on the first part of the procedure, which ensures that only stones are placed in the pile. (Strictly, the procedure should also ensure that the pile is initially empty or contains nothing but stones.)

The procedure also captures heuristic knowledge in that earlier actions may make subsequent actions more likely to succeed. For example, the slingshot may require a certain size and weight of stone; however, instead of this being represented as an *explicit* precondition of the shooting action, it is represented implicitly by the context established by the procedure. In this case, the assumption is that any stone that can possibly be gathered will most likely possess the appropriate characteristics. Note that this does not affect the validity of the procedure; if a stone does not have the necessary properties, the action of shooting the slingshot will fail.

We now give a definition of the formalism. A *process description* is a tuple

$$P = \langle S, F, N, E, \delta, n_I, n_F, C, A, G \rangle \ ,$$

where

- S is a [possibly infinite] set of *state descriptions*
- F is a [possibly infinite] set of *action descriptions*
- N is a set of *nodes*
- E is a set of *arcs*
- $\delta : N \times E \rightarrow N$ is the *process control function*
- $n_I \in N$ is the *initial node*
- $N_F \subset N$ is a set of *final nodes*
- $C : N \rightarrow S$ associates a state description with each node
- $A : E \rightarrow F$ associates an action description with each arc
- G is an action description called the *effects* of the process.

The state descriptions labeling the nodes are called *[partial] correctness assertions*; the one labeling the initial node is called the *precondition* of the process. The action descriptions labeling the arcs are called *goal assertions*.

We choose predicate calculus as the state description language. A state description can be viewed as denoting a set of states; namely, those in which it is true. We distinguish between *local* and *global* variables. Informally, the interpretation of a local variable is fixed in the interval during which a given arc is transitted, but can otherwise vary. A global variable, on the other hand, has a fixed interpretation during the execution of the entire process. (Local variables are needed in loops when it is necessary to identify different elements from one iteration to the next.) A state description is any formula in this calculus in which all global variables are free and all local variables are bound. In the example of Figure A.1, global variables are prefixed by $ and local variables, assumed to be existentially quantified, by %. All correctness assertions are assumed to be *true*.

An action description consists of an action predicate applied to an n-tuple of terms. Action descriptions denote *action types* or *sets* of state sequences. That is, an expression like "`walk (a, b)`" is considered to denote the set of walking actions from point a to b. Any sequence of states satisfies the action description if it is in the set so denoted. In Section 6 we augment the action description language to include various temporal operators.

A.4 Declarative Semantics

The declarative semantics of process descriptions is intended to describe what is *true* about the underlying system of processes and the world in which they operate. Such a semantics says nothing about *how* such knowledge could be used to achieve particular goals — rather, it simply allows one to state *facts* about certain behaviors.

On an intuitive level, the declarative semantics is straightforward. The intended meaning of a process description P is that every behavior that satisfies the goal and correctness assertions for some path through the net also satisfies the effects of P. Alternatively, one may view the body of P as denoting a set of behaviors - namely, those that satisfy the goal and correctness assertions for some path through the net. Then the intended meaning of P is that each behavior in the set satisfies the effects of P.

Unfortunately, allowing only simple paths through the net will not do. For example, if a node has multiple outgoing arcs, we need to allow several of these arcs to be tried until one is found successful. This is exactly the sort of behavior required of any useful conditional plan or program; if a test on one branch of a conditional fails (returns false), it is necessary to try other branches of the conditional. The problem in this case is that an attempted test may change the state of the world. Thus, paths through the network must allow behaviors that explicitly include failed attempts at realizing tests and actions as well as successful ones.

We now give an informal outline of the semantics of process descriptions. The approach is similar to that used for most temporal logics. We first consider single states. A ***state** s* consists of a set of elements from a domain D together with relations and functions defined over these elements. Assuming a fixed interpretation for each constant symbol in the language, a ***state interpretation** I* assigns to each variable in the language an element of D, to each n-ary predicate symbol an n-ary relation in D, and to each n-adic function symbol an n-adic function in D. The truth-value of a state assertion w in a state s with respect to a state interpretation I is defined in the standard way (variables ranging over elements of D). We can also view w as denoting the set of states in which w is true.

While state interpretations may vary from state to state in the course of a behavior, the interpretation of global variables must remain the same. For a process description P, a *global variable assignment* α is defined to be an assignment of an element in D to each global variable in P. Similarly, for each arc in P, we have a local variable assignment β that associates a value with each local variable used by the goal assertion of that arc. In the course of a behavior satisfying the goal assertion, its local variables may take on at most one value. A state interpretation I is said to be *consistent* with a given α (or β) if the assignment to global (local) variables in I is the same as their assignment in α (β). Note that we do not require a fixed interpretation for predicate symbols or function symbols over the sequence of states in a behavior. We define a *process instance* to be a process description together with consistent global and local variable assignments.

Following the discussion above, we consider the set of behaviors denoted by the body of a process instance as falling into either of two classes, one of which we will call the *success set* of the process instance and the other the ***failure set***. The success set represents all those behaviors that constitute successful executions of the underlying process; the failure set represents all those executions that fail somewhere along the way.

Let $\mathcal{P}$ be a set of process instances and let n be a node in a process instance P.

An element Q of P is said to be applicable to an arc a emanating from n if its effects are included in the set of behaviours described by the goal assertion of a.

The allowed behaviors starting at node n are those in which each applicable process instance at n is tried *at most once* until one succeeds or they all fail. [3] Let $succ(n,a)$ be the set of behaviors consisting of some arbitrary number of unsuccessful attempts by applicable process instances (at most one per process instance) on the arcs emanating from n, followed by a behavior of an applicable process instance that succeeds for some arc a. Each of these attempts, both successful and unsuccessful, must begin in a state that satisfies the correctness assertion at node n. Similarly, let $fail(n)$ be the set of all behaviors that fail to reach a successor node of n, i.e., behaviors consisting of failed attempts of all applicable processes. In this case, an attempt may fail because it cannot satisfy the correctness assertion at node n, or because the applicable process instance itself fails.

The success and failure sets for a node n, denoted $S(n)$ and $F(n)$ respectively, are then defined recursively as follows: [4]

1. If n is a final node, then $S(n)$ is the set of states satisfying the correctness assertion at n and $F(n)$ is the set of states that fail to satisfy the correctness assertion at n.

2. If n has arcs a_i to nodes m_i, $1 \leq i \leq k$, then
   $S(n) = \bigcup_i succ(n,a_i).S(m_i)$ and
   $F(n) = \bigcup\{fail(n), \bigcup_i succ(n,a_i).F(m_i)\}$

The success and failure sets of a process description P are then taken to be the success and failure sets, respectively, of the initial node of P. The semantics of P is that any behavior in the success set of P satisfies the effects of P.

As an example, consider the process networks shown in Figure A.2 where the arcs are labeled with applicable process instances. For a process instance P, let $\langle P \rangle$ denote

[3] The decision to try each process instance at most once allows us to realize the control constructs of standard programming languages; various alternatives are possible without substantially affecting the results presented here.

[4] If $w_1 = s_1, \ldots s_k$ and $w_2 = s_k, \ldots s_n$, then $w_1.w_2 = s_1, \ldots s_{k-1}$, $s_k, s_{k+1}, \ldots s_n$. This operation is extended to sets of sequences in the usual way. Note that this formulation allows a single state to satisfy a sequence of goals.

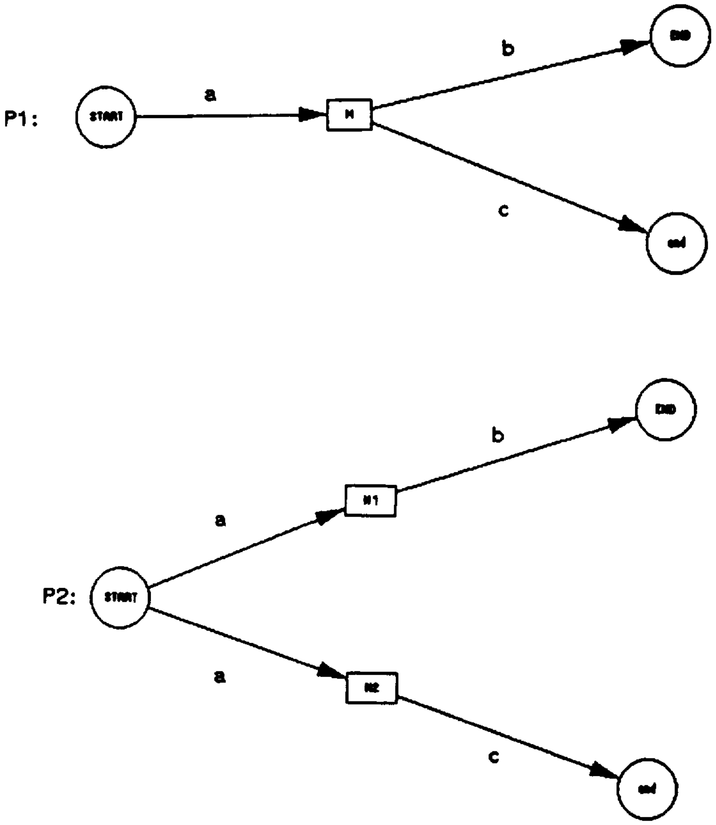


Figure A.2: Sample Process Networks

the set of its successful behaviors, and $\langle P\rangle_F$ the set of its failed behaviors. Then the success and failure sets for each of the process networks in Figure A.2 are as follows:

$\langle P1\rangle$:	$\langle a\rangle\ \langle b\rangle$	$\langle P1\rangle_F$:	$\langle a\rangle_F$
	$\langle a\rangle\ \langle c\rangle$		$\langle a\rangle\ \langle b\rangle_F\ \langle c\rangle_F$
	$\langle a\rangle\ \langle b\rangle_F\ \langle c\rangle$		$\langle a\rangle\ \langle c\rangle_F\ \langle b\rangle_F$
	$\langle a\rangle\ \langle c\rangle_F\ \langle b\rangle$		
$\langle P2\rangle$:	$\langle a\rangle\ \langle b\rangle$	$\langle P2\rangle_F$:	$\langle a\rangle_F$
	$\langle a\rangle\ \langle c\rangle$		$\langle a\rangle\ \langle b\rangle_F$
			$\langle a\rangle\ \langle c\rangle_F$

Notice that backtracking upon failure occurs only up to the current node being exited, and no farther.

Because process descriptions can be recursive, and because loops in process networks introduce self-reference into the definitions of S and F given above, a formal specification of the semantics of process descriptions requires a fixed-point construction. That is, for a given set of process instances $\mathcal{P} = P_1 \ldots P_n$, we need to define a transformation T that maps n-tuples of pairs of success and failure sets into additional such n-tuples. The definition of T is based on the definition of success and failure sets given above. If one assumes a set of primitive tests and actions, the least fixed point of T applied to these primitives can be taken as the denotation of $P_1 \ldots P_n$.

A.5 Operational Semantics

Process descriptions provide a way of describing the effects of actions in some dynamic problem domain. But how can a system or "agent" *use* this knowledge to achieve its goals? That is, we currently have a knowledge representation that allows us to state certain properties about actions and what behaviors constitute what actions. We have not explained, however, how an agent's *wanting* something can provide a rationale for or *cause* an agent to *act* in a certain way. This is the basis of so-called *practical reasoning* [7].

One way to view the causal connection between reasoning and action is as an *interpreter* that takes knowledge about actions and goals as input and as a result performs certain acts in the world. An abstract representation of such an interpreter may be considered to be the *operational semantics* of the knowledge representation language.

If a system is to be able to achieve its goals, it must be able to bring about certain actions, and thus be able to affect the course of behavior. Thus, we assume a system with certain effector capabilities. The actions that the system can effect simply by *choosing* to do so will be called *primitive actions*. The system must also be able to sense the world to the extent of determining the success or failure of the primitive actions. In addition, we assume the system has sensor capabilities for detecting satisfaction of all correctness assertions.

The system tries to achieve its goals by applying the following interpreter to applicable process instances. The interpreter works by exploring paths from a given node n in a process description P in a depth-first manner. To transit an arc, it unifies the corresponding arc assertion with the effects of the set of all process descriptions, and executes those that unify, one at a time, until one terminates satisfactorily. If none of the matching processes terminate successfully, and all leaving arcs fail, the execution of P fails. At each node, we verify that the correctness assertion (c-assertion) is satisfied.

```
function successful (P n)
  if (is-end-node n) then
    if (satisfied (c-assertion n)) then
      return true
    else return false
  else
    arc-set := (outgoing-arcs n)
    pr-a-set := (processes-that-unify arc-set)
    do until (empty pr-a-set)
      if (not (satisfied (c-assertion n))) then
        return false
      pr-a := (randomly-delete pr-a-set)
      pr := (process pr-a)
      a := (arc pr-a)
      if (successful pr (start-node pr)) then
        return (successful P (terminating-node a))
    end-do
    return false
end-function
```

The function processes-that-unify takes a set of arcs and returns the set of processes that unify with some arc in the set, along with the specific arc with which each unifies. The functions process and arc select out the process instance and corresponding arc from each element of this set. The function randomly-delete selects an element from a set, destructively modifying the set as it does so. The order in which selections are made is called the *selection rule*. The function return returns from the enclosing function, not just the enclosing do. The initial system goal is represented by a process description with a single arc labeled with the goal.

Note that, if this theory were to form the basis of the reasoning capabilities of some real-world agent, we would probably want process descriptions to be invoked on the basis of particular facts becoming known as well as because particular goals have been established. A suitable organization for such a system would be to have a list of all applicable process descriptions - some goal-invoked and others fact-invoked - and at each stage of processing select one of these for execution. The above recursive implementation would have to be modified, but the semantics would remain essentially the same (see Section 3.5).

Of course, it is important that the operational and declarative semantics be consistent with each other. The declarative semantics defines a set of behaviors for each process instance. The operational semantics also defines a set of behaviors for each process instance, but this set depends on the selection rule utilized in the above algorithm. Let $\langle P \rangle_D$ be the set of successful behaviors for a process instance P as given by the declarative semantics, and let $\langle P \rangle_{O,R}$ be the set of successful behaviors for P as given by the operational semantics for selection rule R. It is not difficult to show that

$$\langle P \rangle_{O,R} \subset \langle P \rangle_D$$

This means that any behavior generated by the interpreter given above will satisfy the declarative semantics. However, the inclusion, in general, is strict. That is, the interpreter may not achieve some given goal even when, according to the declarative semantics, there exists a way to achieve it. But, assuming that all correctness assertions are directly testable, we do have the following:

> If a behavior s is in $\langle P \rangle_D$, there exists a selection rule R such that s is in $\langle P \rangle_{O,R}$.

This is the best one can really hope for when any particular selection may cause some possibly irreversible action. It means that, provided you are smart enough to choose the right selection rule, the above interpreter will achieve a goal if it is at all achievable. This highlights the importance of reasoning about the selection of applicable processes in any practical implementation (see Section 3.3). It also means that one can reliably *plan* to achieve goals and be guaranteed of finding a finite plan if one exists.

A.6 Action Descriptions

So far, action descriptions have been restricted to simple action predicates. However, it is desirable to also allow a class of action descriptions that relate to conditions on world states.

We thus extend the action description language to include actions that achieve a given world state p (represented as $!p$), actions to test for p ($?p$), and actions that preserve p ($\#p$). We define these action descriptions more formally as follows.

We assume a fixed domain D and a fixed interpretation for constant symbols. Let w be a state assertion, σ an action description of the above form, and $S = s_1, \ldots s_n$ a behavior. Assume fixed global and local variable assignments and let all local interpretations I be consistent with them. We then have the following truth rules:

1. $!w$ is true in S if, for some local interpretation I, w is true in s_n.
2. $?w$ is true in S if, for some local interpretation I, w is true in s_1.
3. $\#w$ is true in S if, for all i, $1 \leq i \leq n$, there exist local interpretations I_i such that w is true of all states in S or $\neg w$ is true in all states in S.

To make effective use of such action descriptions we can use proof rules of the kind given below. We will use the notation $\langle P \rangle$ (σ) to mean that *every* successful behaviour associated with the process description P satisfies the temporal assertion σ. $\langle P \rangle_F$ denotes failed behaviors. The symbols ";" and "|" represent sequential composition and [nondeterministic] branching, 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

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

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 processes, the rules given above prove to be most practical.

The declarative semantics with this extension to the language is standard. The operational semantics simply requires that the interpreter be modified to allow application of the proof rules when necessary.

A.7 Conclusions

This paper has presented a simple model for action and a means for representing knowledge about procedures. We have indicated the importance of reasoning about *processes* rather than simply histories or state sequences. A declarative semantics for the representation was provided that allows a user to specify *facts* about behaviors independently of context. We have also given an operational semantics that shows *how* these facts can be used by an agent to achieve (or form intentions to achieve) its goals.

This knowledge representation can also be used for planning. Indeed, the operators of many standard planning systems (such as NOAH [45], DEVISER [52] and SIPE [53]) can be viewed as restricted forms of process descriptions. The fact that any behavior allowed by the declarative semantics can also be found using the operational semantics means that a planning algorithm that tried all possible selection rules would be "complete" - that is, it would find a solution if one existed.

By modifying the formalism so that failure sets allow full backtracking, single-state theorem proving of Horn clauses becomes a special case. This modification would also include as a special case the realization of "backtracking through triangle tables," as proposed by Nilsson [39]. However, such modifications present practical problems of verification and efficiency, and would appear useful only in some special cases.

In some ways, the declarative semantics is surprisingly complex and would seem to indicate some undesirable properties of the representation. Most of these difficulties arise from the need to model failed as well as successful behaviors. Of course, if we could fully specify necessary correctness conditions independently of context, *and* test for them, failed behaviors would become irrelevant for practical reasoning; we could always test to make sure conditions were true when needed. But experience with programming languages, and indeed the real world, shows that this can often be impractical if not impossible.

The formalism presented here can also be viewed as an *executable specification language* — that is, as a programming language that allows a user to directly describe the behaviors desired of the system being constructed. The fact that the language has a denotational semantics allows *facts* about the behavior of the system to be independently stated and verified. The operational semantics provides a means for directly *executing* these specifications to obtain the desired behavior. In this sense the language has much in common with Prolog, except that it applies to dynamic domains instead of static domains.

The system modeled in this paper has no data base, and thus no storage for knowledge or beliefs. We have a practical implementation of a system that includes such a data base, but have yet to formalize it. This introduces all the standard planning issues, such as the frame problem [31] and consistency maintenance [8]. We also need to investigate concurrency, and extend the model to deal with it. The notion of process failure and correctness assertions play a particularly important part when multiple agents or dynamic environments are allowed, and bear some relationship to formalisms for concurrent program verification. Some work in this direction is described by Georgeff [19].

Appendix B

A Theory of Process

This chapter describes the theoretical foundations of the process representation. A version of this material, entitled "A Theory of Process," was published in the *Proceedings of the Workshop on Distributed Artificial Intelligence*, held at Sea Ranch, California, in December, 1985.

The notion of *process* is essential for reasoning about the behavior of agents in dynamic worlds. The purpose of this work is to show why reasoning about process is so important, and to contrast this with other approaches in artificial intelligence (AI) that are based primarily on the allowable behaviors of agents. A model of events is constructed that provides for simultaneous action, and a model-based *law of persistence* is introduced to describe how events affect the world. No frame axioms or syntactic frame rules are involved in the specification of any given event, thus allowing a proper model-theoretic semantics for the representation. It is indicated how an algebra of processes can be employed to ascertain critical properties of multiagent systems, such as freedom from deadlock, and how systems of processes can be reasoned about given specifications of the component processes. A notion of hidden (internal) events is then introduced, whereupon it is shown how this provides an abstraction capability that can be used to avoid the combinatorial explosion typical of other AI approaches to multiagent planning. Finally, it is shown how the law of persistence, together with notions of causality and derived predication, makes it possible to avoid most of the difficulties associated with the frame problem.

B.1 Introduction

We take the notion of *compositionality* as central to any attempt at reasoning about complex systems. By that we mean that we should be able to compose complex systems from sets of interacting subsystems and have the behaviors of the latter determine the behavior of the whole. Moreover, subsystems that are deemed behaviorally equivalent should be replaceable by one another, without affecting the behavior of the whole.

Traditionally, machines or agents have been characterized by the set of behaviors that they can accept. Influenced by this tradition, formalisms for reasoning about multiagent domains have focused on reasoning about the *allowable behaviors* of agents [2,32]. For multiagent domains, however, sets of allowable behaviors are inadequate in their characterization of agents and hinder compositionality.

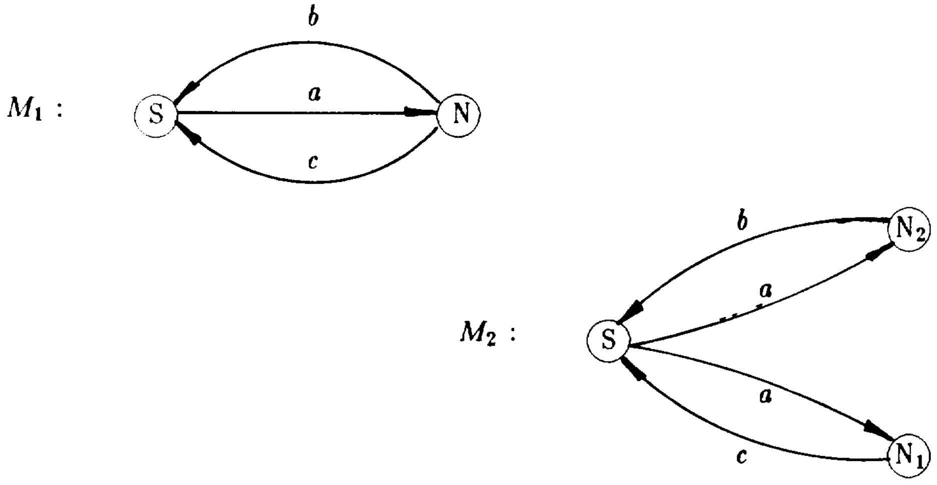


Figure B.1: Nonequivalent Machines

For example, consider the finite automata in Figure 1. Both accept the strings $(a(b+c))^*$. Consequently, the machines are deemed to be equivalent. But they are clearly not the same – they differ in the number of internal states and in the state transitions that are made. This in itself is no reason for viewing the machines as nonequivalent; only if their *observed behavior* is different do we wish to distinguish among machines.

From one point of view, however, their observed behavior *is* different. While the sets of strings that each machine *may* accept are identical, the sets of strings that each *must* accept are different. That is, machine M_2 could *fail* to accept some strings that M_1 would not. For example, after acceptance of an a, machine M_2 may fail to accept a b (if it is at node N_1) or a c (if it is at N_2), whereas M_1 cannot fail to accept either of these. Where concurrency is concerned, this means that the machines could act differently in different environments.

In other words, we want to view machines as equivalent if it is possible to replace one with the other when they appear as components of a larger system without affecting the overall system's behavior. Such a criterion has been called *observational equivalence* [36]. For the automata in Figure 1, however, it is easy to construct a system that is deadlock free when M_1 is used as a subsystem, but is subject to deadlock when M_1 is replaced by M_2.

We thus need something that retains more information about an agent than the agent's allowable behaviors. How we do this is the subject of Section 3. But first we need to define the basic entities in this ontology of agent behavior.

B.2 Events and Actions

We assume that, at any given instant, the world is in a particular *world state*. We consider a world state to consist of a number of *objects* from a given domain, together with various *relations* and *functions* over those objects. We will use predicate calculus for specifying world states, allowing quantification and the usual logical connectives.

A given world state has no duration; the only way the passage of time can be observed is through some change of state. The world changes state by the occurrence of some *event*. The simplest kind is an *atomic event*. As in classical models of events and actions, a single *occurrence* of an atomic event is viewed as a transition from some single world state to another single world state. An atomic event *type* is a set of such occurrences – that is, a relation on world states.[1] In what follows, we shall use

[1] We represent the transition relation as a function from world states to sets of world states.

the terms "action" and "event" synonymously, and will simply refer to event types as events.

We begin by modelling an event as a set of state sequences, representing all possible occurrences of the event in all possible situations (see also [2,32]). In particular, an atomic event must include all possible state transitions, ***including those in which other events occur simultaneously with the given event.*** Consequently, the state transition function for an event must allow most world relations to change, as almost anything can happen in parallel. Thus, the state transition function places restrictions on those world relations that are directly affected by the event, but leaves most others to vary freely (depending upon what else is happening in the world). This is in direct contrast to the classical approach, which views an event as changing some world relations but leaving most others unaltered.

For example, consider a domain consisting of blocks A and B at possible locations 0 and 1. Assume a world relation that represents the location of each of the blocks, denoted loc. Consider two events, $move_A$, which has the effect of moving block A to location 1, and $move_B$, which has a similar effect on block B. Then the classical approach (e.g., see [40]) would model these actions as follows:[2]

$$\begin{aligned} move_A = \{ \; & \langle loc(A,1), loc(B,1)\rangle \rightarrow \langle loc(A,1), loc(B,1)\rangle \\ & \langle loc(A,0), loc(B,1)\rangle \rightarrow \langle loc(A,1), loc(B,1)\rangle \\ & \langle loc(A,1), loc(B,0)\rangle \rightarrow \langle loc(A,1), loc(B,0)\rangle \\ & \langle loc(A,0), loc(B,0)\rangle \rightarrow \langle loc(A,1), loc(B,0)\rangle \} \end{aligned}$$

and similarly for $move_B$.

Every instance (transition) of $move_A$ leaves the location of B unchanged, and similarly every instance of $move_B$ leaves the location of A unchanged. Consequently, it is impossible to compose these two actions to form one that represents the simultaneous performance of both $move_A$ and $move_B$, except by using some interleaving approximation.

In contrast, our model of these actions is:

[2] Indeed, many approaches use syntactic representations that may not even yield proper models for thesee actions.

$$\begin{array}{ll} move_A = \{ & \langle loc(A,1), loc(B,1)\rangle \rightarrow \{\langle loc(A,1), loc(B,1)\rangle, \langle loc(A,1), loc(B,0)\rangle\} \\ & \langle loc(A,0), loc(B,1)\rangle \rightarrow \{\langle loc(A,1), loc(B,1)\rangle, \langle loc(A,1), loc(B,0)\rangle\} \\ & \langle loc(A,1), loc(B,0)\rangle \rightarrow \{\langle loc(A,1), loc(B,1)\rangle, \langle loc(A,1), loc(B,0)\rangle\} \\ & \langle loc(A,0), loc(B,0)\rangle \rightarrow \{\langle loc(A,1), loc(B,1)\rangle, \langle loc(A,1), loc(B,0)\rangle\}\} \end{array}$$

and similarly for $move_B$.

This model represents all possible occurrences of the action, including its simultaneous execution with other actions. For example, if $move_A$ and $move_B$ are performed simultaneously, the resulting action will be the intersection of their possible behaviors:

$$\begin{array}{rll} move_A \| move_B & = & move_A \cap move_B \\ & = \{ & \langle loc(A,1), loc(B,1)\rangle \rightarrow \{\langle loc(A,1), loc(B,1)\rangle\} \\ & & \langle loc(A,0), loc(B,1)\rangle \rightarrow \{\langle loc(A,1), loc(B,1)\rangle\} \\ & & \langle loc(A,1), loc(B,0)\rangle \rightarrow \{\langle loc(A,1), loc(B,1)\rangle\} \\ & & \langle loc(A,0), loc(B,0)\rangle \rightarrow \{\langle loc(A,1), loc(B,1)\rangle\}\} \end{array}$$

Thus, to say that an event has taken place is therefore simply to put constraints on some world relations, and leave most others to vary freely.

We now need to develop a notation for specifying properties of atomic events. Given a sentence ϕ in the state description language, let $m(\phi)$ be the set of states that satisfy ϕ under some class of interpretations for the nonlogical symbols appearing in the language, and let $m(e)$ be the transition function associated with event e. (The formal details are not important for this paper.) We then define the construct $\{e\}\phi$ to mean the set of states that result in ϕ's being true after the occurrence of an event e. More formally, the meaning of $\{e\}\phi$ is given by

$$m(\{e\}\phi) = \{s \mid \forall t. t \in m(\phi) \equiv \langle s,t\rangle \in m(e)\}$$

The "if" part of the equivalence states that s must be able to pass to all states satisfying ϕ during the performance of e, and the "only if" part states that all e transitions from s will satisfy ϕ.[3]

[3] Note that the [ ] operator of dynamic logic includes just the "only if" part of the { } operator. It is defined as $m([e]\phi) = \{s \mid \forall t.\langle s,t\rangle \in I_e(e) \supset t \in m(\phi)\}$

Thus, if we have $\psi \supset \{e\}\phi$, the performance of event e in a state in which ψ holds will result in a state in which ϕ holds. Moreover, depending on which events, if any, occur simultaneously with e, every state satisfying ϕ can be reached from each state satisfying ψ. For simplicity, we shall write $\psi\{e\}\phi$ for $\psi \supset \{e\}\phi$. We shall call $\langle\psi, \phi\rangle$ a *precondition/postcondition pair* for e.[4] If ϕ is satisfiable, we say that e is *performable* in situation ψ. The set of states for which an event is performable is called the *domain* of the event.

For example, the event $move_A$ satisfies $true\{move_A\}loc(A, 1)$. As $true$ covers the entire domain of the action, $move_A$ is *completely* characterized by the precondition/postcondition pair $\langle true, loc(A, 1)\rangle$. Similarly, the paramaterized action $move_A(x)$ that moves A to location x would satisfy $true\{move_A(x)\}loc(A, x)$; it would be completely characterized by the precondition/postcondition pair $\langle true, loc(A, x)\rangle$.

Specifying the precondition/postcondition pairs of an event does not require a large number of frame axioms stating what relations the performance of the event leaves unchanged; the event, in and of itself, usually places no restrictions on the majority of world relations so that these do not have to appear in the specification. In contrast to the classical approach, we therefore need not introduce any *frame rule* [22] or STRIPS-like assumption [11] regarding the specification of events.

B.3 The Law of Persistence

We have been viewing atomic events as imposing certain constraints on the way the world changes while leaving other aspects of the situation free to vary as the environment chooses. That is, an event transition function describes all the potential changes of world state that could take place during the occurrence of an event. Which transition actually occurs in a given situation depends, in part, on the events that take place in the environment. However, we have not specified what happens if no environmental

[4] By varying ϕ over the powerset of world states, $\{e\}\phi$ induces an equivalence relation over initial states of the event e. We shall call the equivalence classes that are so induced the *initial state sets* of e. Each initial state set uniquely determines a *final state set*, which is the set of final states accessible from every state in the initial state set. The set of all such pairs for an event e is unique. Thus, there is a set of precondition/postcondition pairs for e that fully specifies e and is unique up to equivalence.

event occurs or, more specifically, if no environmental event occurs that affects some given world relation.

What we need is some notion of *persistence* that specifies that, in general, world relations only change when forced to [32]. For example, because the action $move_A$ defined in the previous section places no constraints on the location of B, we would not expect the location of B to change when $move_A$ was performed in isolation from other environmental events.

At this point we encounter a serious deficiency in the event model we have been using and, incidentally, in all others that represent events as the set of all their possible behaviors (e.g., [2,32]) . Consider, for example, a world consisting of a source of light, L, the location of an object, A, and the location of the shadow cast by the object, S. We shall assume there are no other entities in the world, and that the possible locations for both L and A are 0 and 1. In our simplified world, the constraint on locations of these various objects is $loc(S) = 2 \times loc(A) - loc(L)$ (so that S has more possible locations than A and L).[5]

Now consider an action $move_A$ that moves A to location 1 (see Figure 2). The transition function for $move_A$ will map every possible initial state into the two final states $\langle loc(A,1), loc(L,1), loc(S,1)\rangle$ and $\langle loc(A,1), loc(L,0), loc(S,2)\rangle$.

If A is initially at location 0, the effect of $move_A$, in addition to changing the location of A, will be to change either the location of L or the location of S. The question is, if no other event occurs simultaneously with $move_A$, *which* of L and S changes location? While intuitively we would expect the location of S to change, using our current event model there is simply no way to distinguish these two possibilities. We cannot restrict the transition relation so that the first state above can only be achieved when the initial location of L is 1 (and the second state when L's location is 0), because that would prevent L from being moved simultaneously with A. Furthermore, the constraint on locations is a contingent fact about the world, not an analytic one – thus, we cannot sensibly escape the dilema by considering any of the relations *derived* from the others.

From a purely behavioral point of view, and with no additional knowledge about the nature of shadows, this is how things should be. If an observer had no sensors for

[5] We should really write something like $\forall x, y, z. loc(A,x) \wedge loc(L,y) \wedge loc(S,z) \supset z = (2 \times x - y)$. However, we will use the functional form here and throughout for simplicity of expression.

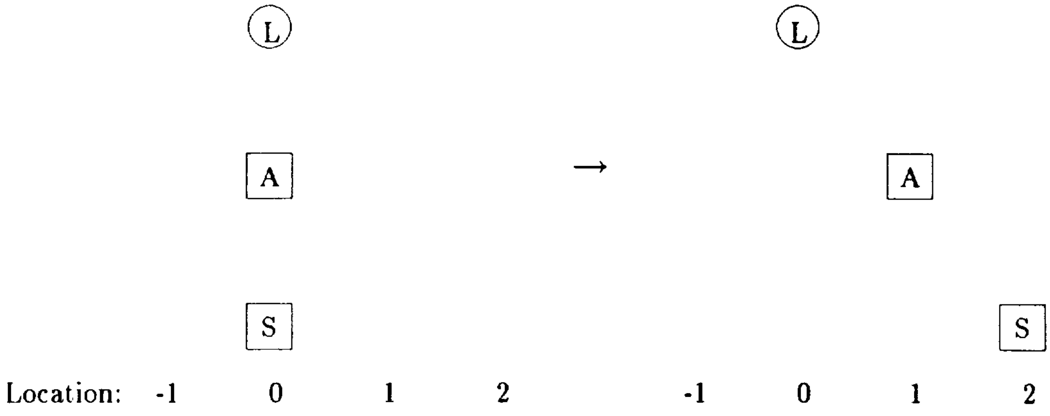


Figure B.2: Sample Transition for $move_A$

detecting the location of A, but could observe the location of L and S, then it would appear to such an observer that $move_A$ sometimes changed $loc(S)$ (whenever the location of L was not changed simultaneously with $move_A$) and sometimes changed $loc(L)$ (whenever the location of L was changed simultaneously). As there is no *observation* that could allow the observer to detect whether or not another action was occurring simultaneously, there is no way the two situations could be distinguished.

On the other hand, one might want to *choose* an ontology where these cases were distinguished, and be able to say that if $loc(L)$ were observed to change, there must have occurred some simultaneous event that changed it. Although it is not clear that being able to make these distinctions provides any more deductive power, such an ontology better matches commonsense reasoning.

In addition to problems such as these, there are also some technical reasons for finding our current representation of events unsatisfactory. (We shall encounter some of these problems later.) Therefore, we extend our model of an atomic event to include, in addition to the state transition function, a component specifying the relational tuples that *might be directly affected* by the event. For example, in the above case we could view $loc(A)$ and $loc(S)$ as the only relational tuples directly affected by $move_A$. We call these tuples the *direct effects* of the event.

Note that all the direct effects of an event e need not be involved in any single occurrence of e – they represent only *possible* effects. Also, the direct effects of an event do not define the possible state transitions - this is given, as before, by the state transition function for e.

It is important to be clear about the consequences of this extended model of atomic events. In particular, it means that events with the *same* transition function - that is, events that are indistinguishable through observation of all their possible behaviors - may *not* be identical. For example, consider the atomic event $move_A$ with the transition function given above and direct effects $loc(A)$ and $loc(S)$, and the atomic event $move_{AL}$ with the same transition function but with direct effects $loc(A)$, $loc(S)$, *and* $loc(L)$. (The action $move_{AL}$ can be viewed as the action that changes the location of A to 1 and arbitrarily chooses whether or not to move L.) The behaviors exhibited by both these actions are identical, yet we consider the actions to be different. The difference is essentially a matter of viewpoint - $move_A$ can change the location of S, but any change in the location of L is attributed to the simultaneous occurrence of some other event that affects L; on the other hand, $move_{AL}$, *in and of itself*, could change the location of L as well as the locations of A and S.

With this representation, we can now state how atomic events may be combined to form more complex atomic events. The two means of composition are simultaneity of events, denoted $\|$, and choice between events, denoted $+$. In both cases, the set of direct effects of the composite event is the union of the direct effects of the component events. The state transition relation, denoted tr, is given as follows:

- $tr(e_1 \| e_2) = tr(e_1) \cap tr(e_2)$
- $tr(e_1 + e_2) = tr(e_1) \cup tr(e_2)$

We are now in a position to say how an event changes the world if no other events occur simultaneously (i.e., if the event occurs *in isolation*).

The Law of Persistence:

> If for an event e, r is not a direct effect of e and no event e' with direct effect r occurs simultaneously with e, then r remains unchanged by the occurrence of e.

In other words, unless a relation is the direct effect of some event, the relation will remain unchanged by that event. This law is crucial; without it we could deduce little about the world state resulting from the occurrence of a given event.

For example, if $move_A$ given above was performed in isolation, then the location of L would remain unaffected by the performance of the action. An action *no-op* that allowed all possible state transitions but had no direct effects would do nothing when performed in isolation; an action *chaos* having the same transition function but with every relation a direct effect could do anything at all.[6]

To have well-defined final states in the presence of this law, the transition function and direct effects of an atomic event must satisfy the following compatibility requirement:

> Given an atomic event e with transition function tr, for every state s in the domain of e there is a state $s' \in tr(s)$ that preserves all the relational tuples of s that are not direct effects of e.

In our model, therefore, the effects of events can be ascertained without recourse to any frame axioms (stating what remains unchanged by the occurrence of an event) or any syntactic frame rules (such as a STRIPS-like assumption). Indeed, to fully specify an atomic event, all we need is a specification of the precondition/postcondition pairs of the event and its direct effects.

B.4 Processes

As we observed in the introduction, for multiagent domains it is not sufficient to represent the behaviors of agents simply by giving the sequences of events that they *can* engage in. We must also specify what sequences of events can lead to failure. To do this we introduce the notion of *process*.

An agent may be thought of as being able to perform a number of different actions. In a multiagent domain, or wherever any interaction with an environment may occur, certain of these actions will be constrained to occur simultaneously with other events. For example, if an agent has its hand on a light switch, the action of moving its hand upward occurs simultaneously with the movement of the light switch, which in turn

[6] Consider a two state domain with possible states p and $\neg p$. If e_1 is an event that brings about p and e_2 is an event that brings about $\neg p$, then $e_1 + e_2$ is *chaos*. This is as expected – as we have a choice of doing e_1 or e_2 anything at all could happen. However, without the extended model of events that we described above, there would be no way to distinguish $e_1 + e_2$ from *no-op*.

occurs simultaneously with a flow of electricity. Thus, we need to specify which actions an agent can perform and which of these must occur simultaneously with other events. If two atomic events e_1, e_2 are constrained to occur simultaneously, we shall say that they are ***synchronous***, denoted $e_1 \Leftrightarrow e_2$. The event in which both e_1 and e_2 occur simultaneously is $e_1 \| e_2$. We shall assume that if $e_1 \Leftrightarrow e_2$, then $e_1 \| e_2$ is performable.

A ***process*** can be viewed as ***generating*** all possible behaviors of an agent in all possible environments. A ***renewal behavior*** of a process is the behavior of the agent after some action has been performed. We write $P \rightarrow_e Q$ to mean that process P can evolve into process Q after performing the atomic action e.

Two processes have no renewals. These are $FAIL$, which represents the failure of a process, and NIL, which represents successful completion of a process. These are the simplest forms of process, upon which all others are built by means of various composition operators.

The composition operators we introduce below are based on the synchronized behavior algebras of Milne [35]. (Other alternatives (e.g., [26]) could equally well be employed.)

B.4.1 Prefixing (:)

The process $e : P$ is one that can begin by performing the event e, after which it behaves exactly like P. We therefore have the rule

- $e : P \rightarrow_e P$

B.4.2 Sequencing (;)

The process $P \; ; \; Q$ behaves first like P and, if that concludes successfully, behaves next like Q. We thus have

- If $P \rightarrow_e P'$ then $P \; ; \; Q \rightarrow_e P' \; ; \; Q$

- If $P \rightarrow_e NIL$ then $P \; ; \; Q \rightarrow_e Q$

B.4.3 Ambiguity (+)

The process $P + Q$ can behave like either P or Q. We have

- If $P \rightarrow_e P'$ then $P + Q \rightarrow_e P'$
- If $Q \rightarrow_e Q'$ then $P + Q \rightarrow_e Q'$

For example, if

$R = (b : P) + (c : FAIL)$,

then R can either perform b and evolve into P or perform c and fail.

The ambiguity operator allows the environment to choose between events, whereas prefixing allows no such choice. This may be viewed as the difference between events initiated by the environment and those initiated by the agent. Note that $(e_1 : P)+ (e_2 : P)$ is equivalent to $(e_1 + e_2) : P$, which is why we have chosen to use the + symbol for both processes and events.

B.4.4 Parallelism (&)

The process $P \& Q$ involves both prOcesses P and Q running concurrently. Events that are designated as synchronous must occur simultaneously, whereas other events can choose to occur simultaneously with one another or be arbitrarily interleaved. We therefore have

- If e is not synchronous with any event in Q and $P \rightarrow_e P'$ then $P \ \& \ Q \rightarrow_e P' \ \& \ Q$
- If e is not synchronous with any event in P and $Q \rightarrow_e Q'$ then $P \ \& \ Q \rightarrow_e P \ \& \ Q'$
- If b and c can (or must) be performed simultaneously and $P \rightarrow_b P'$, $Q \rightarrow_c Q'$ then $P \ \& \ Q \rightarrow_{b\|c} P' \ \& \ Q'$

The operators given above are not all we could define, but they are sufficient for our present purposes. Indeed, the machinery we now have is much more powerful than that described in any other work on multiagent planning, with the exception of some new research by Lansky [28] and Rosenschein and Kaelbling [44].

Using results from concurrency theory (e.g., [26,35]), various algebraic laws for reasoning about these processes can be developed [21]. These laws allow us to discover certain properties of systems of processes, such as system deadlock. For example, using slightly different process models from that presented herein, the author [17] and Stuart [49] have applied such laws to synthesize synchronized multiagent plans. However, because we do not require events of the same type to be synchronized, and because a given relation can be affected by different events, we cannot utilize many of the specification axioms developed in concurrency theory. In the following sections, axioms appropriate to our model of concurrency are provided.

B.5 State Change Axioms

Our goal is to use descriptions of events and actions given in terms of precondition/postcondition pairs to derive constraints on possible event orderings.[7] Furthermore, we wish to derive rules that allow us to determine the behavior of systems of processes, given properties of the behaviors of the component processes.

For reasoning about world states, the axioms need to include the standard axioms of predicate calculus. To these we must add other axioms or rules for reasoning about events and processes. While the rules given below are probably not complete, they are nevertheless sound.

We have a particularly simple rule for reasoning about simultaneous actions:[8]

- If $\psi_1\{e_1\}\phi_1$ and $\psi_2\{e_2\}\phi_2$ then $(\psi_1 \wedge \psi_2)\{e_1 \| e_2\}(\phi_1 \wedge \phi_2)$

We also have a simple but somewhat surprising rule for the choice operation:

- If $\psi_1\{e_1\}\phi_1$ and $\psi_2\{e_2\}\phi_2$ then $(\psi_1 \wedge \psi_2)\{e_1 + e_2\}(\phi_1 \vee \phi_2)$

[7] It is constraints such as these that Lansky [28] uses for her plan synthesis system. However, she requires that they be specified for the problem domain under consideration; she does not attempt to derive them from a state-based representations for events. Indeed, she takes the opposite approach and *defines* state predicates in terms of restrictions upon event sequences.

[8] Note that the notation of dynamic logic does not immediately lend itself to such simple axioms; for example, if $\psi_1 \supset [e_1]\phi_1$ and $\psi_2 \supset [e_2]\phi_2$, and $e_1 \| e_2$ is taken to represent the intersection of e_1 and e_2, it *does not follow* that $\psi_1 \wedge \psi_2 \supset [e_1 \| e_2]\phi_1 \wedge \phi_2$.

In what follows, we shall use a temporal logic consisting of the modal operators $\Box$ (always) and $\Diamond$ (eventually) to describe behaviors. If every behavior generated by a process P satisfies a temporal assertion ϕ, we shall write $\langle P\rangle\phi$. If ψ is a nontemporal state description (i.e., contains no temporal operators), we will also write $\psi\langle P\rangle\phi$ to mean $\langle P\rangle(\psi \supset \phi)$. Thus, for example, if ψ is a nontemporal state description, $\psi\ \langle P\rangle \Diamond\Box\phi$ means that, if the first state in a behavior of P satisfies ψ, then eventually ϕ will always be satisfied – in particular, if ϕ is also nontemporal, the last state, if one exists, will satisfy ϕ. Note that this formalism does not allow us to make statements about what *might* happen in a process – we can only state facts about what *must* happen.

Unfortunately, because the set of behaviors associated with a process include *all possible* behaviors, there is little one can say about the effects of individual processes if the environment is allowed to change the world in arbitrary ways. We therefore distinguish between events that occur within the process and events that occur in the environment, and qualify our axioms by placing constraints on what behaviors may or may not occur in the environment (see also [4]).

To do this, we shall qualify process axioms by annotating prefix operators with $[\psi]$ wherever ψ is a constraint on behaviors in the environment. That is, for an event e and process P, the intended meaning of $e : [\psi]\ P$ is that ψ must be true of the sequence of states following the completion of e until the beginning of P. Sequencing operators can be annotated similarly.

We begin with the following general laws:

- If $\langle P\rangle\psi$ and $\langle P\rangle\phi$ then $\langle P\rangle(\psi \wedge \phi)$

  (This extends in the obvious way to universal quantification.)

- If $\langle P\rangle\psi$ and $\psi \supset \phi$ then $\langle P\rangle\phi$

- $\langle FAIL\rangle true$

B.5.1 Prefixing (:)

For a process of the form $e : P$, one might at first expect that the final states of the event e should at least intersect the possible initial states of the process P, if not be a subset of them. However, this is not so, because the environment can intervene after the performance of e and change the state of the world before P begins. We therefore have

- If $\theta\{e\}\gamma$ and $\psi\langle P\rangle\phi$ then $\theta\ \langle e : [\gamma \wedge \Diamond\Box\psi)]\ P\rangle\ \Diamond\phi$

This says that ϕ will eventually be satisfied by $e : P$ if, after e has been performed, the environment can bring about ψ starting from an initial state in which γ is true, and can keep ψ true long enough for process P to begin. Note that, even if e can directly bring about ψ (i.e., if $\gamma \supset \psi$), the environmental constraint is necessary to prevent the environment from interfering detrimentally and undoing the effects of e.

Similar rules are used by Lansky [28] for defining state predicates, and by Pednault [40] and Chapman [5] for plan synthesis. In these works, there is some rule or axiom allowing an event that deletes a desired precondition but requiring that the latter be reestablished by some subsequent event.

B.5.2 Sequencing (;)

The rule for sequencing is very similar to that for prefixing; in this case, the environment can intervene between the completion of one process and the start of the next. We have

- If $\psi_1\langle P_1\rangle\phi_1$ and $\psi_2\langle P_2\rangle\phi_2$ then $\psi_1\ \langle P_1\ ;\ [\phi_1 \wedge \Diamond\Box\psi_2)]\ P_2\rangle\ \Diamond\phi_2$

It is important to note that the sequencing operators defined by both Allen [2] and McDermott [32] do not permit the interleaving of events between the component processes. This is much too strong a restriction on sequencing; for example, the standard sequencing operators of programming languages that support concurrency cannot then be modeled.

B.5.3 Ambiguity (+)

In this case, we simply require that either of the specifications for P_1 and P_2 be satisfied:

- if $\langle P_1\rangle\phi_1$ and $\langle P_2\rangle\phi_2$ then $\langle P_1 + P_2\rangle(\phi_1 \vee \phi_2)$

B.5.4 Parallelism (&)

Because parallelism can be reduced to prefixing (:) and ambiguity (+) by means of certain algebraic laws (see [21]), there is no need for any axioms about the state behaviors of parallel processes. Nevertheless, the following rule is very convenient, as it avoids examining all interleavings of the composed processes:

- if $\langle P_1 \rangle \phi_1$ and $\langle P_2 \rangle \phi_2$ then $\langle P_1 \ \& \ P_2 \rangle (\phi_1 \wedge \phi_2)$

This rule assumes that each component process satisfies the environmental constraints of the other. The conditions for *interference freedom* described in a previous paper [19] represent a particular case in which these constraints are satisfied. If any events are constrained to occur synchronously, the combined process may fail. However, any behavior that does not fail will satisfy the foregoing rule.

The rules given above enable us to infer properties of systems from the properties of their components. For example, consider the following. Two people are engaged in lifting a table. We assume that the world can be in any state $\langle l, r \rangle$, where l and r are integers representing the height of the left and right ends of the table, respectively. The initial world state is $\langle 0, 0 \rangle$, and the goal is to raise the table more or less evenly, i.e., so that $-1 \leq l - r \leq 1$. Each person can perform two atomic events:

Person 1:

- A test t_1 where $(l \leq r)\{t_1\}(l \leq r)$ and $(l > r)\{t_1\}$*false*
- An action a_1 where $(l = n)\{a_1\}(l = n + 1)$

Person 2:

- A test t_2 where $(r \leq l)\{t_2\}(r \leq l)$ and $(r > l)\{t_1\}$*false*
- An action a_2 where $(r = n)\{a_2\}(r = n + 1)$

The processes representing each person are[9]

$P_1 = t_1 : a_1 : P_1$ and

$P_2 = t_2 : a_2 : P_2$

[9] Strictly speaking, we need a fixed-point operator to define these processes (see [26]).

Let q_1 represent any sequence of states in which l does not increase and r does not decrease (i.e., $q_1 \equiv (l - r = n) \supset \Box(l - r \leq n)$), and let q_2 represent any sequence of states in which r does not increase and l does not decrease (i.e., $q_2 \equiv (n \leq l - r) \supset \Box(n \leq l - r)$. The behaviors generated by these processes satisfy the following:

Process 1: $(l - r \leq 1)\ \langle t_1 : [q_1]\ a_1 : [q_1]\ P_1 \rangle\ \Box(l - r \leq 1)$

Process 2: $(-1 \leq l - r)\ \langle t_2 : [q_2]\ a_2 : [q_2]\ P_2 \rangle\ \Box(-1 \leq l - r)$

Furthermore, neither process violates the environmental constraints of the other (in fact, they are *interference free* [19]). Therefore, we must have

$$(-1 \leq l - r \leq 1)\ \langle P_1\ \&\ P_2 \rangle\ \Box(-1 \leq l - r \leq 1)$$

Of course, this relies on the external environment satisfying the environmental constraints specified in each of the process descriptions.

To show that the above two processes actually enable the table to be raised, we need to prove that each "cycle" of each process increases l and r, respectively, and that at least one process can always proceed. This is straightforward.

We now need to consider how to conceal the internal structure of processes so that we have the capability of abstraction.

B.6 Internalization

For a given process, certain relational tuples will be nonobservable (and thus incapable of being influenced) from outside that process. We call these *internal relations*. In addition, some world relations will be nonobservable from inside the process. We call these *external relations*. The remaining relations are called *interface relations*.[10]

The external relations of a given process (agent) are those relations that are not directly affected by any event in a process *and* that do not occur critically in any precondition of any event in the process. Knowledge of the external relations for every

[10] Although we talk of "relations", we strictly mean relational tuples. Thus, for example, the tuples $On(x, y)$, for all x and y in a given room, may be external to processes operating outside the room, but the relation On applied to other objects may not be external to these processes.

process in a system of processes fully determines the internal and interface relations for each such process.

Let us first consider the externally observable behavior of processes. For a given world state w over a domain involving relations R, let $w|_R$ be the restriction of w to the set of relations in R. We extend this operation to atomic events in the natural way: the direct effects are restricted to R and the transition relation is given by

$$tr(e|_R) = \{\langle s|_R, t|_R\rangle | \langle s, t\rangle \in tr(e)\}$$

For a given process P with interface relations R, let $P|_R$ be the process in which all atomic events e are replaced by $e|_R$. The set of behaviors of $P|_R$ is equal to the set of *observable* behaviors of P. For processes P_1 and P_2 with interface relations R, if the set of behaviors of $P_1|_R$ equals the set of behaviors of $P_2|_R$, we say that P_1 is *observationally equivalent* to P_2 with respect to R.

Once we restrict the events in a process P to the interface relations R of P, many of the restricted events will turn out to be *no-ops* – that is, many of the events, restricted to R, will have no direct effects and no constraints on the allowed state transitions. If, in addition, these events are not constrained to be synchronous with any external event, they will be nonobservable from outside P. Such *hidden* events will be denoted by the symbol τ.

The algebraic laws relating to the special hidden event τ are given elsewhere [21]. These are critical in allowing the behavior of a system of processes to be ascertained without examining all interleavings of the component processes, and thus enable us to avoid computational intractability. In this way we can reason about processes without concerning ourselves with their internal behavior (i.e., how they are implemented).

The combination of internalization (τ) and ambiguity ($+$) also enable us to distinguish between *external* and *internal* nondeterminism. The distinction between these two kinds of nondeterminism is essential when dealing with real-world systems. For example, it allows us to differentiate between a machine that will dispense either a bag of candy or a chocolate bar, depending on what the user of the machine does, and a poker-playing slot machine that chooses for itself, independently of the user's will, what hand to display next.

The difference in the two cases is simply a matter of whether or not the selecting event is hidden from the external world. Consider, for example, the following two processes:

$EXTERNAL = (e_1 : P) + (e_2 : Q)$

$INTERNAL = (\tau : P) + (\tau : Q)$

In the first of these, the environment can influence the choice between P and Q by performing either e_1 (or some event constrained to occur simultaneously with e_1) or e_2; in the second process, the initial event is hidden and the environment thus has no influence over the subsequent behavior of the agent.

Now consider the internal behavior of processes. Because the environment cannot affect the internal relations of a process, we can strengthen considerably the axioms given in the preceding section. In particular, for a process $e : P$, any final state of e must satisfy the precondition of P with respect to the internal relations of P. As before, however, the enviroment can intervene between the completion of e and the beginning of P to change the interface relations.

We consider below the simplified rules for the various composition operators. Let R be the interface relations for a process P and let I be the internal relations of P. We then have

- if $\theta\{e\}\gamma$ and $\psi\langle P\rangle\phi$ and $(\gamma \supset \psi)|_I$ then θ $\langle e : [(\gamma \wedge \Diamond\Box\psi)|_R]\ P\rangle\ \Diamond\phi$

This rule states that, if the internal constraints of ψ are satisfied by the event e and the interface constraints of ψ are satisfied by the environment, then eventually ϕ will hold. In particular, if ψ does not contain any interface relations, we have

- If $\theta\{e\}\gamma$ and $\psi\langle P\rangle\phi$ and $\gamma \supset \psi$ then $\theta\langle e : P\rangle\Diamond\phi$

This is the standard rule used in most planning systems, in which one attempts to extend a plan (P) to accomplish a given goal (ϕ) by prefixing the plan with an action (e) that will achieve the current plan's preconditions (ψ). The law for sequencing is similar.

The laws for ambiguity and parallel composition are as before, but now we only have to check compatibility of the environmental constraints as restricted to the interface relations. Furthermore, as mentioned above, any reduction of parallelism to prefixing and ambiguity (as needed, for example, in the analysis of deadlock) is greatly simplified by the introduction of hidden events.

B.7 The Frame Problem and Causality

The frame problem, as Hayes [22] describes it, is dealt with in our approach by means of the law of persistence. Because this law is a property of our event model, and not of our event specification language, we thus avoid all of the semantic difficulties usually associated with the frame problem.

However, in our representation the specification of the direct effects of an event can be cumbersome. For example, if we have a domain with relations representing the location of objects, the distance between objects, the farthest object, and the closest object, then any event that changed the location of an object would have to include as direct effects each of the other relations.

To avoid this problem, we allow that certain predicates be specified as *derived* predicates [12]. We require that derived predicates be defined in terms of other predicates and that they be well-founded. We can then adopt the convention that any derived predicate need not be mentioned in the set of direct effects of an event, as it is always possible to determine its truth value in a given state by examining the truth value of its definiens. Of course, the law of persistence would not then apply to derived predicates.

For example, consider a domain consisting of two blocks A and B, and assume relations loc representing location and $dist_{AB}$ representing the distance between A and B. We could specify that distance was a derived predicate, defining it in terms of the locations of A and B. Then an action $move_A$ that changed the location of A would only need to mention $loc(A)$ as a direct effect of the action.

We could, if desired, extend the convention in the other direction and allow derived predicates to be mentioned as direct effects. In such situations, the definiens of the derived predicate are also to be considered direct effects of the event (and so on, recursively). For example, if the explicit direct effect of some event $stretch_{AB}$ was $dist_{AB}$, then we would consider the actual direct effects to be $dist_{AB}$, $loc(A)$ and $loc(B)$. That is, $stretch_{AB}$ could affect, in addition to $dist_{AB}$, the location either of A or B or both.

Another problem with specifying the direct effects of an event is that it requires considering *all* the relations and functions the event could possibly affect, in any conceivable circumstance. This does not conform well with commonsense views of action. For example, in the case of the block and the shadow (Section 3), the direct effects of the $move_A$ event included the location both of the block (A) and the shadow (S). As mentioned before, we do not want to consider the location of the shadow to be a derived predicate, but it does not seem we should have to mention it in the definition of the event that simply moves block A.

We handle this problem by introducing a notion of causality. What count as direct effects of an event can now be simplified, but certain events are forced to occur simultaneously that restrict the application of the law of persistence to the composite parallel event. That is, if an event e_1 is stated to *cause* an event e_2, we require that e_1 always occur simultaneously with e_2. The law of persistence could then be applied to the event $e_1 \| e_2$, but not to e_1 alone.

For example, in the above case we might specify that a simultaneous event ($move_S$, say) is caused by the movement of A that results in the location of S being changed. Thus, the direct effect of $move_A$ is simply $loc(A)$ – the change to S's location comes about through the direct effect of the event $move_S$ which is always constrained to be simultaneous with $move_A$. This conforms better to commonsense reasoning, where we would be inclined to say that moving A *caused* A's shadow to move, rather than considering the movement of A and the shadow to be a single event. Also note that, provided that the causal laws are stated correctly, they need not presuppose that no further event occurs simultaneously – for example, the laws should still apply even if the light source and the block were both moved at the same time.

The introduction of derived predicates does not provide any increase in representational power: they can always be replaced by their definiens. However, causal laws allow us to represent events that cannot be described by a single event specification. The reason is that event specifications can only include a finite number of direct effects, whereas with causality we can represent events with an infinite number of effects. However, the major reason for introducing causality is to allow for simpler event specifications and more natural reasoning about behaviors, rather than greater expressive power.

The notion of causality used by us is actually more general than that described above, and is fully described in another paper [21]. Essentially, we view causality as a relation between atomic events and processes that is conditional on the state of the world. We also relate causation to the temporal ordering of events, and assume that an event cannot cause a process (and thus other events) that precedes it. However, we do allow an event to cause another that occurs simultaneously (as in the above example). This differs from most formal models of causality [28,32,46].

At this point we could ask what has happened to the frame problem – in particular, why do most other formalisms that use a general rule for determining the relations preserved by an event base that rule on provability of formulae? The answer is twofold. First, our representation of events and the law of persistence are model-based rather than syntactic. Unlike most other approaches, this provides us with a proper model-theoretic semantics for our specification language.

Second, we have really shifted the place in which questions of provability arise. In the usual approaches, any action specification is guaranteed to be consistent, provided the axioms describing world states are consistent. However, determination of what conditions hold after performance of an action requires, in general, determining the consistency of a some given set of formulae. In our case, determining the effects of an event do not require determining the consistency of any formulae, but *the action specifications themselves* can be inconsistent. Thus, while using the usual frame rules it is undecidable what the effects of action are [41], in our case it is undecidable whether or not a given action specification is consistent. However, the latter is really no disadvantage at all: given that the consistency of the axioms describing world states is undecidable, it hardly matters that the consistency of the formulae describing actions is also undecidable.[11]

There are also important implementation considerations. Any approach where the effects of an action are dependent on determination of the consistency of formulae is simply intractable. In contrast, the approach outlined here can be implemented very efficiently, as the relations and functions that can be affected by the occurrence of an event require, at most, provability of the formulae of interest. Interestingly, one of the most efficient action representations so far employed in AI planning systems – the STRIPS representation – is essentially the special case in which (1) each action has a

[11] Had we adopted the representation of events that did not include specification of their direct effects, we would have encountered problems similar to those of the traditional approaches.

single precondition and postcondition, (2) the postcondition is a conjunction of literals, (3) the direct effects include all the literals mentioned in the postcondition, and (4) no events ever occur simultaneously with any other.

Some researchers take a more general view of the frame problem, seeing it as the problem of reasoning about actions and events in the presence of *incomplete information* about processes (usually the environment). Unfortunately, this problem is often confused with the representation of events, with the result that there is usually no clear model-theoretic semantics for the representation.

In our case, the problem of reasoning about processes and that of making assumptions about them are quite separate issues. That is, given a description of a set of processes, we can use the approach outlined above to determine certain properties of any composite system of these processes. If we want stronger results, we may need to make additional assumptions about the system. We may wish to assume, for example, that a certain relation r in a given process will not be influenced by other processes (i.e., that r is internal to the process). But making assumptions as to which other events are likely to occur or which relations are internal to given processes is quite a separate problem from that of reasoning on the basis of these assumptions. Indeed, we can (to some extent) qualify any statement about the effects of a given process, thus making any assumptions explicit.[12]

It is not our intention to consider the problem of making useful assumptions about events and internalization. However, the means of making such assumptions is likely to be very domain-dependent, and not something that can be sensibly encompassed in domain-independent rules based on minimal models or syntactic properties of the representation (as in most approaches to the frame problem). For example, it at first seems reasonable to assume that my car is still where I left it this morning, unless I have information that is inconsistent with that assumption. However, this assumption gets less and less reasonable as hours turn into days, weeks, months, years, and centuries. This puts the problem where it should be – in the area of making reasonable assumptions, not in the area of *defining* the effects of actions [11,22] or the persistency of facts [32].

[12] Much of the qualification problem revolves around the internalization of relations. We may, for example, internalize a relation about the connection of two components in a car, whereas, in reality, it is possible for this connection to be externally affected (or for there to be an unspecified internal event that affects the connection).

Appendix C

Sample Knowledge Base for the RCS system

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.

C.1 Glossary of Identifier Prefixes

Individual elements in the system are represented by unique identifiers. These are of the form $\langle word\rangle.\langle number\rangle.\langle number\rangle\ldots$, 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

C.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 system in its standard configuration (see Figure 4.2).

C.2.1 Top Level Reactant Control Systems

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

C.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)
```

C.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)
```

C.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)
```

C.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)
```

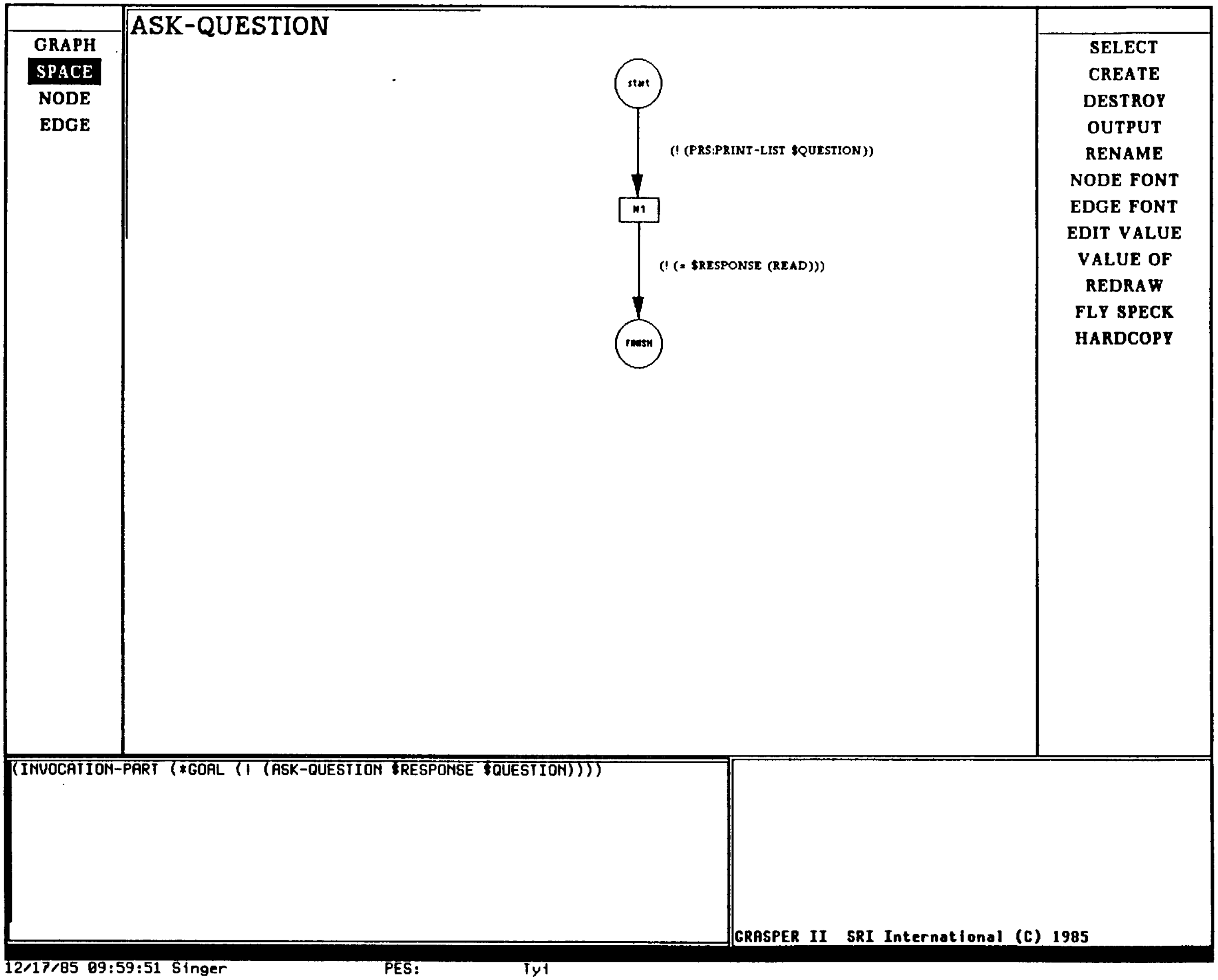
```
(PART-OF BIV.1.7.2 THR.1.7)

(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 RCS systems.

C.3 Knowledge Areas

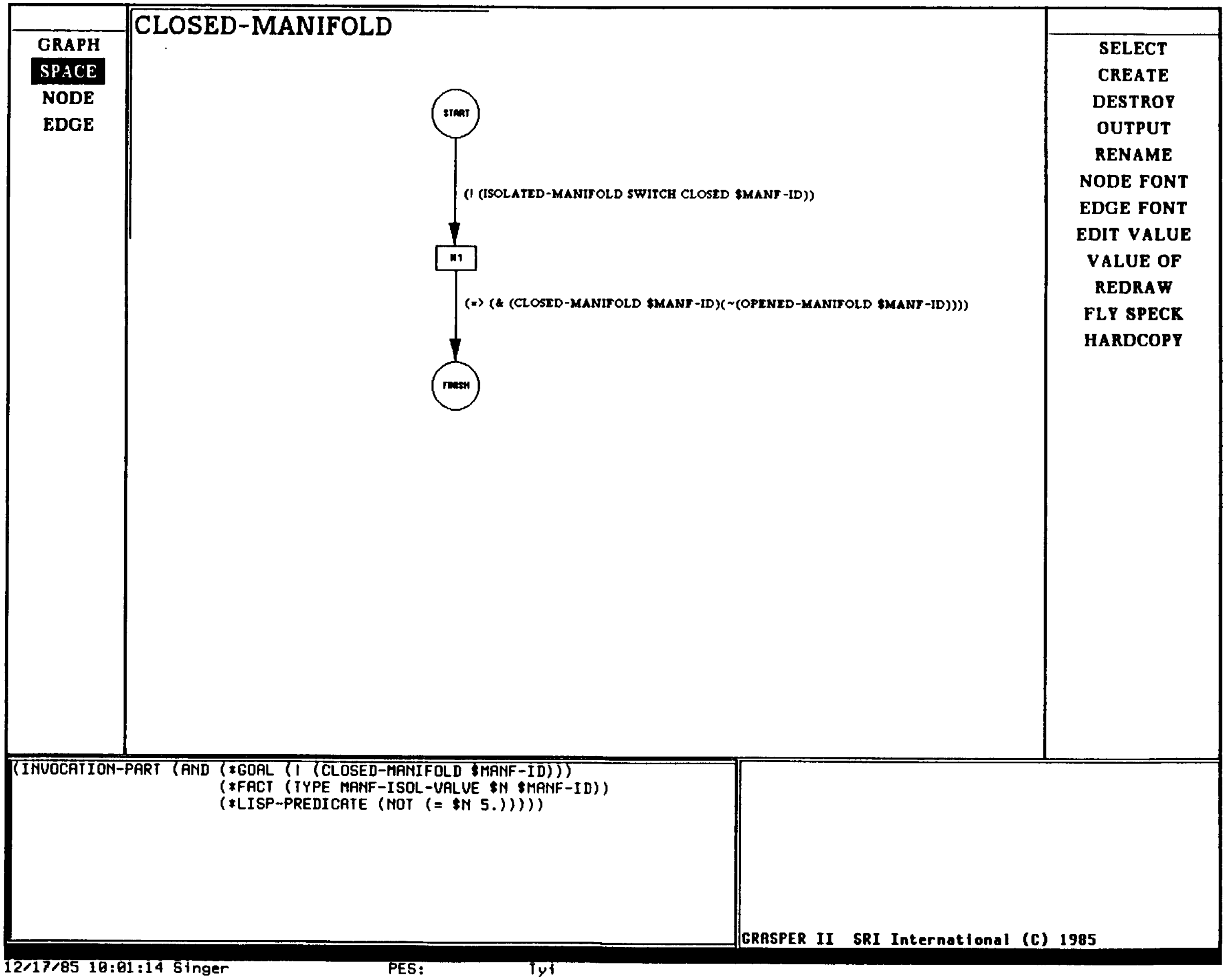
This section contains sample processes representing some of the RCS malfunction handling procedures (see Figures 1.1, 1.2 and 1.3). Note that the syntax varies slightly from that given in the main body of the report in that the metalevel predicates **`goal`** and `fact` are prefixed here with a * sign.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



```
(INVOCATION-PART (*GOAL (! (ASK-QUESTION $RESPONSE $QUESTION))))
```

GRASPER II SRI International (C) 1985

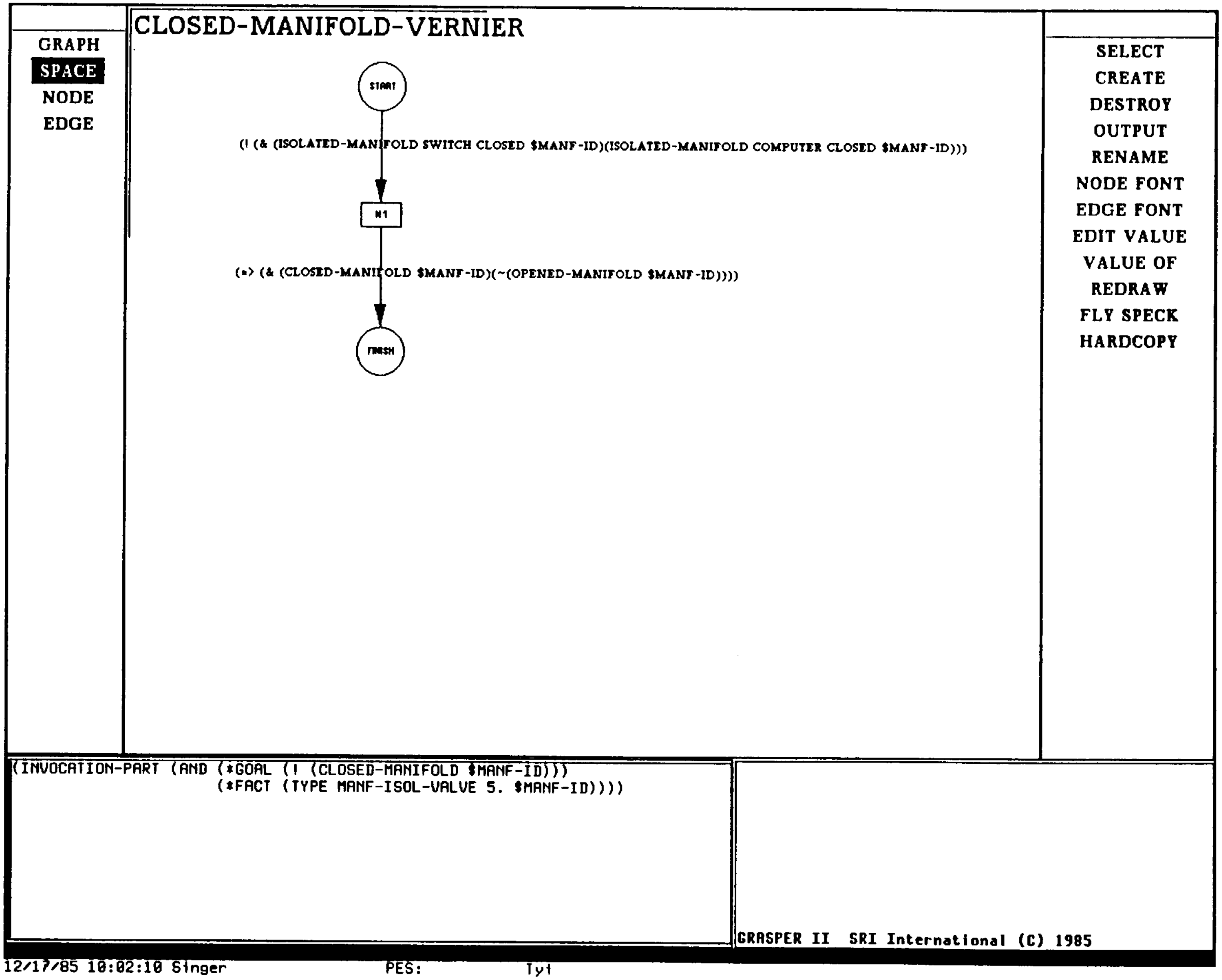
12/17/85 09:59:51 Singer   PES: Tyi



```
(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: Ty!

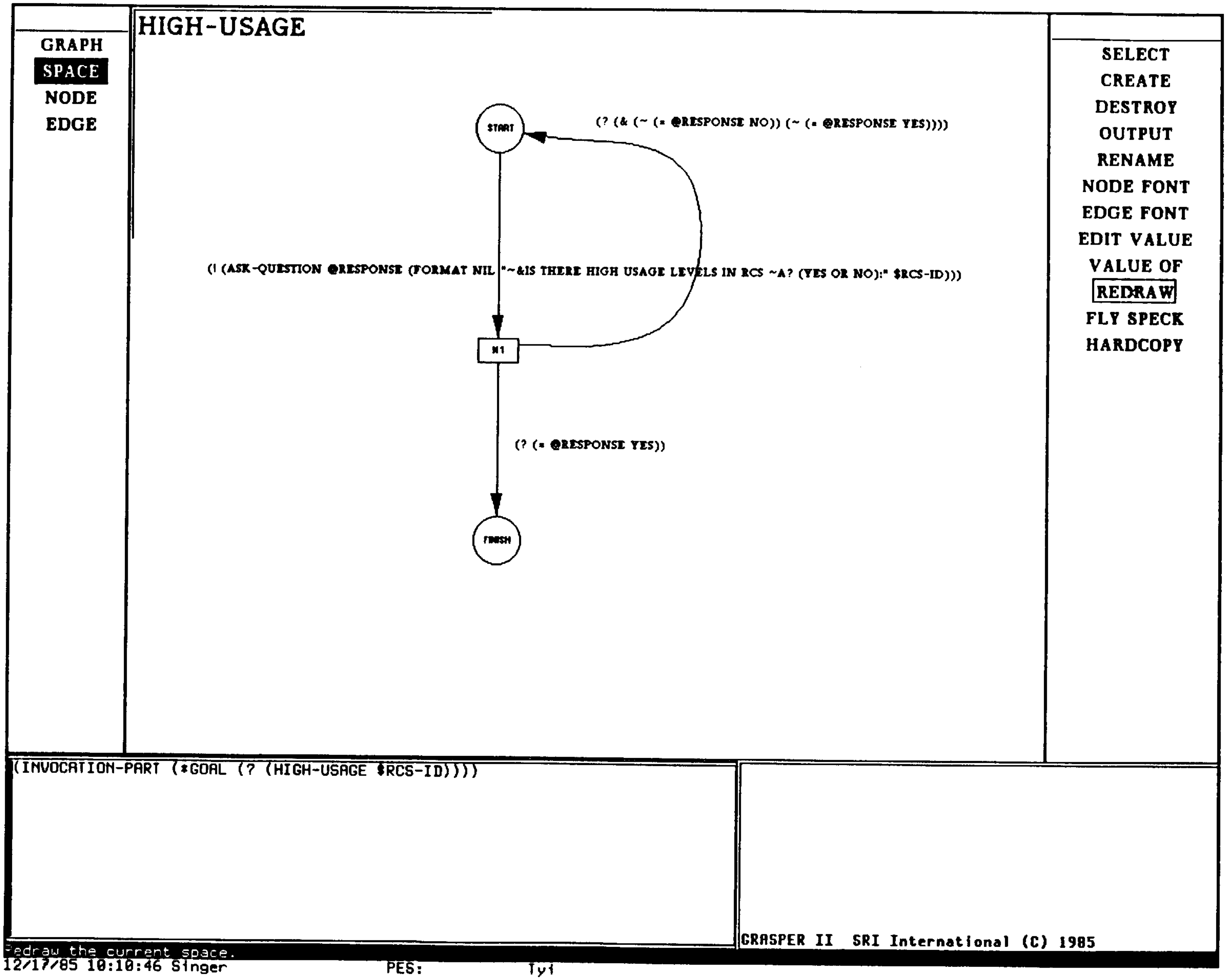


```
(! (& (ISOLATED-MANIFOLD SWITCH CLOSED $MANF-ID)(ISOLATED-MANIFOLD COMPUTER CLOSED $MANF-ID)))

(=> (& (CLOSED-MANIFOLD $MANF-ID)(~(OPENED-MANIFOLD $MANF-ID))))
```

```
(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: Ty!

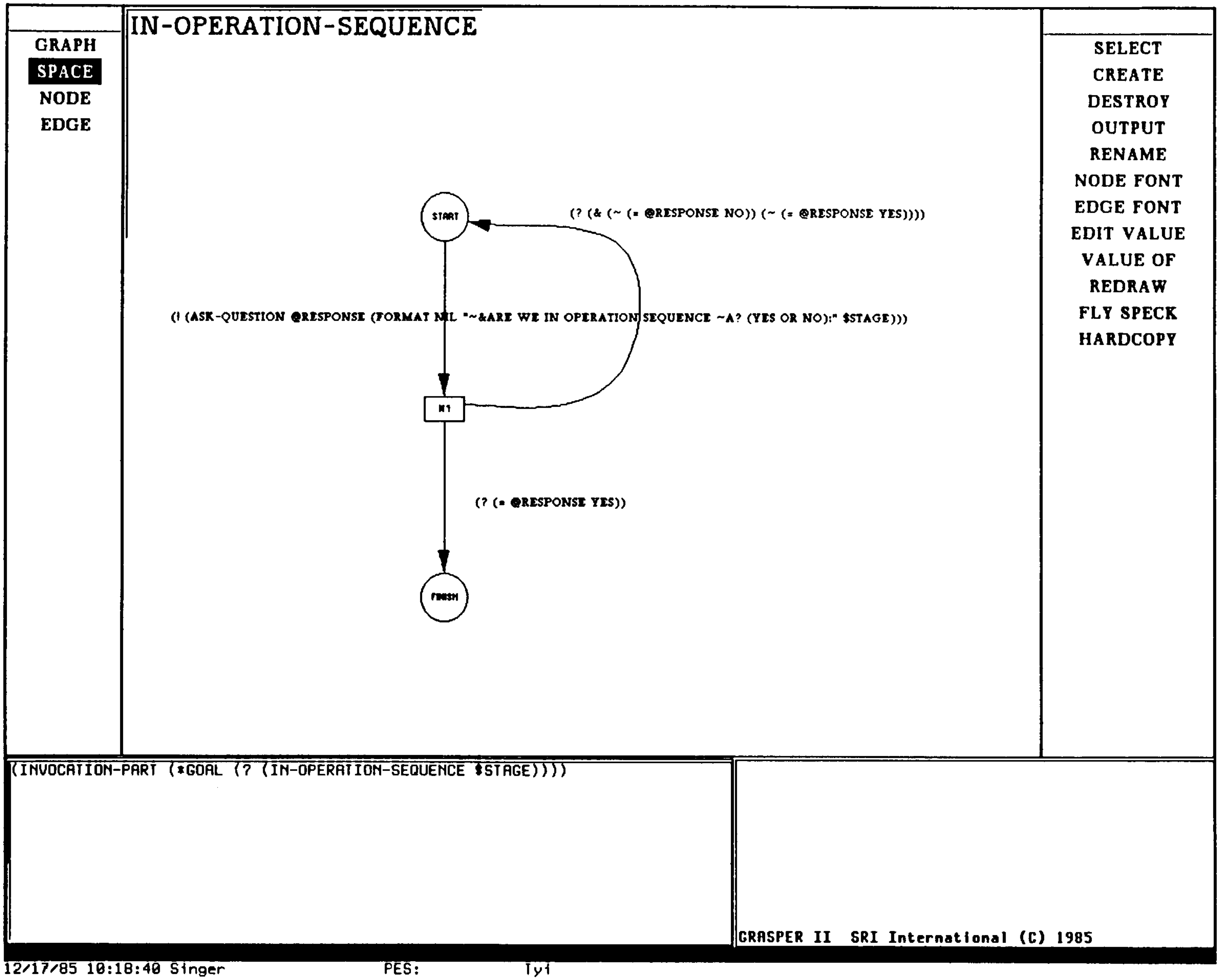


```
(? (& (~ (= @RESPONSE NO)) (~ (= @RESPONSE YES))))
(! (ASK-QUESTION @RESPONSE (FORMAT NIL "~&IS THERE HIGH USAGE LEVELS IN RCS ~A? (YES OR NO):" $RCS-ID)))
(? (= @RESPONSE YES))
```

```
(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:          Tyi
```



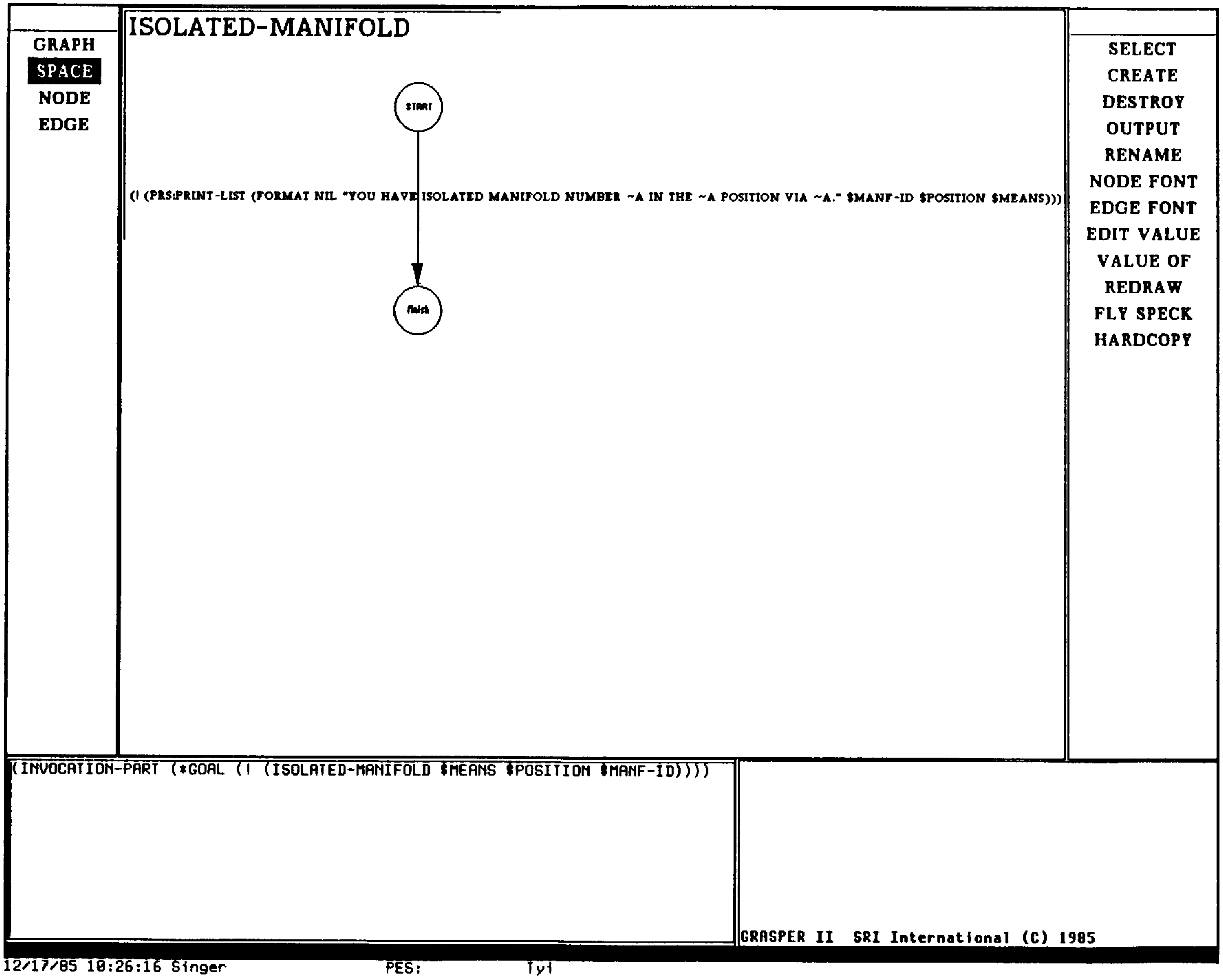
IN-OPERATION-SEQUENCE

```
(? (& (~ (= @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 10:18:40 Singer            PES:          Tyi
```

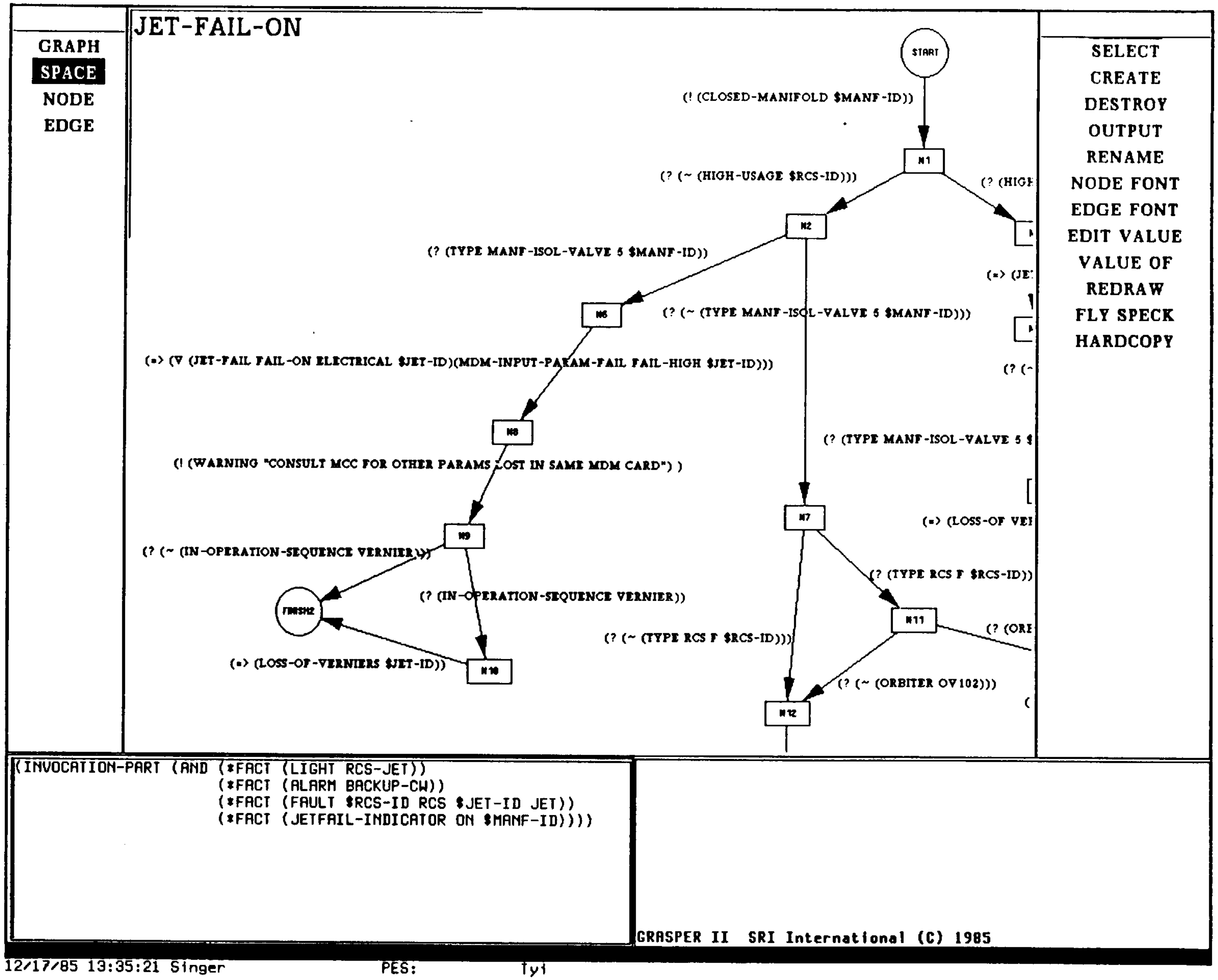


```
(! (PRS:PRINT-LIST (FORMAT NIL "YOU HAVE ISOLATED MANIFOLD NUMBER ~A IN THE ~A POSITION VIA ~A." $MANF-ID $POSITION $MEANS)))
```

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

GRASPER II SRI International (C) 1985

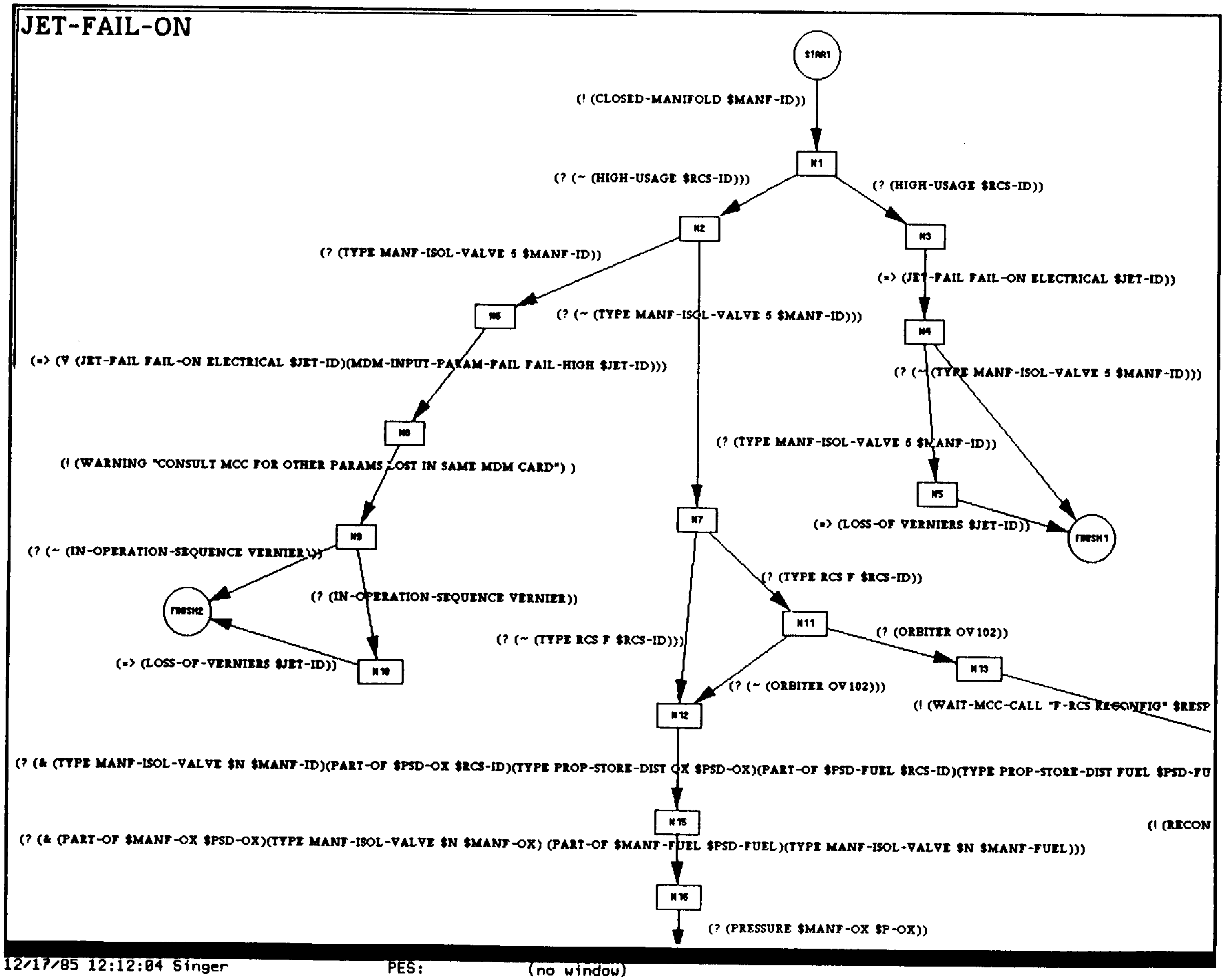
12/17/85 10:26:16 Singer   PES: Tyi


GRAPH
SPACE
NODE
EDGE
JET-FAIL-ON
START
(! (CLOSED-MANIFOLD $MANF-ID))
N1
(? (~ (HIGH-USAGE $RCS-ID)))
(? (HIGH
N2
(? (TYPE MANF-ISOL-VALVE 5 $MANF-ID))
(=> (JE
N6
(? (~ (TYPE MANF-ISOL-VALVE 5 $MANF-ID)))
(=> (V (JET-FAIL FAIL-ON ELECTRICAL $JET-ID)(MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID)))
(? (~
N8
(? (TYPE MANF-ISOL-VALVE 5 $
(! (WARNING "CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
N7
(=> (LOSS-OF VE
N9
(? (~ (IN-OPERATION-SEQUENCE VERNIER)))
(? (TYPE RCS F $RCS-ID))
(? (IN-OPERATION-SEQUENCE VERNIER))
FINISH2
N11
(? (ORI
(? (~ (TYPE RCS F $RCS-ID)))
(=> (LOSS-OF-VERNIERS $JET-ID))
N10
(? (~ (ORBITER OV102)))
N12
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))))
```

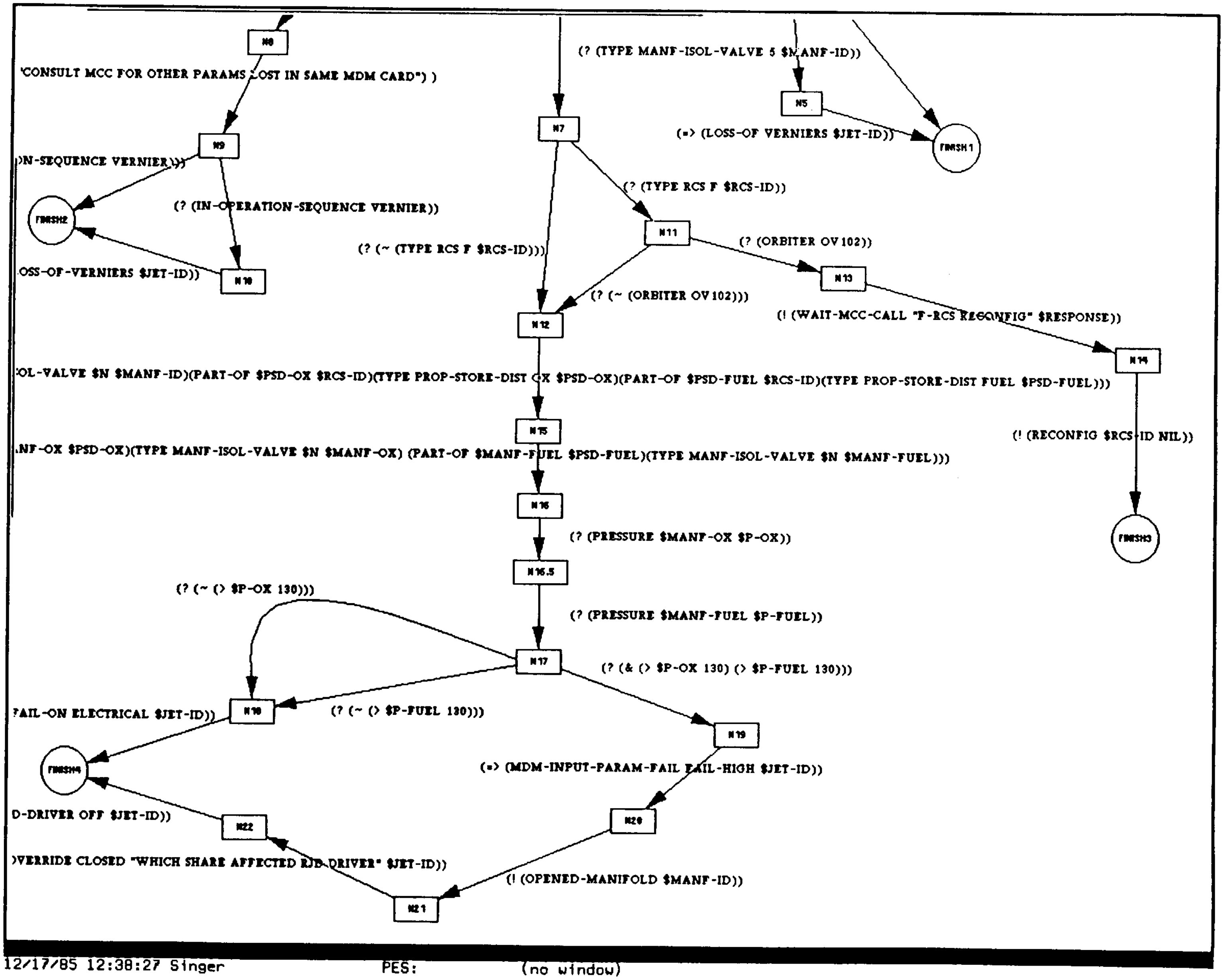
GRASPER II SRI International (C) 1985

12/17/85 13:35:21 Singer   PES: Tyi


JET-FAIL-ON
START
(! (CLOSED-MANIFOLD $MANF-ID))
N1
(? (~ (HIGH-USAGE $RCS-ID)))
(? (HIGH-USAGE $RCS-ID))
N2
N3
(? (TYPE MANF-ISOL-VALVE 6 $MANF-ID))
(=> (JET-FAIL FAIL-ON ELECTRICAL $JET-ID))
N6
(? (~ (TYPE MANF-ISOL-VALVE 5 $MANF-ID)))
N4
(=> (V (JET-FAIL FAIL-ON ELECTRICAL $JET-ID)(MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID)))
(? (~ (TYPE MANF-ISOL-VALVE 5 $MANF-ID)))
N8
(? (TYPE MANF-ISOL-VALVE 6 $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))
FINISH2
(? (IN-OPERATION-SEQUENCE VERNIER))
N11
(? (ORBITER OV102))
(? (~ (TYPE RCS F $RCS-ID)))
(=> (LOSS-OF-VERNIERS $JET-ID))
N10
N13
(? (~ (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 MANF-ISOL-VALVE $N $MANF-OX) (PART-OF $MANF-FUEL $PSD-FUEL)(TYPE MANF-ISOL-VALVE $N $MANF-FUEL)))
N16
(? (PRESSURE $MANF-OX $P-OX))

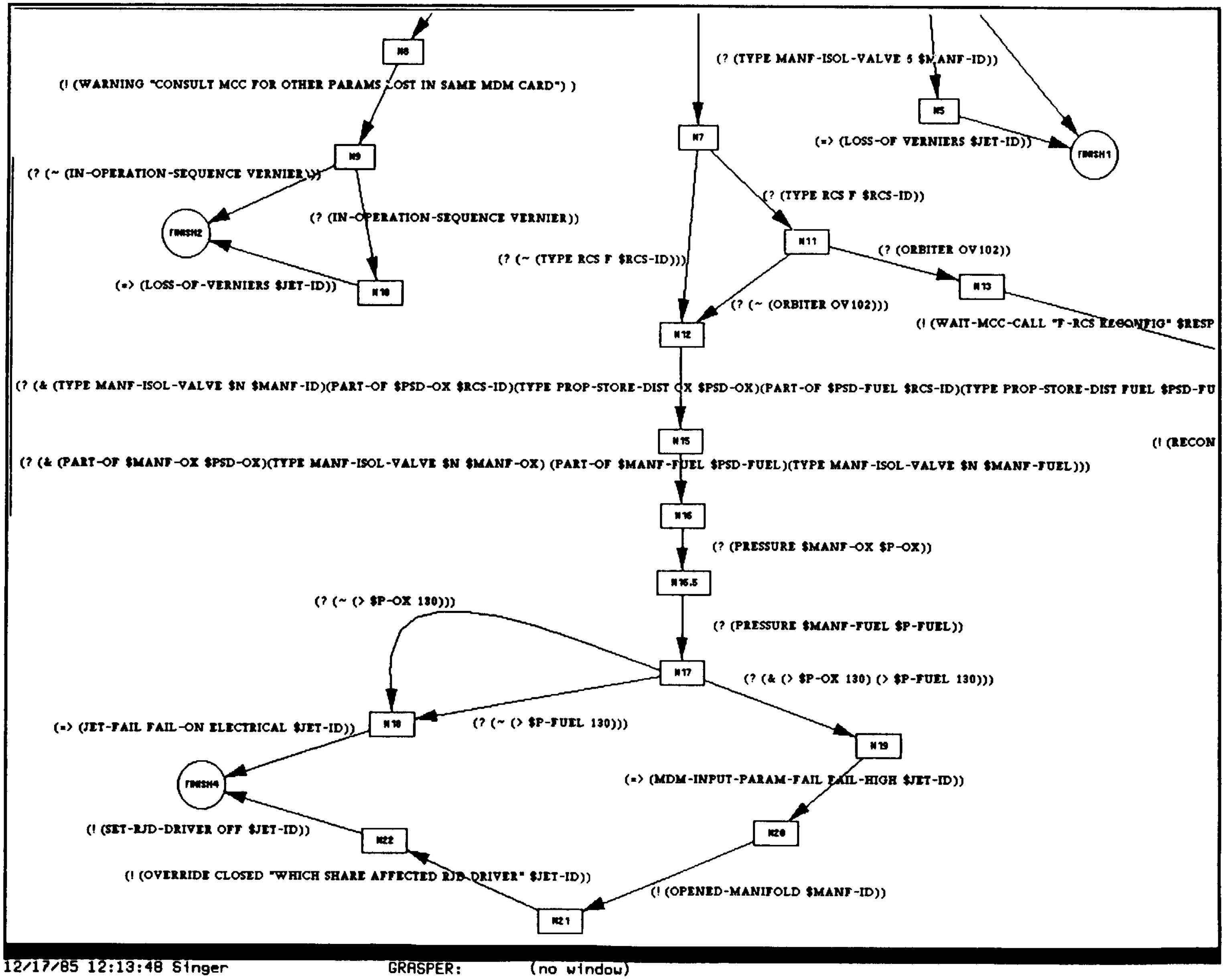
12/17/85 12:12:04 Singer PES: (no window)

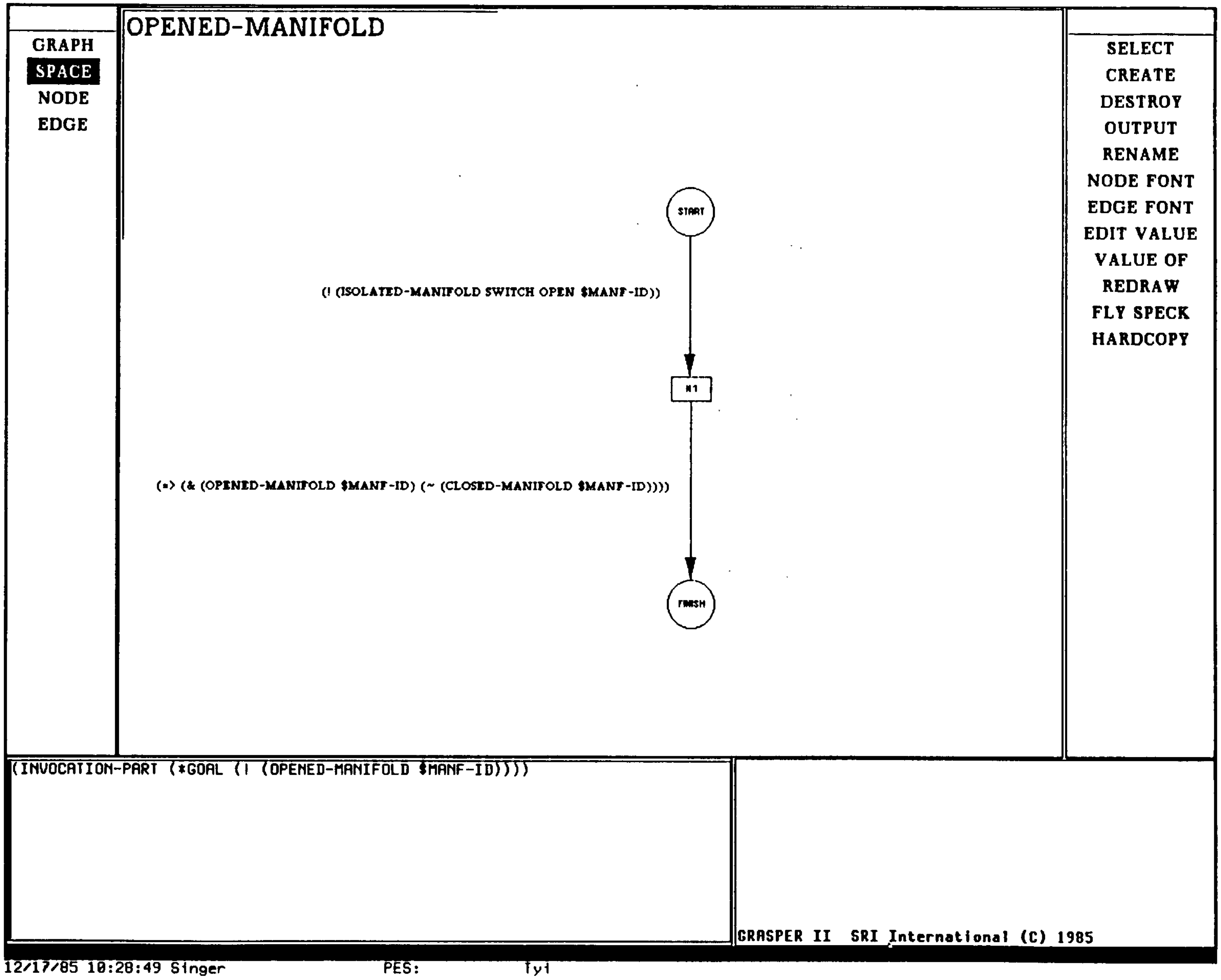
JET-FAIL-ON CONTINUED •


N8
'CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
N9
)N-SEQUENCE VERNIER)))
FINISH2
(? (IN-OPERATION-SEQUENCE VERNIER))
OSS-OF-VERNIERS $JET-ID))
N10
(? (TYPE MANF-ISOL-VALVE 5 $MANF-ID))
N5
N7
(=> (LOSS-OF VERNIERS $JET-ID))
FINISH1
(? (TYPE RCS F $RCS-ID))
N11
(? (ORBITER OV102))
(? (~ (TYPE RCS F $RCS-ID)))
N13
(? (~ (ORBITER OV102)))
(! (WAIT-MCC-CALL "F-RCS RECONFIG" $RESPONSE))
N12
N14
OL-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-FUEL)))
N15
(! (RECONFIG $RCS-ID NIL))
.NF-OX $PSD-OX)(TYPE MANF-ISOL-VALVE $N $MANF-OX) (PART-OF $MANF-FUEL $PSD-FUEL)(TYPE MANF-ISOL-VALVE $N $MANF-FUEL)))
N16
(? (PRESSURE $MANF-OX $P-OX))
FINISH3
N16.5
(? (~ (> $P-OX 130)))
(? (PRESSURE $MANF-FUEL $P-FUEL))
N17
(? (& (> $P-OX 130) (> $P-FUEL 130)))
FAIL-ON ELECTRICAL $JET-ID))
N18
(? (~ (> $P-FUEL 130)))
N19
FINISH4
(=> (MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID))
D-DRIVER OFF $JET-ID))
N22
N20
)VERRIDE CLOSED "WHICH SHARE AFFECTED RJD-DRIVER" $JET-ID))
(! (OPENED-MANIFOLD $MANF-ID))
N21

12/17/85 12:38:27 Singer PES: (no window)

JET-FAIL-ON CONTINUED


N8
(! (WARNING "CONSULT MCC FOR OTHER PARAMS LOST IN SAME MDM CARD"))
N9
(? (~ (IN-OPERATION-SEQUENCE VERNIER)))
FINISH2
(? (IN-OPERATION-SEQUENCE VERNIER))
(=> (LOSS-OF-VERNIERS $JET-ID))
N10
N7
(? (TYPE MANF-ISOL-VALVE 5 $MANF-ID))
N5
(=> (LOSS-OF VERNIERS $JET-ID))
FINISH1
(? (TYPE RCS F $RCS-ID))
N11
(? (ORBITER OV102))
N13
(? (~ (TYPE RCS F $RCS-ID)))
(? (~ (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 MANF-ISOL-VALVE $N $MANF-OX) (PART-OF $MANF-FUEL $PSD-FUEL)(TYPE MANF-ISOL-VALVE $N $MANF-FUEL)))
N16
(? (PRESSURE $MANF-OX $P-OX))
N16.5
(? (~ (> $P-OX 130)))
(? (PRESSURE $MANF-FUEL $P-FUEL))
N17
(? (& (> $P-OX 130) (> $P-FUEL 130)))
(=> (JET-FAIL FAIL-ON ELECTRICAL $JET-ID))
N18
(? (~ (> $P-FUEL 130)))
N19
FINISH4
(=> (MDM-INPUT-PARAM-FAIL FAIL-HIGH $JET-ID))
(! (SET-RJD-DRIVER OFF $JET-ID))
N22
N20
(! (OVERRIDE CLOSED "WHICH SHARE AFFECTED RJD-DRIVER" $JET-ID))
(! (OPENED-MANIFOLD $MANF-ID))
N21
12/17/85 12:13:48 Singer
GRASPER:
(no window)



OPENED-MANIFOLD

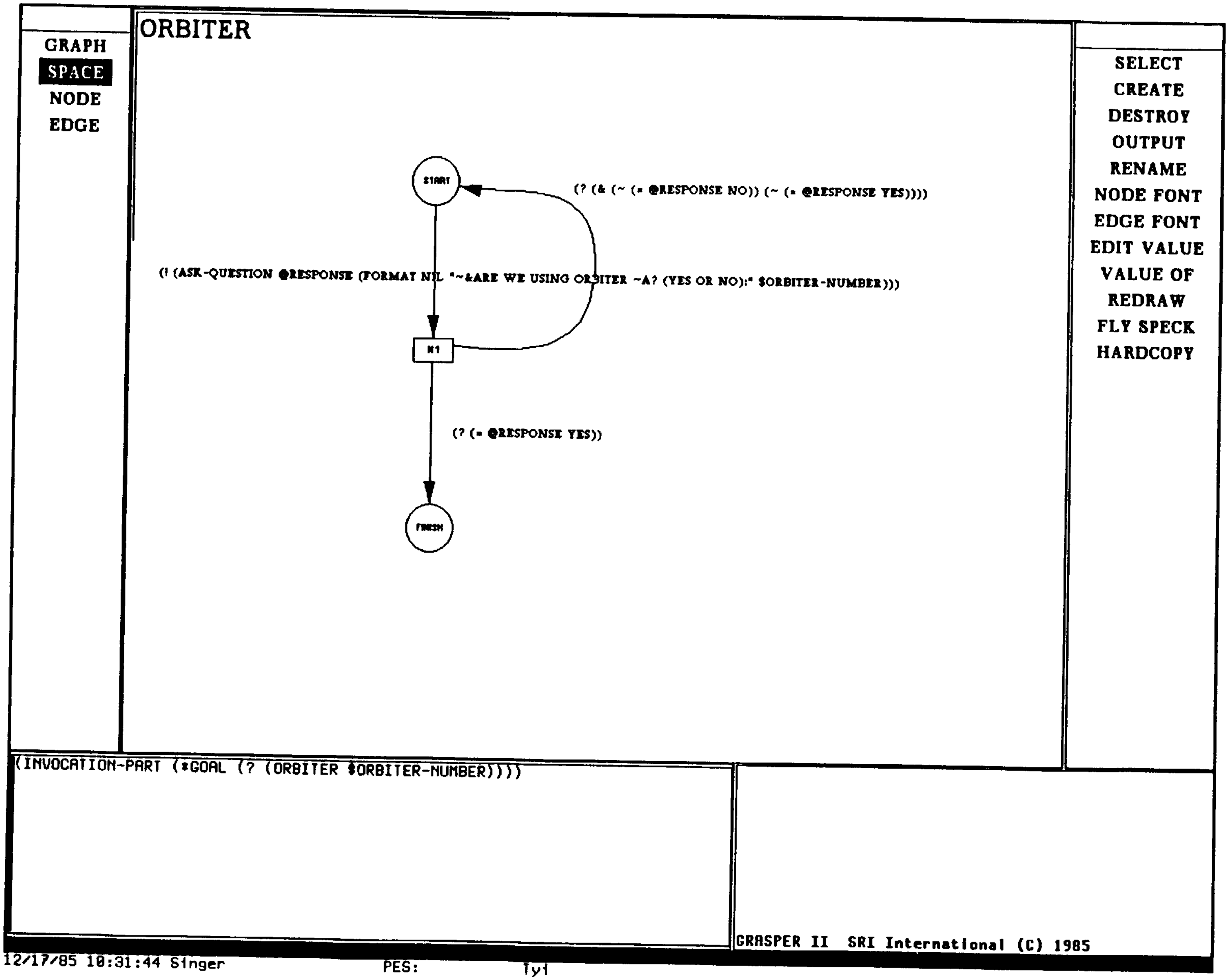
(! (ISOLATED-MANIFOLD SWITCH OPEN $MANF-ID))

(=> (& (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: Tyi

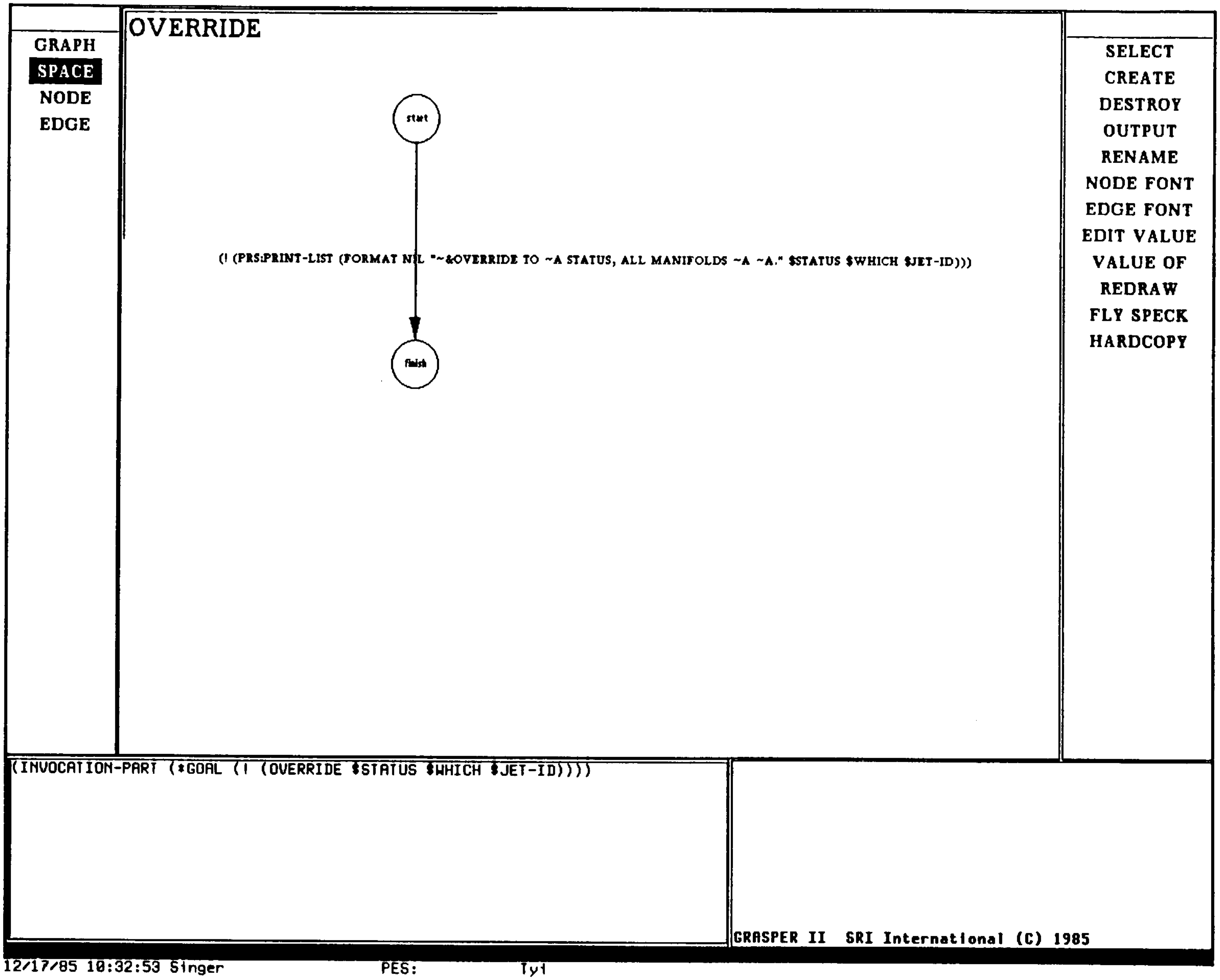


```
(? (& (~ (= @RESPONSE NO)) (~ (= @RESPONSE YES))))
(! (ASK-QUESTION @RESPONSE (FORMAT NIL "~&ARE WE USING ORBITER ~A? (YES OR NO):" $ORBITER-NUMBER)))
(? (= @RESPONSE YES))
```

```
(INVOCATION-PART (*GOAL (? (ORBITER $ORBITER-NUMBER))))
```

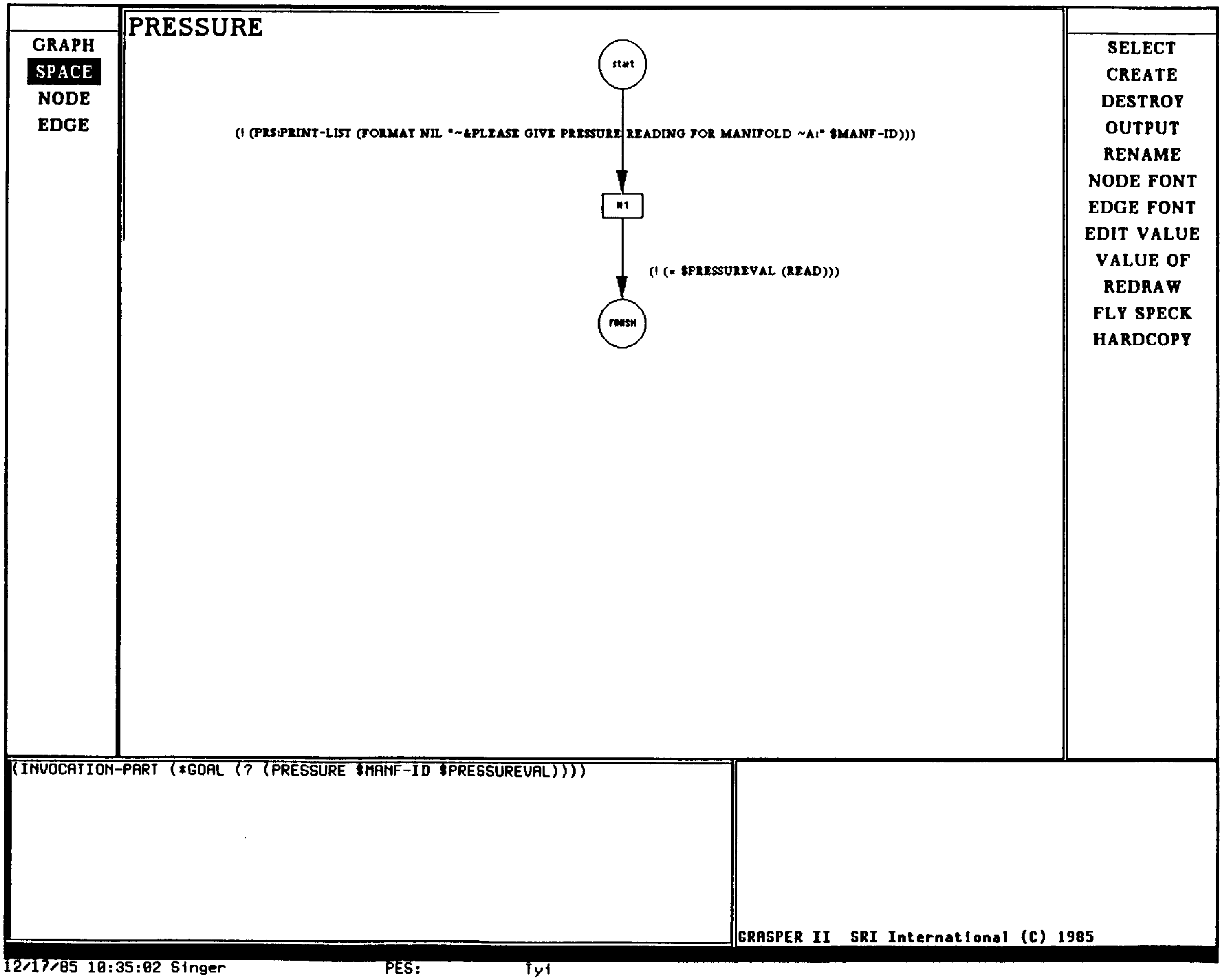
GRASPER II SRI International (C) 1985

```
12/17/85 10:31:44 Singer                 PES:          Ty1
```



```
(! (PRS:PRINT-LIST (FORMAT NIL "~&OVERRIDE TO ~A STATUS, ALL MANIFOLDS ~A ~A." $STATUS $WHICH $JET-ID)))
```

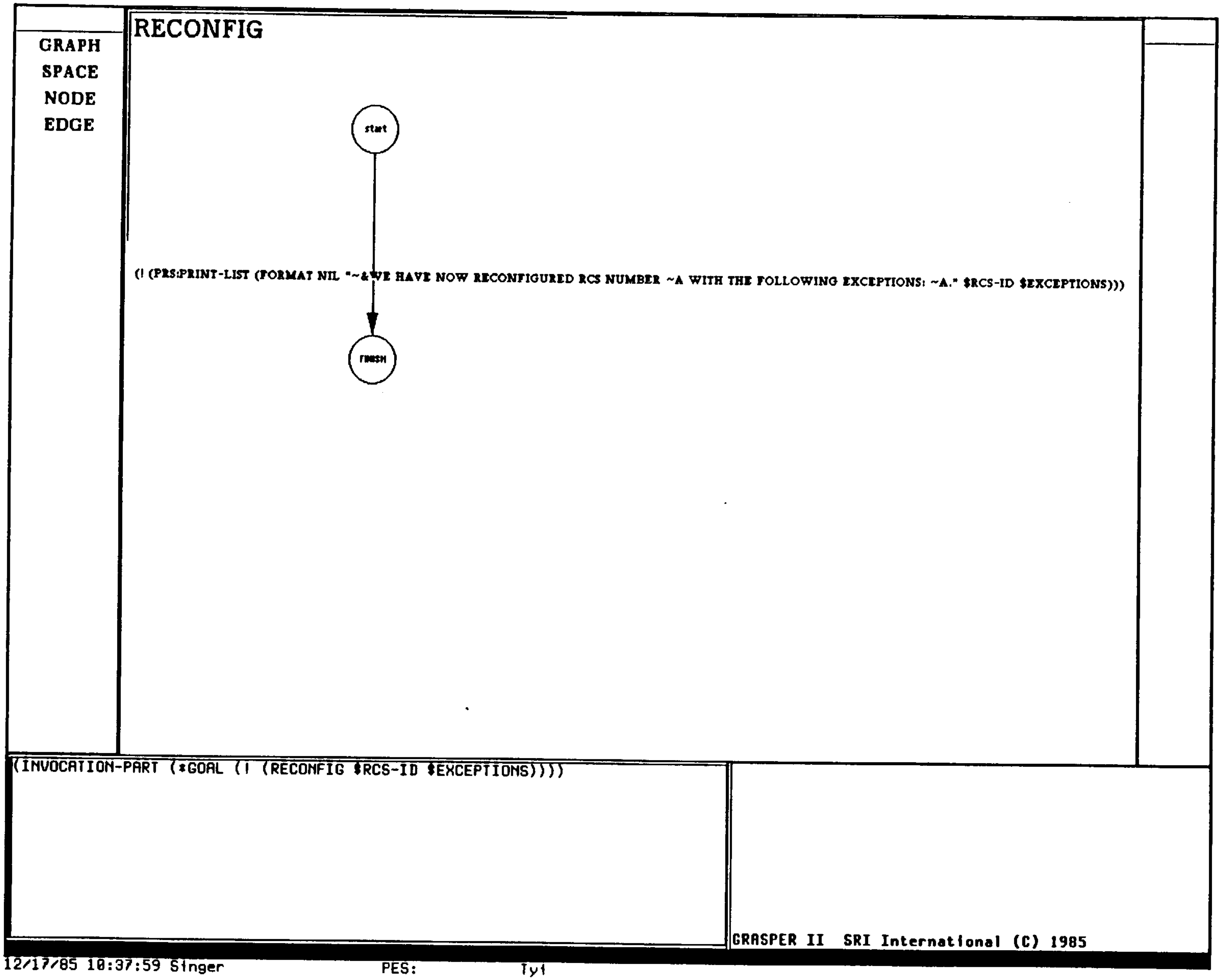
```
(INVOCATION-PART (*GOAL (! (OVERRIDE $STATUS $WHICH $JET-ID))))
```



```
(! (PRS:PRINT-LIST (FORMAT NIL "~&PLEASE GIVE PRESSURE READING FOR MANIFOLD ~A:" $MANF-ID)))
(! (= $PRESSUREVAL (READ)))
```

```
(INVOCATION-PART (*GOAL (? (PRESSURE $MANF-ID $PRESSUREVAL))))
```

```
GRASPER II  SRI International (C) 1985
12/17/85 10:35:02 Singer             PES:          Tyi
```



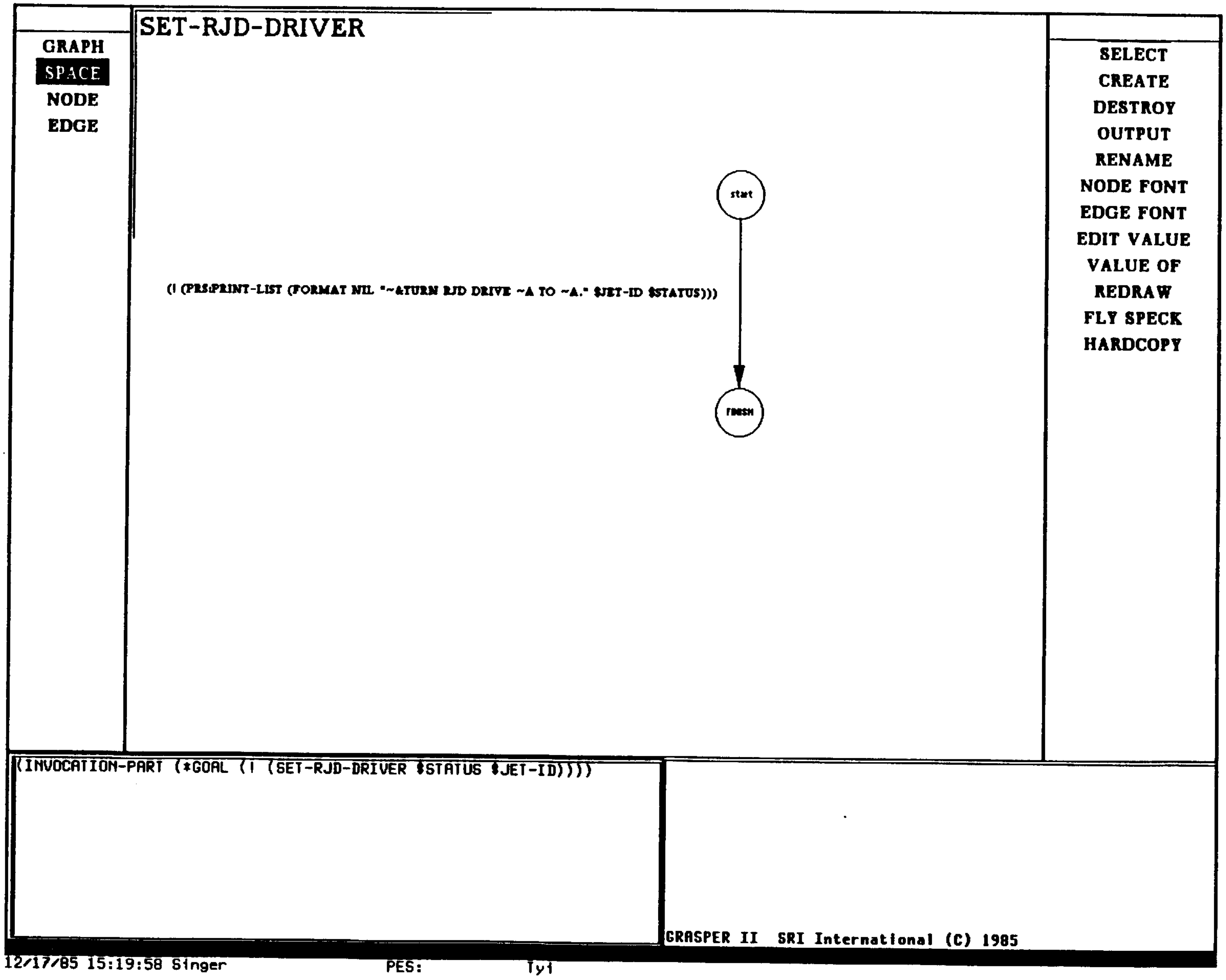
RECONFIG

```
(! (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))))
```

GRASPER II SRI International (C) 1985

```
12/17/85 10:37:59 Singer                PES:          Tyi
```

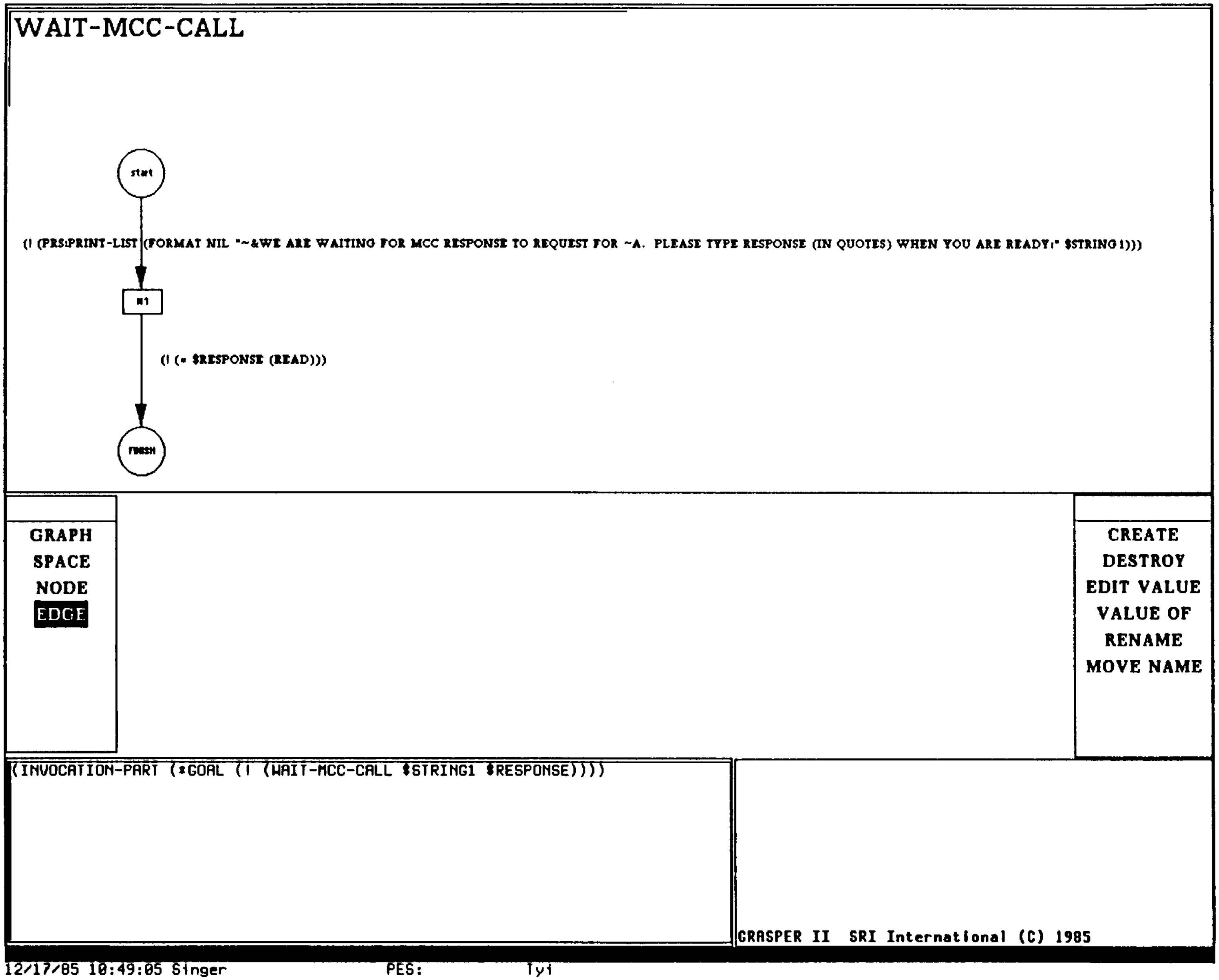


```
(! (PRS:PRINT-LIST (FORMAT NIL "~&TURN RJD DRIVE ~A TO ~A." $JET-ID $STATUS)))
```

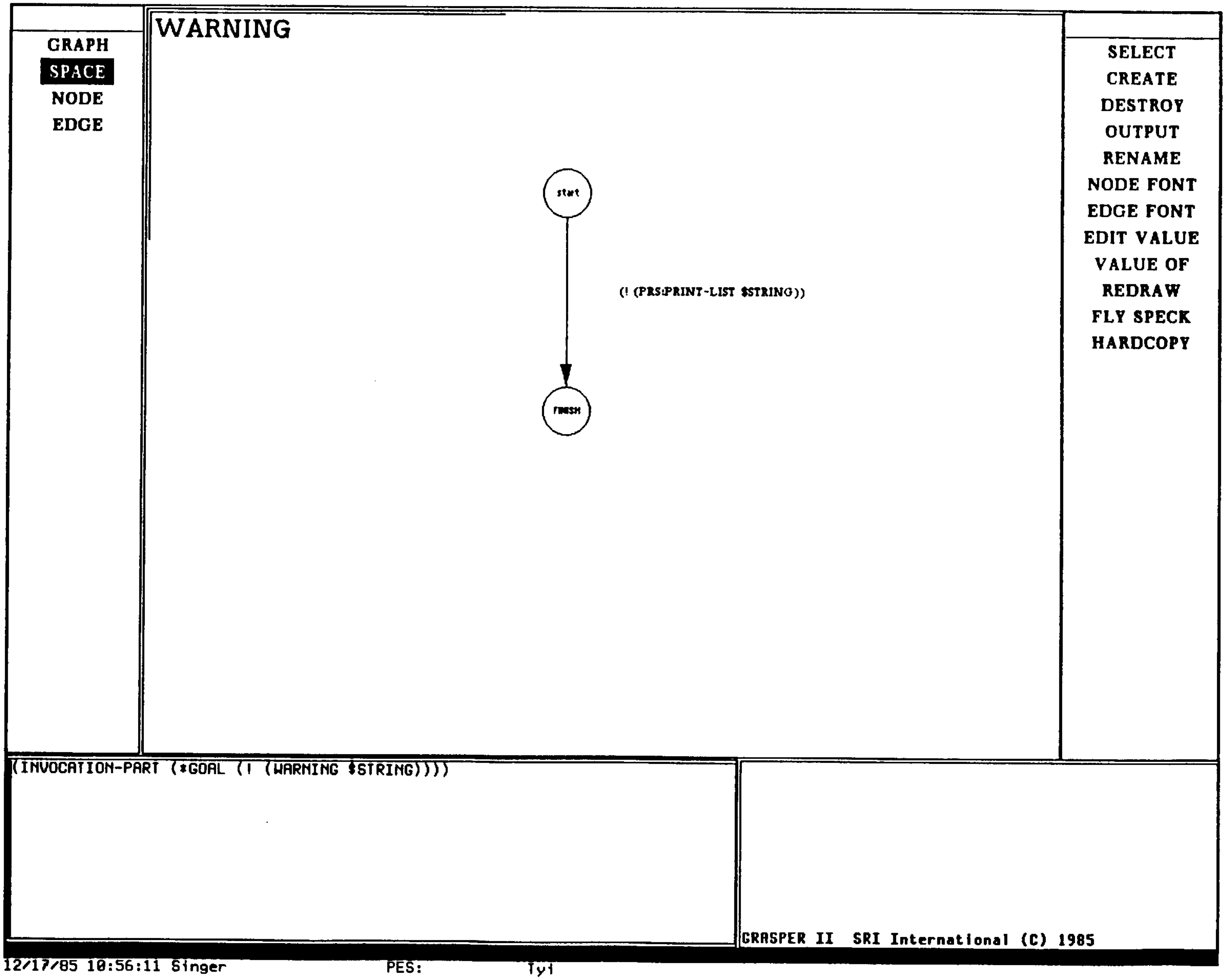
```
(INVOCATION-PART (*GOAL (! (SET-RJD-DRIVER $STATUS $JET-ID))))
```

```
GRASPER II  SRI International (C) 1985
```

```
12/17/85 15:19:58 Singer                PES:            Tyi
```



```
(! (PRS:PRINT-LIST (FORMAT NIL "~&WE ARE WAITING FOR MCC RESPONSE TO REQUEST FOR ~A.  PLEASE TYPE RESPONSE (IN QUOTES) WHEN YOU ARE READY:" $STRING1)))

(! (= $RESPONSE (READ)))

(INVOCATION-PART (*GOAL (! (WAIT-MCC-CALL $STRING1 $RESPONSE))))

GRASPER II  SRI International (C) 1985

12/17/85 10:49:05 Singer                    PES:             Tyi
```



WARNING

```
(! (PRS:PRINT-LIST $STRING))
```

```
(INVOCATION-PART (*GOAL (! (WARNING $STRING))))
```

```
GRASPER II  SRI International (C) 1985
```

```
12/17/85 10:56:11 Singer                PES:          Tyi
```

Appendix D

Notational Conventions

Symbol	Representing class
$\alpha, \beta, \gamma, \phi, \psi$	State and temporal propositions
δ	Transition function
τ	Special hidden event
;	Sequential composition
:	Prefixing
$+, \vert$	Nondeterministic choice
$\Vert, \&$	Parallelism
$\Leftrightarrow$	Synchrony
$\wedge$	Conjunction
$\vee$	Disjunction
$\neg$	Negation
?	Temporal operator ("test")
!	Temporal operator ("achieve")
#	Temporal operator ("preserve")
$\Rightarrow$	Metapredicate ("insert in data base")
$x, etc.	Global variables
%x, etc.	Local variables
@x, etc.	Program variables
$\langle P \rangle$	Successful behaviors of process P
$\langle P \rangle_F$	Failed behaviors of process P

References

[1] Aikins, J.S., "Prototypical Knowledge for Expert Systems," *Artificial Intelligence*, Vol. 20, pp 163-210 (1983).

[2] Allen, J. F., "A General Model of Action and Time," Computer Science Report TR 97, University of Rochester, Rochester, New York (1981).

[3] Allen, J.F., "Maintaining Knowledge about Temporal Intervals," *Comm. ACM*, Vol. 26, pp 832–843 (1983).

[4] Barringer, H., Kuiper, R., and Pnueli, A., "Now You May Compose Temporal Logic Specifications," *Proceedings of the Symposium on Theory of Computing* (1984).

[5] Chapman, D. "Planning for Conjunctive Goals," Master's thesis, Technical Report MIT-AI-TR-802, MIT Laboratory for Artificial Intelligence, Cambridge, Massachusetts (1985).

[6] Davis, R., "Applications of Metalevel Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases," STAN-CS-76-552, HPP-76-7, Stanford University, Stanford, California (1976).

[7] Davidson, D., *Actions and Events*, Clarendon Press, Oxford, England (1980).

[8] Doyle, J., "A Truth Maintenance System," *Artificial Intelligence*, Vol. 12 (1979).

[9] Doyle, J. and London, P. "A Selected Descriptor-Indexed Bibliography to the Literature on Belief Revision," AI Memo 568, MIT-AI Lab, Massachusetts (1980).

[10] Feurzeig, W., Frederiksen, J., White, B., and Horwitz, P. "Designing an Expert System for Training Automotive Electrical Troubleshooting," in *Artificial Intelligence in Maintenance: Proceedings of the Joint Services Workshop*, ed. J.J. Richardson (1984).

[11] Fikes, R. E., and Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence* Vol. 2, pp. 189-208 (1971).

[12] Fikes, R.E., "Knowledge Representation in Automatic Planning Systems," SRI Tech Note 119, SRI International, Menlo Park, California (1976).

[13] Fikes, R.E., and T. Kehler, "The Role of Frame-Based Representation in Reasoning," Comm. ACM, Vol. 28, pp 904-920 (1985).

[14] Forgy, C., and McDermott, J., "OPS, a Domain-Independent Production System Language" *Proceedings of the 5th Int'l. Joint Conf. on Artificial Intelligence*, pp 933-939 (1977).

[15] Genesereth, M.R., and Smith, D.E., "Metalevel Architecture," Memo HPP-81-6, Stanford University, Stanford, California (1982).

[16] Georgeff, M.P., "Procedural Control in Production Systems," *Artificial Intelligence*, Vol. 18, pp 175-201 (1982).

[17] Georgeff, M. P. "Communication and Interaction in Mulitagent Planning," *Proceedings of the Third National Conference on Artificial Intelligence*, Washington, D.C. (1983).

[18] Georgeff, M.P. and Bonollo, U. "Procedural Expert Systems," *Proceedings of the 8th Int'l. Joint Conf. on Artificial Intelligence*, Karlsruhe, Germany (1983).

[19] Georgeff, M.P., "A Theory of Action for Multiagent Planning," *Proceedings of the National Conference on Artificial Intelligence*, Austin, Texas (1984).

[20] Georgeff, M.P., Lansky, A. and Bessiere, P., "A Procedural Logic", *Proceedings of the 9th Int'l. Joint Conf. on Artificial Intelligence*, Los Angeles, California (1985).

[21] Georgeff, M.P. "Reasoning about Process" forthcoming Technical Note, Artificial Intelligence Center, SRI International, Menlo Park, California (1985).

[22] Hayes, P. J., "The Frame Problem and Related Problems in Artificial Intelligence," in *Artificial and Human Thinking*, A. Elithorn and D. Jones (eds.), Jossey-Bass (1973).

[23] Hayes, P.J., "In Defense of Logic," *Proceedings of the 5th Int'l. Joint Conf. on Artificial Intelligence*, Cambridge, Massachusetts, pp 559-565 (1977).

[24] Hayes-Roth, F., "Rule-Based Systems," Comm. ACM, Vol. 28, pp 921-932 (1985).

[25] Hendrix, G. G., "Modeling Simultaneous Actions and Continuous Processes," *Artificial Intelligence*, Vol. 4, pp 145-180 (1973).

[26] Hoare, C. A. R., *Communicating Sequential Processes*, Series in Computer Science, C. A. R. Hoare (ed.), Prentice Hall, Englewood Cliffs, New Jersey (1985).

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

[28] Lansky, A. L., "Behavioral Specification and Planning for Multiagent Domains," Technical Note 360, Artificial Intelligence Center, SRI International, Menlo Park, California (1985).

[29] Lenat, D.B., "Automated Theory Formation in Mathematics,," *Proceedings of the 5th Int'l. Joint Conf. on Artificial Intelligence*, Cambridge, Massachusetts, pp 833-842 (1977).

[30] McCarthy, J., "Programs with Common Sense," in *Semantic Information Processing* M. Minsky ed., MIT Press, Cambridge, Massachusetts (1968).

[31] McCarthy, J., and Hayes, P.J., "Some Philosophical Problems from the Standpoint of Artificial Intelligence, in *Machine Intelligence 4*, pp 463-502 (1969).

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

[33] Manna, Z., and Waldinger, R., "Problematic Features of Programming Languages: A Situational-Calculus Approach," *Acta Informatica*, Vol. 16, pp 371-426 (1981).

[34] Maylin, J. T., and N. Lance, "An Expert System for Fault Management and Automatic Shutdown Avoidance in a Regenerative Life Support Subsystem," *Proceedings of the Instrument Society of America First Annual Workshop on Robotics and Expert Systems*, Houston, Texas (1985).

[35] Milne, G., "Synchronized Behaviour Algebras: A Model for Interacting Systems," Technical Report, Department of Computer Science, University of Southern California (1979).

[36] Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer Verlag, New York (1980).

[37] Moore, R.C., "Reasoning about Knowledge and Action," Technical Note 191, Artificial Intelligence Center, SRI International, Menlo Park, California (1980).

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

[39] Nilsson, N.J., "Triangle Tables: A Proposal for a Robot Programming Language", Technical Note 347, Artificial Intelligence Center, SRI International, Menlo Park, California (1985).

[40] Pednault, E. P. D., "Toward a Mathematical Theory of Plan Synthesis," Ph.D. thesis, Department of Electrical Engineering, Stanford University, Stanford, California (forthcoming).

[41] Reiter, R., "A Logic for Default Reasoning," *Artificial Intelligence*, 13, pp 81-132 (1980).

[42] Richardson, J.J., *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).

[43] Rosenschein, S.J., "Plan Synthesis: A Logical Perspective," *Proceedings of the 7th Int'l. Joint Conf. on Artificial Intelligence*, pp 331-337 (1981).

[44] Rosenschein, S. J., and Kaelbling, L. P., "A Formal Approach to the Design of Intelligent Embedded Systems," to appear in *Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge*, Monterey, California (1985).

[45] Sacerdoti, E.D. *A Structure for Plans and Behaviour*, Elsevier, North Holland, New York (1977).

[46] Shoham, Y. and Dean, T., "Temporal Notation and Causal Terminology," Working Paper, Department of Computer Science, Yale University, New Haven, Connecticut (1985).

[47] Stefik, M. "An Examination of a Frame-Structured Representation System," *Proceedings of the 6th Int'l. Joint Conf. on Artificial Intelligence*, pp 845-852 (1979).

[48] Stefik, M. "Planning with Constraints," *Artificial Intelligence*, Vol. 16, pp 111-140 (1981).

[49] Stuart, C. J., " Implementation of a Multiagent Plan Synchronizer Using a Temporal Logic Theorem Prover," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, California (1985).

[50] Tate, A. "Goal Structure – Capturing the Intent of Plans," *Proceedings of the 6th European Conference on Artificial Intelligence*, pp 273-276 (1984).

[51] vanMelle, W. "A Domain-Independent System That Aids in Constructing Knowledge-Based Consultation Programs," Memo HPP-80-1, Report No. STAN-CS-80-814, Computer Science Department, Stanford University (1980).

[52] Vere, S., "Planning in Time: Windows and Durations for Activities and Goals," Jet Propulsion Lab, Pasadena, California (1981).

[53] Wilkins, D.E., "Domain Independent Planning: Representation and Plan Generation," *Artificial Intelligence*, Vol. 22, pp 269-301 (1984).