SRI International

Procedural Knowledge

Technical Note No. 411

January 1987

By: Michael P. Georgeff, Program Director
Amy L. Lansky, Computer Scientist
Representation and Reasoning Program
Artificial Intelligence Center

and

Center for the Study of Language and Information
Stanford University

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

This paper was originally published in *The Proceedings of the Institute of Electrical and Electronics Engineers,* Special Issue on Knowledge Representation, Volume 74, Number 10, pages 1383-1398, October 1986.

This research has been made possible in part by a gift from the System Development Foundation, by the Office of Naval Research under Contracts N00014-80-C-0296 and N00014-85-C-0251, and by the National Aeronautics and Space Administration, Ames Research Center under Contract NAS2-11864. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representative of the official policies, either expressed or implied, of the Office of Naval Research, NASA, or the United States Government.

333 Ravenswood Avenue • Menlo Park, CA 94025-3493 • (415) 326-6200 • FAX: (415) 326-5512 • Telex: 334486

Abstract

Much of commonsense knowledge about the real world is in the form of *procedures* or *sequences* of actions for achieving particular goals. In this paper, a formalism is presented for representing such knowledge using the notion of *process*. A declarative semantics for the representation is given, which allows a user to state *facts* about the effects of doing things in the problem domain of interest. An operational semantics is also provided, which shows *how* this knowledge can be used to achieve particular goals or to form intentions regarding their achievement. Given both semantics, our formalism additionally serves as an executable specification language suitable for constructing complex systems. A system based on this formalism is described, and examples involving control of an autonomous robot and fault diagnosis for NASA's space shuttle are provided.

1 Introduction

There is an increasing demand for systems or artificial agents that can interact with dynamic environments to achieve particular goals. Common applications of this type include robotic functions, contruction and assembly tasks, navigation and exploration by autonomous vehicles, control and monitoring of systems, and servicing and maintenance of equipment. Agents capable of operating effectively in these kinds of domains must be able to reason about their tasks and determine how to act in given situations – i.e., they must be capable of *practical reasoning* [6]. To build such systems we need to be able to represent knowledge about the effects of actions and how these actions can be combined to achieve specific goals.

Within artificial intelligence (AI), there have been two approaches to this problem, 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 [1,12,17]. This research has focused mainly on problems in natural-language understanding concerned with the meaning of action sentences. Some attempts have also been made to indicate how these theories could be used for general reasoning about actions [2,17]. Second, there is work on planning – that is, the problem of constructing a plan by searching for a sequence of actions that will yield a particular goal [2,7,21,22,23,25,26,27].

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. Actually searching the space of possible future courses of action, which is the basis of most AI planning systems, is relatively rare.

For example, consider the task of driving to work each day. For most of us, our plan of action has been worked out in advance. Once we establish a goal for ourselves to leave home and get to work, we follow some internal procedure or pattern of getting to the car, getting in, driving a certain route, searching the parking lot in a particular fashion once we get there, parking, and then walking to the office. Rarely do we ever derive this plan from first principles. In fact, we often seem to perform these actions without even thinking! This pattern applies to many of the tasks we perform everyday.

Of course, there are often situations in which our normal procedures or plans must be modified or reconsidered. Rather than derive completely new plans, we usually adjust to situations by operating in a piecemeal fashion; we keep an overall plan in mind and elaborate it as we proceed and acquire more knowledge of the world. For example, if we run into an obstacle on the road to work, a new route may need to be found for part of the journey. Although the top level plan of action may remain the same, different means of realizing pieces of the plan would be used, depending on the particular situation. For instance, one might be able to avoid the obstacle simply by driving around it. On the other hand, if the obstacle were large, one may have to use more complex avoidance procedures that involved turning onto side streets, continuing in the same general direction, and the like.

This strategy of operation might be called *partial hierarchical* planning. The idea is simply to intermix the formation of plans and their execution; i.e., form a partial overall plan, figure out near term means, execute them, further expand the near-term plan of action

some more, execute, and so on. This approach has many advantages. First of all, systems generally lack sufficient knowledge to expand a plan of action to the lowest levels of detail – at least if the plan is expected to operate effectively in a real-world situation. The world around us is simply too dynamic to anticipate all circumstances. By finding and executing relevant procedures when they are truly needed, a system may stand a better chance of achieving its goals effectively.

A combined planning/execution architecture can also be *reactive*. By reactive, we mean more than a capability of modifying current plans in order to accomplish given goals; a reactive system should also be able to completely change its focus and pursue new goals when the situation warrants it. This is essential for domains in which emergencies can occur and is an integral component of human practical reasoning. In a system that expands plans dynamically and incrementally, there are frequent opportunities to react to new situations and changing goals. Such a system is therefore able to rapidly modify its intentions (plans of action) on the basis of what it currently perceives as well as upon what it already believes, intends, and desires.

Of course, how we represent knowledge is just as important as how we use it. Representing knowledge of dynamic environments as procedures, rather than as sets of rules about individual atomic actions, has many advantages. Most obvious is the computational efficiency gained in not having to reconstruct particular plans of action over and over again from knowledge of individual actions.

Second, much expert knowledge is already procedural in nature: for example, consider the knowledge one might have about kicking a football, performing a certain dance movement, cooking a roast dinner, solving Rubik's cube, or diagnosing an engine malfunction. In such cases, it is highly disadvantageous to "deproceduralize" this knowledge into disjoint rules or descriptions of individual actions. To do so invariably involves encoding the control structure of the procedure in some way. Usually this is done by linking individual actions with "control conditions," whose sole purpose is to ensure that the rules or actions are executed in the correct order. This approach can be very tedious and confusing, destroys extensibility, and lacks any natural semantics.

By having direct access to the particular behavior that a procedure has or will follow, we can also make much more effective use of procedural knowledge. For example, suppose that an agent knows that in order to achieve G it will follow a course of action of the form: $X \rightarrow Y \rightarrow Z$. When performing X, the agent will know that it is setting up a situation for later performance of Z. When the agent finally does get around to Z, it will likewise know that X and Y have been performed, thereby allowing it to make various assumptions – for instance, that certain environmental conditions will be in place because X and Y tend to make them that way.

Having access to one's history of actions can also free an agent from stringent reliance on its sensors – the agent can derive much information about the state of the world simply from knowledge of its previous activities (see also [4,20,19]). For example, if cooks follow the steps of a recipe exactly, they can usually assume the food will turn out right – they don't have to perpetually taste it. They might not even know what it *should* taste like, especially at intermediate points in the recipe. Work by Lansky [14] has taken very seriously the notion of

encoding domain requirements primarily in terms of actions and their interrelationships and deriving knowledge from past activity. Such an approach appears particularly advantageous for describing the properties of multiagent domains.

Another advantage of representing knowledge as procedures is that we are able to reason about those procedures as *whole entities*. For example, given two procedures for fixing a broken pipe, we can evaluate those procedures in their entirety and decide which has the best cost/benefit properties in a particular situation. This type of "metalevel" or reflective reasoning about our own internal procedures enables us to perform effectively and would be intractible if the steps of the procedure had been broken down into seemingly unrelated rules.

The primary aim of this paper is provide a basis for representing and reasoning about procedural forms of knowledge in a way that allows an agent to deal effectively with a dynamically changing world. It is important that the agent be able to use these procedures to form intentions to achieve given goals, to react to particular events, to modify intentions in the light of new beliefs or goals, and to reason about these things in a timely way.

To do this, we first introduce the notions of action and process. We then provide a means for describing these and define a declarative and operational semantics for our formalism. Together, these provide a way of both stating procedural facts about the problem domain and a method of practical reasoning about how to use this knowledge to achieve given goals.

More generally, the formalism may also be viewed as the basis for an executable specification language. Just as for Prolog [5], the declarative semantics provides a means for stating *facts* about the problem domain, and the operational semantics yields a means of *using* these facts to achieve given goals.

A system based on the proposed representation has been implemented in ZETA-LISP. It is currently being used for the control of an intelligent robot and has been applied to problems of fault isolation and diagnosis on NASA's space shuttle. An early version of an implemented system is described by Georgeff and Bonollo [9] and the latest work by Georgeff and Lansky [11].

2 Processes and Actions

We assume that, at any given instant, the world is in a particular *world state*. This state embodies not only the external environment, but also the world inside an agent – its internal cognitive world. As time progresses, the world state changes through the occurrence of *actions* (or *events*).[1]

Most early work in AI represented actions as mappings from world states to world states [7,16,21]. However, these 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 [17] considers an action or event to be a set of sequences of states, and describes a temporal logic for reasoning

[1] Although some authors make a distinction between actions and events, in this paper we treat the two terms synonymously.

about such actions and events. Allen [1] 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, although it is possible to state arbitrary properties of actions and events, it is not obvious how these logics can be used effectively for practical reasoning. [2]

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, to reason about how to achieve given goals or test certain properties of the world, the concept of action alone is not sufficient. In particular, we need to know *how* the actions are actually generated.

To do this, we introduce a notion of *process*. Informally, a process can be viewed as an 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. [3]

Having a notion of process allows us to make a distinction that is critical for practical reasoning – we can distinguish between behaviors that are *successful* executions of the process (i.e., successful instances of the action) and those that are unsuccessful (those that have *failed*). The ability to represent both successful *and* failed behaviors is very important in commonsense reasoning and is critical in multiagent and dynamic environments (e.g., see [13]).

The need for representing both failed and successful behaviors is most clearly seen in problems that involve multiple agents. In such worlds, the potential side effects of an agent's activities can have a dramatic influence upon other agents. In a system that uses a combined planning/execution framework as described earlier, and in which knowledge of the world is incomplete or uncertain, it is usually not possible to predict whether a process will succeed or fail. (Of course, even if we were fully planning everything out in advance, we still could not realistically anticipate all of the consequences of executing a process.) Given this, it is clear that both failed and successful process executions will occur, and thus both must be available for reasoning about potential process interactions.

Using a notion of process is particularly important for reasoning about failed attempts to accomplish given goals. For example, suppose that we have two ways to get to the airport to catch a plane; one involving making a bus connection and the other, although less convenient, by car along a busy route frequented by numerous taxis. Clearly, the successful behaviors of both methods (processes) will result in catching the plane. However, failure to make the bus connection could leave us in a state from which we could not recover (because the next available means of transport to the airport will arrive too late), whereas failure of the other method, given the availability of taxis, would not be so catastrophic. We might, therefore, decide to use the second, less convenient method, if we compare their possible modes of *failure*. And this can only be done with a notion of process – an intrinsic

[2] Allen [2] does, however, propose a method of forming plans that is based on a very restricted form of his interval logic.

[3] In this paper we restrict our attention to sequential, nonconcurrent processes. Our work on implementing a system based on this theory, however, has incorporated the notion of concurrently active, communicating processes.

part of deducing a failed behavior is knowing exactly how that behavior was generated in the first place.

The need for representing failed behaviors also arises in natural-language understanding. For example, it is important to have a denotation for action sentences (such as "she was painting a picture") that allows for action failure, even in midperformance ("she was painting a picture when her paints ran out"). The action referred to in the second sentence must be one of the failed behaviors of the picture-painting process, and there is no way to derive this solely from a set of successful picture-painting behaviors.[4]

Finally, the notion of process failure also allows us to represent tests on world states in a particularly simple way without the introduction of knowledge or belief structures (cf. [18]). To do this, we let certain successful process behaviors stand as tests for a given condition. This can only be done if we are guaranteed that the process will only succeed when the condition is indeed true. (If the process fails, we can, of course, assume nothing about the condition. Although we might usually be happy to equate process failure with the negation of the condition being tested, we may not always wish to do so. In such cases, we might need one process to test for a given condition and another process to test for its negation.)

3 Process Descriptions

Abstractly, a process can be modeled by two sets of behaviors, one set representing the successful behaviors of the process and the other the failed behaviors. However, to reason about these behaviors we need some means for describing them; moreover, whatever descriptions we use need to be amenable to efficient reasoning techniques.

In this section we present a means of describing processes as sequences of particular *subgoals* or behaviors to be achieved. Each process is represented by a labeled transition network with distinguished start and finish nodes and arcs labeled with subgoal descriptions (see Figure 1). Any realizable behavior that achieves each of the subgoals labeling some path through the network from the start node to a final node constitutes a successful behavior of the process; a failed behavior is one which terminates with failure to achieve a subgoal on the path.

Operationally, we may view a process description as being *executed* in the following manner. At any moment during execution, control is at a given node n. An outgoing arc a may be transitted by *successfully* executing a process that achieves the subgoal (behavior) labeling a. If none of the outgoing arcs from n can be transitted, process execution *fails*. Execution begins with control at the start node and *succeeds* if control reaches the finish node.

[4]The reason is that different processes or procedures can generate the same set of successful behaviors yet have different failure modes. For example, consider two procedures for jumping across a stream. In the first procedure, we check that the stream is sufficiently narrow to jump and, if so, jump it; otherwise, we do not even attempt to cross. In the second procedure we perform the same test but attempt the jump irrespective of the outcome of the test. Assuming the test accurately determines whether or not the stream can be jumped, both procedures yield the same set of successful behaviors, but the failed behaviors are quite different!

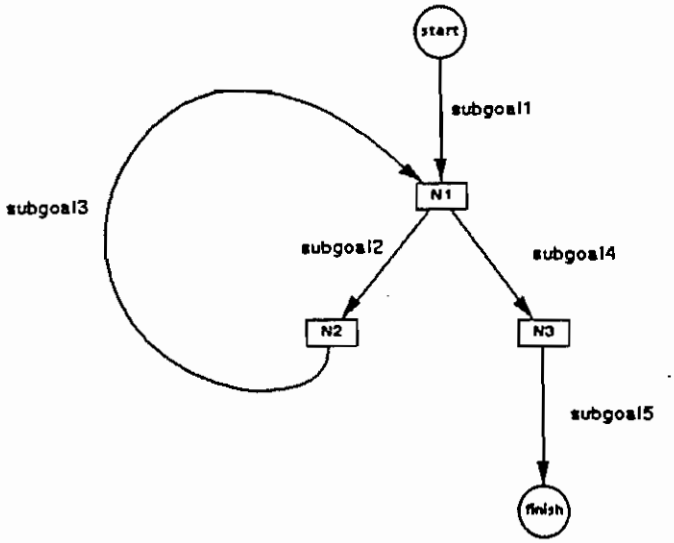


Figure 1: A Process Description

We now give a more formal definition of our process description language. First, we assume a fixed set S, possibly infinite, of *state descriptions* and a fixed set G, also possibly infinite, of *action descriptions.*

A *process description* can then be represented as a tuple $P = \langle N, E, \delta, n_I, N_F, A, \rangle$, where

- 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 *start node*
- $N_F \subset N$ is a set of *final nodes*
- $A : E \rightarrow G$ associates an action description with each arc; these action descriptions are called *goal descriptions.*

Rather than represent process descriptions in this formal mathematical way, we use a graphical form as typified in Figure 1.

Both the state and action description languages are based on predicate calculus. Each state description is a first-order predicate-calculus formula and can be viewed as denoting a set of states; namely, those in which it is true. For example, a formula of the form ((on a b) $\wedge$ (on b c)) could be used to denote the world states in which block a is on top of block b, which in turn is on top of block c.

An action description consists of an action predicate applied to an n-tuple of terms. Each action description denotes an *action type* or set of *behaviors.* For example, an expression

like (walk a, b) could be taken to denote the set of behaviors in which an agent walks from point a to point b.

It is also desirable to allow a class of action descriptions that relate directly to world states. We thus extend the action description language to include actions that achieve a given state condition p (represented by an action description of the form $(!p)$), actions to test for p (represented as $(?p)$), and actions that preserve p (represented as $(\#p)$). We call these *temporal action descriptions.* In each of these, p is a state description – i.e., a description of the type of state to be achieved, tested for, or preserved. For example, an action that achieves a state in which block a is on block b might be described by a temporal action description of the form (! (on a b)).

Action descriptions may also be combined into *action expressions.* These are composed in the usual way using conjunctive and disjunctive operators. Thus, an action expression of the form $(!p) \wedge (?q)$ denotes an action that both tests for q and achieves p.

Having a means for describing processes, we now need a way to state properties about them. In this paper, we are interested primarily in describing the effects of successful behaviors of a process; that is, we want to be able to express the fact that, under certain conditions, successful execution of the process will result in a certain behavior being achieved. We will call such facts *process assertions.*

A process assertion consists of a *process description*, P, describing a process; a *precondition*, c, denoting a set of world states in which the process is applicable; and an *effect*, g, characterizing the set of successful behaviors the process can actually generate when commenced in a state satisfying c. We will write such an assertion as $c\langle P\rangle g$.

The intent or meaning of this assertion is that any successful behavior of process P whose first state satisfies precondition c will also satisfy the effect g. From an operational viewpoint, if c holds at the commencement of execution of process P, g will be realized by a successful execution of the process.

Process assertions may also use variables. Such variables may appear in the precondition c, in the process description P, as well as in the effect g. We make a distinction between *local* variables (prefixed by %) and *global* variables (prefixed by $). All global variables must have a fixed interpretation over the entire process assertion and are taken to be universally quantified. In contrast, local variables must have a fixed intepretation in the interval of states during which a given arc is transitted, but can otherwise vary. They cannot appear in the preconditions or the effect of a process assertion, and are existentially quantified over the scope of the arc on which they appear. (Local variables are often needed in loops where it is necessary to identify different elements from one iteration to next). [5]

A typical process assertion is shown in Figure 2.

[5] In fact, because we want to allow local variables to denote different objects on different *transitions* of the *same* arc, we strictly have to interpret variables with respect to the underlying tree structure of a process obtained by "unwinding" all loops appearing in the process description.

Precondition: TRUE

Effect: (! (DEFEATED $GIANT))

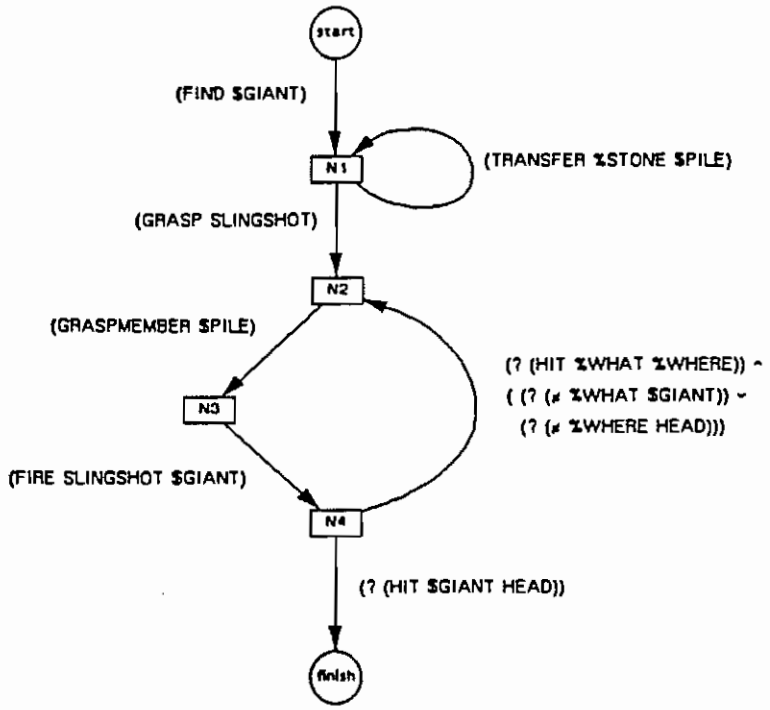


Figure 2: David and Goliath

4 Declarative Semantics

The declarative semantics of process assertions 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 can be used to achieve particular goals — rather, it simply allows one to state *facts* about certain behaviors. In the preceding section we provided a preliminary, intuitive meaning for process assertions. In this section we present a more formal declarative semantics. First, however, we begin with a closer examination of the process assertion depicted in Figure 2.

The "David and Goliath" procedure can be viewed as a plan for defeating a giant with a slingshot. The procedure involves gathering stones, placing them in a pile, getting a slingshot, and then repeatedly taking up a stone and shooting it until the giant is hit on the head. In this particular domain, hitting a giant on the head with a stone hurled by a slingshot always results in the giant's defeat. 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 cannot be accomplished (and no other alternative path can be taken).

It is important to note how the process assertion captures *implicit* knowledge of the problem domain. This knowledge is of two kinds: one concerning the validity of the procedure, the other heuristic. For example, hitting giants on the head with an object propelled from a slingshot will not always defeat them (e.g., if it is a cotton ball), but will if it is a stone. Thus, the validity of the effects of the procedure depends critically on the structure of the procedure itself, 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 test that precedes 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.

At first glance, it seems that the semantics of a process description could be determined solely on the basis of successful behaviors which satisfy each of its subgoals. On closer examination, however, it becomes clear that this will not quite do. For example, if a node has multiple outgoing arcs (such as nodes $N1$ and $N4$ in Figure 2) 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. Similarly, in many real-world situations, it is often desirable to allow multiple attempts to achieve a goal before relinquishing that goal (for example, if a stone is accidently dropped when trying to pick it from the pile). The problem with failed attempts, however, is that they may change the state of the world. Thus, to obtain a proper semantics, paths through a process network must allow behaviors that explicitly include failed attempts at realizing tests and actions as well as successful ones.

We now give a more formal definition of the semantics of process assertions. We first need to specify the interpretation for the symbols appearing in our description language. We will assume a fixed domain D of objects and a possibly infinite set of states. Given a particular state, a *state interpretation* associates with each constant symbol and variable an object from D, with each predicate symbol a relation over D, and with each function symbol a function on D. The meaning of a given state description is then defined under the usual semantics for first-order predicate calculus.

Similarly, we can define a *behavior interpretation* that associates a set of behaviors with each action predicate. The meaning of an action predicate is then taken to be the corresponding set of behaviors in the interpretation of that predicate. We also need to specify the meaning of temporal action descriptions. If p is a state description, then

- $(!p)$ denotes those behaviors whose last state satisfies p.
- $(?p)$ denotes those behaviors whose first state satisfies p.
- $(\#p)$ denotes those behaviors all of whose states satisfy p.

Conjunction and disjunction of action descriptions denote behavior-set intersection and union, respectively.

Finally we are in the position to give a meaning to process descriptions. Each process description will be taken to denote a set of *successful* behaviors *and* a set of *failed* behaviors. To build a description of these behaviors, we first introduce the notion of process applicability. A process P is said to be *applicable* to a goal (i.e., an action type) B for a set of states S, if every behavior in the success set of P that begins in a state $s \in S$ is also in B.

Now let n be a node in a process description P. An *allowed behavior* starting at node n is a sequence composed of behaviors of processes applicable to the goals labeling the arcs emanating from n. Each allowed behavior represents a series of attempts by applicable processes to transit an arc leaving n, until one succeeds or they all fail. Thus, the set of allowed behaviors starting at a node n can be partitioned into two sets: those representing successful transits to a succeeding node, and those that represent failures to leave the node. The first set is denoted by $succ(n, a)$. Each of its behaviors must be a sequence of zero or more unsuccessful attempts by processes applicable to goals on arcs emanating from n, followed by a behavior of an applicable process that succeeds for some arc a. The second set, $fail(n)$, consists of the null behavior along with those behaviors composed only of failed attempts of applicable processes.

Given these two types of allowed behaviors, we can then recursively define the *success* and *failure sets* for a node n, denoted $S(n)$ and $F(n)$ respectively, as follows: [6]

1. If n is a final node, then $S(n)$ and $F(n)$ are both empty.
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.

As an example, consider the process networks shown in Figure 3 where the arcs are labeled with applicable processes. For a process A, let A denote the set of its successful behaviors, and A_F the set of its failed behaviors. Then the success and failure sets for each of the process networks in Figure 3 may be described as follows:[7]

$P1$:	$(A_F)^*.A.(B_F\|C_F)^*.B$	$P1_F$:	$(A_F)^+$
	$(A_F)^*.A.(B_F\|C_F)^*.C$		$(A_F)^*.A.(B_F\|C_F)^+$
$P2$:	$(A_F)^*.A.(B_F)^*.B$	$P2_F$:	$(A_F)^+$
	$(A_F)^*.A.(C_F)^*.C$		$(A_F)^*.A.(B_F)^+$
			$(A_F)^*.A.(C_F)^+$

[6] 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.

[7] The notation used is that for standard regular expressions. The symbols * and + denote zero or more and one or more repetitions, respectively. The symbol | is used to denote a choice between alternatives, e.g., $(A|B)$ denotes a behavior of form A or B.

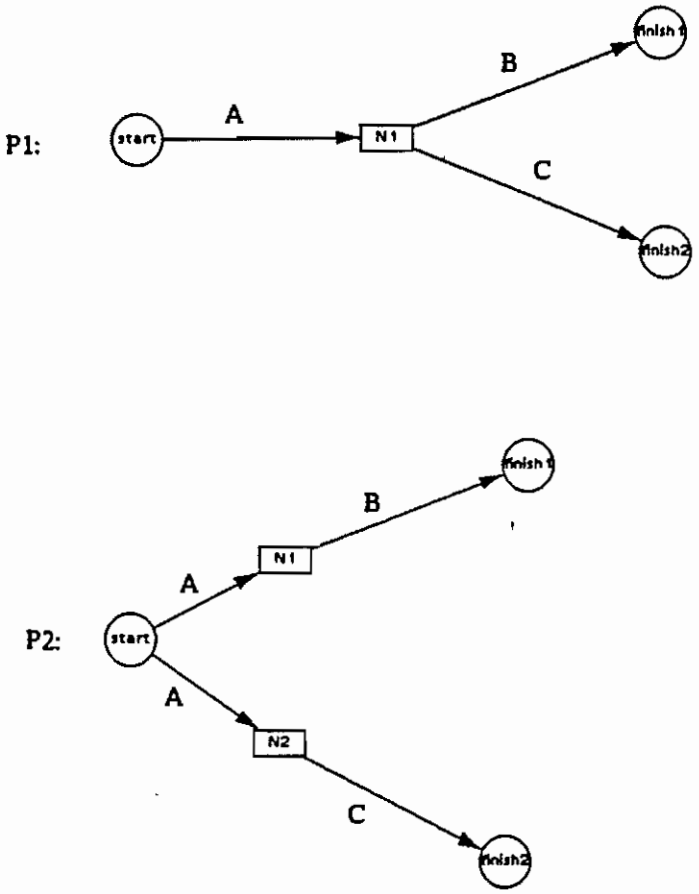


Figure 3: Sample Process Networks

Notice that, while the semantics given above allows for multiple attempts to achieve the goals exiting any given node, it does not allow for backtracking to previous nodes in the net.

Now that we have given an interpretation for process descriptions, we can specify the meaning of a set of process assertions. Consider a process assertion $c\langle P\rangle g$. This assertion is actually a requirement of the following form: for each behavior b in the success set denoted by P, if the first state of b satisfies c, then b satisfies g. A set of processes is a *model* for a set of process assertions (or *satisfies* a set of process assertions) if and only if all process assertions are true of the processes.

5 Operational Semantics

Process assertions 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.

One way to view the causal connection between reasoning and action is as an *interpreter* that takes goals as well as knowledge about the state of the world as input and, as a result, forms intentions to perform certain actions and then acts accordingly. An abstract representation of such an interpreter may be considered to be the *operational semantics* of the knowledge representation language. In this section we provide a description of such an interpreter.

To ground our interpreter in some executable framework, we must make certain assumptions. First of all, 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 containing certain *primitive processes* capable of activating various external effectors. The system must also be able to sense the world to the extent of determining the success or failure of primitive processes – indeed, this is the only way it can sense the state of the world.

Given these capabilities, the system tries to achieve its goals by applying the interpreter given in Figure 4 to applicable processes.[8] 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, the interpreter unifies the corresponding arc assertion with the effects of the set of all process descriptions, and executes a set of the unifying processes, one at a time, until one terminates satisfactorily. If there are no matching processes, or none of the matching processes on any of the outgoing arcs are successful, the execution of P fails. Note that the precondition of each process must be satisfied when it is applied, in order for it to be truly applicable.

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-select randomly selects an element from a set. The order in which selections are made is called the *selection rule*. We call the rule governing the number of times a process may be tried the *application rule* (for this particular interpreter, the application rule is embodied in the function processes-that-unify).[9] The function return returns from the enclosing function, not just the enclosing do. The system starts by executing a process description with a single arc labeled with the initial goal.

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. The operational semantics also defines a set of behaviors for each process, but this set depends on the selection and application rules used in the above algorithm. Let P_D be the set of

[8] This interpreter is very similar to the parsers and generators used for Augmented Transition Networks [28]. It differs in the amount of backtracking allowed and in the use of unification to match arc labels and networks.

[9] In a practical implementation of the operational semantics, it is usually best to use an application rule that tries each matching process exactly once. This allows the realization of all the control constructs of standard programming languages while meeting reasonable bounds on time resources. However, variations in which each process is tried multiple times could be incorporated without conflicting with the declarative semantics of process characterizations.

```
function successful (P n)
  if (is-end-node n) then
    return true
  else
    arc-set := (outgoing-arcs n)
    pr-a-set := (processes-that-unify arc-set)
    do until (empty pr-a-set)
      pr-a := (randomly-select pr-a-set)
      pr := (process pr-a)
      a := (arc pr-a)
      if (satisfied (precondition pr)) then
        if (successful pr (start-node pr)) then
          return (successful P (terminating-node a))
      pr-a-set := (processes-that-unify arc-set)
    end-do
    return false
end-function
```

Figure 4: Abstract Interpreter

successful behaviors for a process P as given by the declarative semantics, and let $P_{O,R,A}$ be the set of successful behaviors for P as given by the operational semantics for selection rule R and application rule A. It is not difficult to show that $P_{O,R,A} \subset P_D$.

This means that any behavior generated by the interpreter given above will satisfy the declarative semantics. The proof involves showing that both the success set and failure set of a process under the operational semantics are each a subset of the success set and failure set, respectively, of the process under the declarative semantics. This can be done using double induction, first, on the number of processes that are applied at a node, and second, on the length of a particular path through the process (where length is measured in number of nodes in the path). The proof is straightforward once it is recognized that any path resulting from use of a selection rule R and an application rule A will automatically be one of the paths covered by the declarative semantics, and that any sequence of process attempts (as well as primitive actions) will be considered successful (or a failure) both declaratively and operationally.

Note that we have made no assumptions about whether a process will succeed or fail – this is determined solely by the environment. As discussed earlier, in the real world the success or failure of processes simply cannot always be predicted. Thus, the above interpreter must be embedded in an environment in order to be truly useful. Without this, the operational semantics given above would be of little interest: it would produce just one possible success set for a given process or goal without any expectation that this behavior could be realized. However, because the interpreter is actually operating using a mixed planning/execution strategy, the environment itself determines process success or failure.

This is quite different from standard AI planning systems, where success of primitive actions is assumed. It is also quite different from the operational semantics of pure Prolog, though would be similar to a semantics for Prolog with input and output streams.

Of course, if we did have additional knowledge about the state of the environment and the success or failure of the primitive actions, we could use the above interpreter in a pure planning mode. However, the inclusion of $P_{O,R,A}$ in P_D would be, in general, 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. This is partly because the interpreter fixes the selection rule and application rule. Even by allowing all possible selection and application rules, however, we would still not attain completeness. The problem is that the interpreter does not have the machinery to deduce facts about world state that can be inferred using the declarative semantics. If an interpreter were capable of deducing all possible behaviors of a process from its description, and if it could also arbitrarily combine processes to generate any achievable behavior, that interpreter would also be able to generate any behavior in P_D. It is clear that such an interpreter would be extremely difficult (if not impossible) to construct. However, in the next section we provide a limited set of proof rules for deducing facts about process behaviors as well as for combining processes to achieve particular effects.

6 Action Decomposition Rules

As described above, the operational semantics we have provided is actually not as strong as it could be. For example, if an arc is labeled with a goal of the form $(!(p \vee q))$, we can determine from the declarative semantics that a process with effect p (or effect q) will be applicable (assuming its preconditions are satisfied). The interpreter given above, however, cannot make this determination.

One way of strengthening our interpreter is to provide it with additional proof rules about the behavior of processes. For example, we might use standard rules of logic along with proof rules such as the following:

$$c1\langle P\rangle g1 \wedge c2\langle P\rangle g2 \equiv (c1 \wedge c2)\langle P\rangle(g1 \wedge g2)$$
$$c1\langle P\rangle g1 \vee c2\langle P\rangle g2 \supset (c1 \wedge c2)\langle P\rangle(g1 \vee g2)$$

We can also devise additional rules for *combining* processes. As before, we use the notation $c\langle P\rangle g$ to mean that every successful behavior of P whose first state satisfies c also satisfies the temporal assertion g. However, we extend the notation to failed behaviors as well, using assertions of the form $c\langle P\rangle_F g$ to describe the effects of failed behaviors. The symbols ";" and "|" represent sequential composition and [nondeterministic] branching, respectively.

Conjunctive Testing

$$\frac{c\langle P_1\rangle(?p \wedge !c' \wedge (\#q \vee \#(\neg q))) \quad c'\langle P_2\rangle(?q)}{c\langle P_1 \,;\, P_2\rangle\,(?(p \wedge q))}$$

Conjunctive Achievement

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

Disjunctive Testing

$$\frac{\begin{array}{c} c\langle P_1\rangle(?p) \\ c\langle P_2\rangle(?q) \\ c\langle P_1\rangle_F((\#q \vee \#(\neg q)) \wedge !c) \\ c\langle P_2\rangle_F((\#p \vee \#(\neg p)) \wedge !c) \end{array}}{c\langle P_1 \mid P_2\rangle(?(p \vee q))}$$

Disjunctive Achievement

$$\frac{\begin{array}{c} c\langle P_1\rangle(!p) \\ c\langle P_2\rangle(!q) \\ c\langle P_1\rangle_F(\#c) \\ c\langle P_2\rangle_F(\#c) \end{array}}{c\langle P_1 \mid P_2\rangle(!(p \vee q))}$$

The above rule for disjunctive achievement, for example, can be read as follows: if process P_1 achieves p when begun in state c, and if process P_2 achieves q when begun in state c, and if, in addition, failures of processes P_1 and P_2 leave c intact, then processes P_1 and P_2 can be combined into a disjunctive process that generates a behavior of the form $(!(p \vee q))$ when begun in state c.

To obtain a new operational semantics that incorporates these proof rules, the interpreter provided in the previous section would have to be modified to allow application of the proof rules when necessary.

7 System Description

We now describe a procedural reasoning system (PRS) based on the theory described in the previous sections. It goes beyond the theory in several ways, including the addition of a data base of heliefs and an enhancement of the way procedures can be invoked. These additions enable the system to exhibit not only goal-directed behavior, but behavior that is *reactive* to particular situations.

The PRS system also makes extensive use of world states and actions that refer to an agent's internal cognitive components. These "metalevel" states and actions are manipulated by the system in the same way it handles states and actions that deal solely with the outside world. They enable the system to form goals or react to situations that deal with the system's internal workings – for example, to figure out how to choose between multiple applicable procedures for a particular task, how to establish new desires, beliefs, or intentions based on particular situations, and so on. All of this metalevel reasoning, however, is done within the same formal context in which reasoning about the external world is done – i.e., in the context of the formalism presented in this paper.

The overall structure of a procedural reasoning system is shown in Figure 5. The system consists of a *data base* containing currently known *facts* (or beliefs) about the world, a set of current *goals* (tasks) to be accomplished, a set of *process assertions* (plans) that describe procedures for achieving given goals or reacting to particular situations, and an *interpreter* (inference mechanism) for manipulating these components. At any moment in time, the system will also have a *process stack* that contains all currently active processes. This stack can be viewed as the system's current *intentions*. We now look at these components in more detail.

The data base is intended to describe the state of the world at the current instant, and thus contains only state descriptions. Its primary function is to keep track of facts about world state that can be inferred as consequences of process executions. We also use the data base to provide the system with knowledge of the initial world state. A STRIPS-like rule is used for determining the full effects of processes; that is, facts are assumed to remain unchanged throughout a process unless we can infer otherwise. Updates to the data base require the use of consistency maintenance procedures.

As in the preceding sections, goals are represented by action descriptions, and can be viewed as specifying a desired behavior of the system. This view of goals as behaviors is unlike that used by most planning systems. In such systems, goals can only be represented as descriptions of state conditions to be achieved. The scheme adopted here allows us to express a much wider class of goals, including goals of maintenance (e.g., "achieve p while maintaining q true") and goals with resource constraints (e.g., "achieve p without using more than one tool").

As indicated earlier, we have also enhanced the way procedures supplied to PRS may be invoked. Rather than just being applicable to given goals, some procedures become applicable when certain facts become known to the system – i.e., they are *fact invoked*. Fact-invoked procedures are particularly useful for creating a *reactive* system – i.e., one that can change focus in reaction to particular situations. Such procedures are usually associated with some *implicit* goal. For example, a fact-invoked procedure for putting out a fire might become applicable whenever the system notices that there is a fire. Although this procedure does not respond to any explicit goal per se, it actually achieves an underlying implicit goal of all organisms – to stay alive. Within the context of the formalism presented in the preceding sections, a process assertion for a fact-invoked procedures has a precondition that describes the condition under which it is applicable and an effect part that matches all implicit goals of the system. This guarantees that each fact-invoked procedure becomes applicable whenever its precondition becomes true.

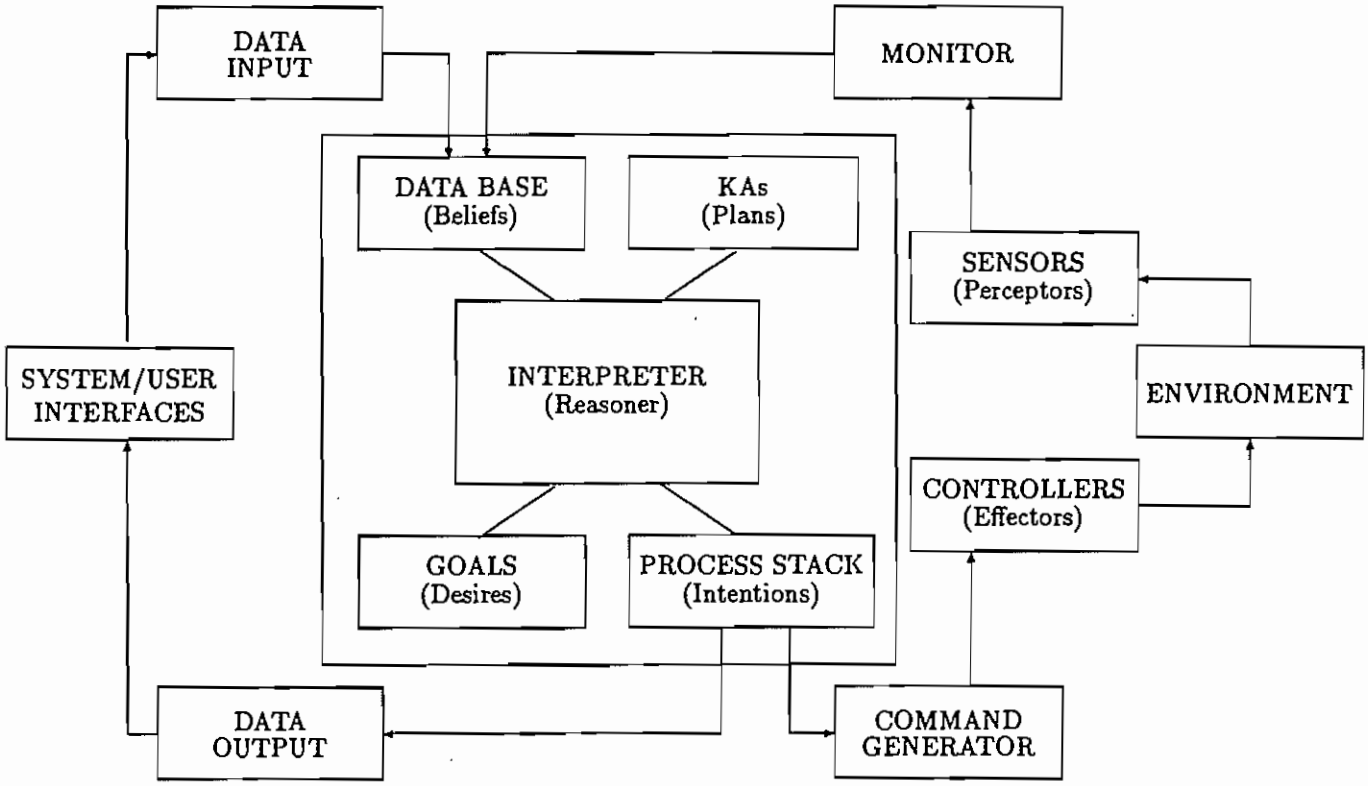


Figure 5: System Structure

The PRS interpreter runs the entire system. From a conceptual viewpoint, it operates in a relatively simple way. At any particular point in time, certain goals are active in the system, and certain facts or beliefs are held in the system data base. Given these goals and facts, a subset of the procedures in the system (both fact invoked and goal invoked) will be applicable. One of these procedures will then be chosen for execution. In the course of transitting the body of the chosen procedure, new goals will be formulated and new facts will be derived. When new goals are added to the goal stack, the interpreter checks to see if any new procedures are relevant, selects one, and executes it. Likewise, whenever a new fact is added to the data base, the interpreter will perform appropriate consistency-maintenance operations on the data base and possibly activate newly applicable procedures.

Because the system is repeatedly assessing its current set of goals, beliefs, and the applicability of procedures, the system exhibits a very reactive form of behavior, rather than being merely goal driven. For example, when a new fact enters the system data base, execution of the current process network might be suspended, with a new relevant process network taking over. One of the ways the system resolves which procedures to execute at any given time is by using other *metalevel* process networks. These metalevel procedures are manipulated and invoked by the system in the same way as any other procedure. However,

they respond to facts and goals pertaining to the system itself, rather than just those of the application domain. For example, one typical metalevel procedure might respond to the metalevel goal (CHOOSE-BEST-PROCESS $GOAL $LIST-OF-PROCEDURES); i.e., *choose* the best procedure to execute from among a given a list of procedures ($LIST-OF-PROCEDURES) that achieve a given object-level goal ($goal) to execute among the various given list of of the form: which we can take to have the . Besides the task of process selection, metalevel procedures can also be used to combine process networks in order to achieve composite goals. For example, metalevel procedures are used in the current system to apply some of the proof rules described in Section 6.

As in the interpreter of Section 5, unification is used to determine whether or not a given process matches a given goal or data base fact. Just as in Prolog, and unlike standard programming languages, this means that it is not necessary to decide before execution which process variables are to count as input variables and which are to count as output variables. This is important for providing flexibility and ease of verification. It also means that variables will not be unnecessarily bound, which can often be advantageous in allowing difficult decisions to be avoided or deferred. For example, if we had a goal of the form (P $X) (where $X is not bound), and a procedure for achieving (P $X) for all objects $X, we could use this procedure for achieving the goal without having to select a particular object to apply it to.

PRS also differs from conventional programming languages in its more flexible means of representing and using procedural information. For example, procedures are not "called" as in standard programming languages. Instead, they are invoked whenever they can contribute to accomplishing some goal or reacting to some situation. Just as procedures cannot be called, neither can they call any other procedure; they can only specify what goals are to be achieved and in what order. This makes PRS procedures much more amenable to modular verification techniques. Another difference that sets PRS apart from standard programming languages is that it is not deterministic. For example, several procedures may be relevant to a goal at any one time, and the order in which they are chosen for execution may, in general, be nondeterministic.

We have implemented an experimental system based on the ideas presented above. The implemented system is written in LISP and runs on a Symbolics 3600 machine. User interaction occurs via a graphical package that allows direct entry and manipulation of process networks.

8 Sample Problem Domains

8.1 Space Shuttle

One area in which advanced automation could be of particular practical benefit is fault isolation and diagnosis in complex systems. PRS is particularly suited to this kind of application not only because a diagnostic domain is dynamic, requiring quick response to faults, but also because much of diagnostic knowledge is specifically procedural in nature. In this section we describe one such application – diagnosis of the Reaction Control System

(RCS) of NASA's space shuttle. The structure of an RCS module is depicted in Figure 6. Sample malfunction procedures from the NASA diagnostic manuals are shown in Figure 7.

The manner in which we have represented procedures for our RCS application reflects what we have said in the preceding sections — i.e., that actions and tests must be represented by whatever condition they achieve or test for, rather than by some arbitrary name. For example, there exist several malfunction procedures for lowering the pressure in tanks that have high pressure readings, and likewise, raising the pressure in tanks with low pressure. Currently, the NASA diagnostic procedures for these tasks make explicit calls to particular procedures that raise and lower tank pressure. Whenever the desired methods for altering tank pressure change, these procedure calls also have to be explicitly changed.

Our methodology for invoking procedures obviates the need for such changes; goals to lower or raise tank pressure may be posted as such – any applicable procedure may then respond. In fact, for this particular example, the goal underlying the pressure alterations may simply be to "normalize" the pressure of the tank. Invoking procedures on the basis of desired effects results in a much more modular and useful way of constructing large systems. Given a set of procedures associated with the actual goal that they achieve, the procedures may be reused in other circumstances in which they might be useful, or easily replaced by other procedures that achieve their particular goal in a better way.

8.1.1 The RCS Data Base

Our first task in encoding the RCS application was to capture the structure of the physical RCS system(depicted in figure 6) as a set of data base facts. 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.

```
(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)
```

Two types of structural facts have been used for this application – TYPE facts, which declare specific components or subsystems and assign to them unique identifiers, and PART-OF facts, which state which components are part of which subsystems. For example, (TYPE RCS F RCS.1) declares the entire front RCS and assigns it the identifier RCS.1. Each RCS contains two helium pressurization subsystems, one for the oxidant part of the system, the other for the fuel subsystem. For the system RCS.1 these are labeled as HEP.1.1 and

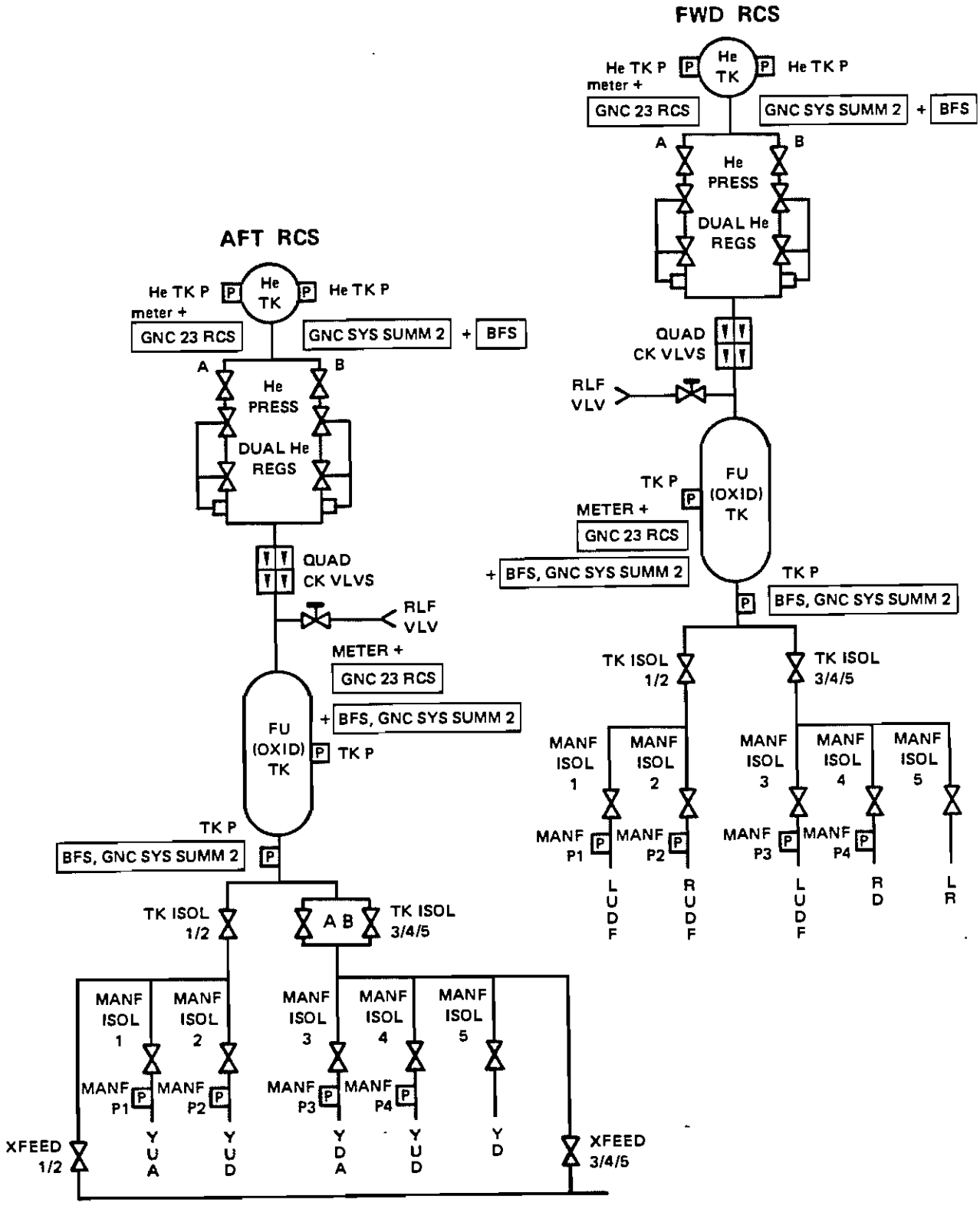


Figure 6: The Reaction Control System

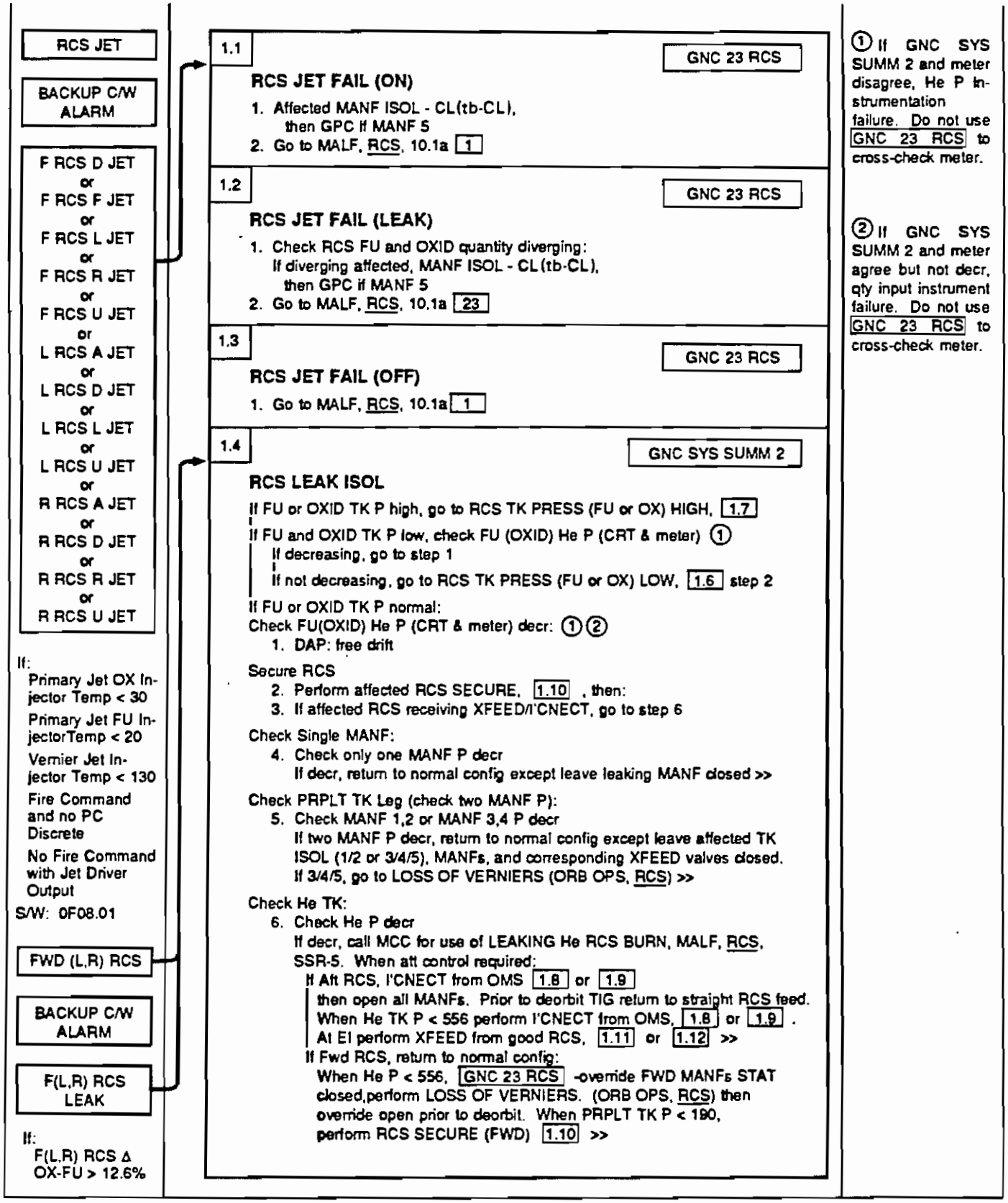


Figure 7: Some RCS Malfunction Procedures

HEP.1.2 respectively. Finally, each helium pressurization system contains its own helium tank.

Once we encode the structure of the RCS in this fashion, the diagnostic procedures can make use of this information to perform what might be considered simple commonsense tasks for an astronaut. For example, if a malfunction procedure has the test "Is the oxidant helium tank pressure greater than the fuel helium tank pressure for the front RCS?", the test can be represented in a way that is impervious to system reconfiguration, is not hard-wired to particular identifiers, and can be used for any RCS. This is done using unification – matching data base facts against queries composed as logical combinations of atomic formulas. In this case, the query would have the following form:

```
(?  ((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)))
```

8.1.2 JET-FAIL-ON Process

We will now concentrate 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 (a portion of the entire malfunction procedure is shown in Figure 8). Notice how diagnostic conclusions (such as "JET DRIVER FAILED-ON ELECTRICALLY") are displayed in highlighted boxes.

PRS uses several processes to implement this diagnostic procedure. The main top-level process for dealing with the "JET FAIL (ON)" failure is called JET-FAIL-ON and is shown in Figure 9. This process is fact invoked – that is, it responds when the system notices that certain lights, alarms, and computer monitor readings appear. Its precondition has the form:

```
(LIGHT RCS-JET) ∧ (ALARM BACKUP-CW) ∧
(FAULT $rcs-id RCS $jet-id JET) ∧
(JETFAIL-INDICATOR ON $manf-id)
```

In order to get JET-FAIL-ON 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:

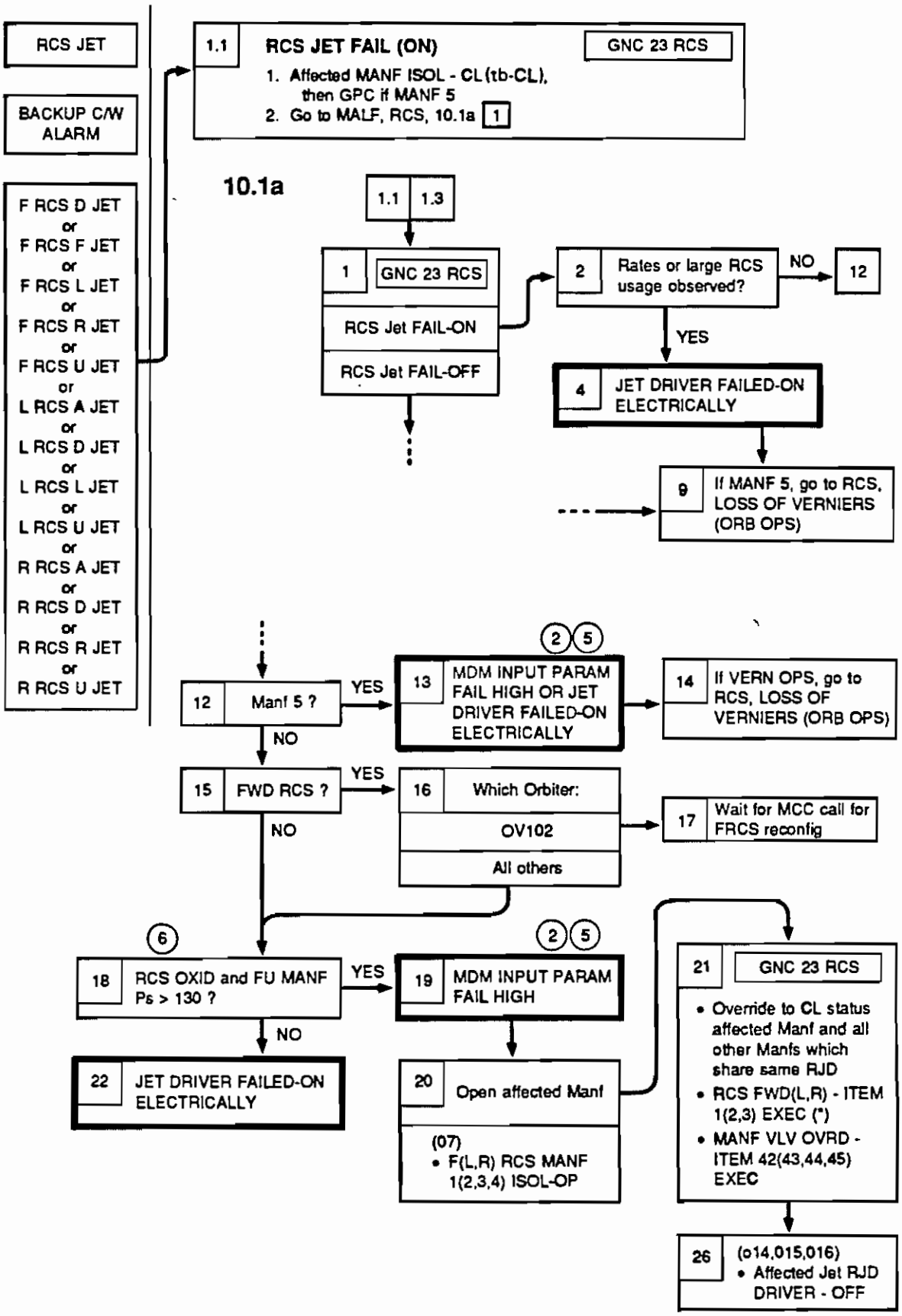


Figure 8: RCS JET FAIL (ON) Malfunction Procedure

Precondition: (LIGHT RCS-JET) ∧ (ALARM BACKUP-CW) ∧
(FAULT $RCS-ID RCS $JET-ID JET) ∧
(JETFAIL-INDICATOR ON $MANF-ID)

JET-FAIL-ON

Effect:

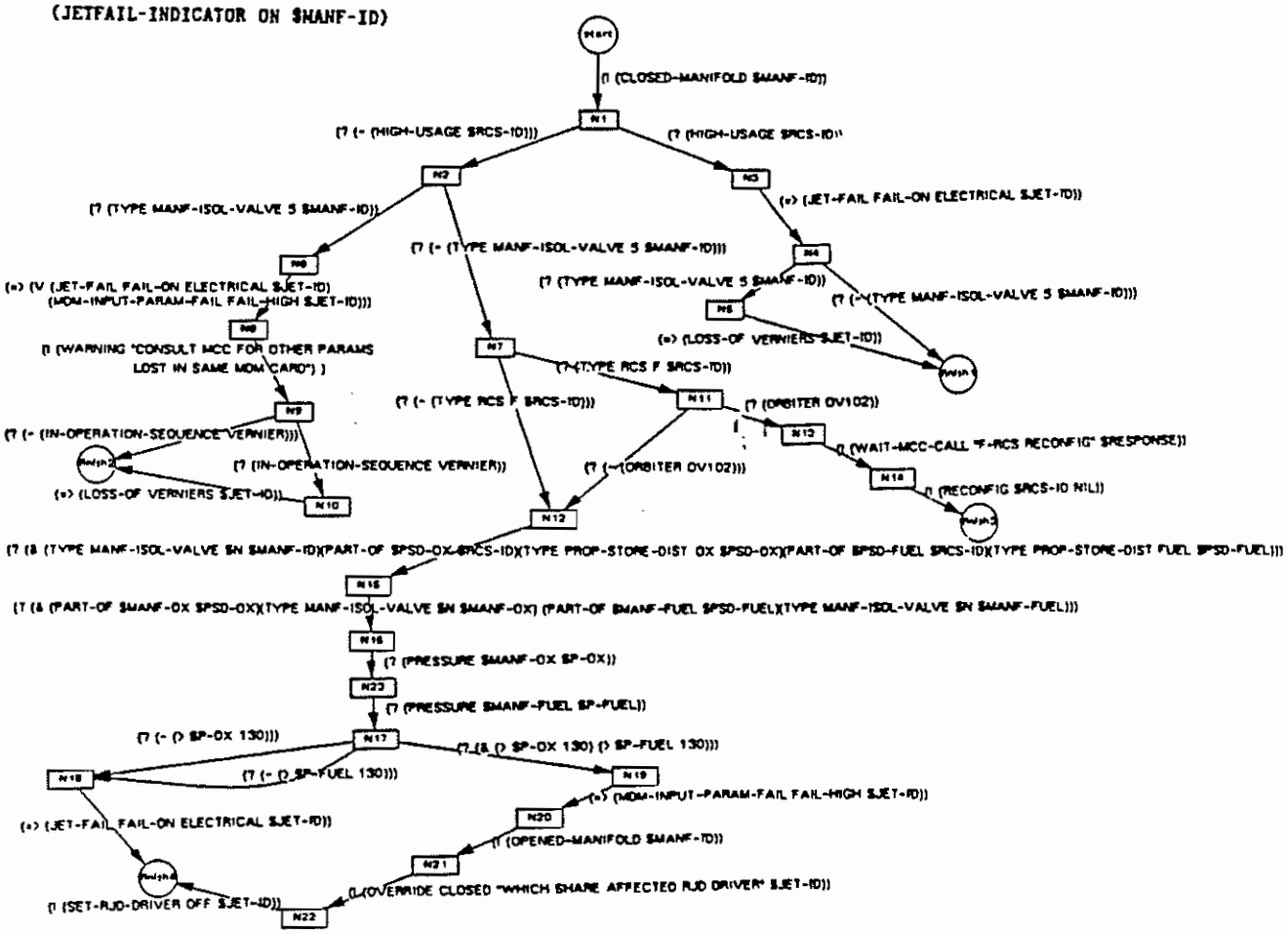
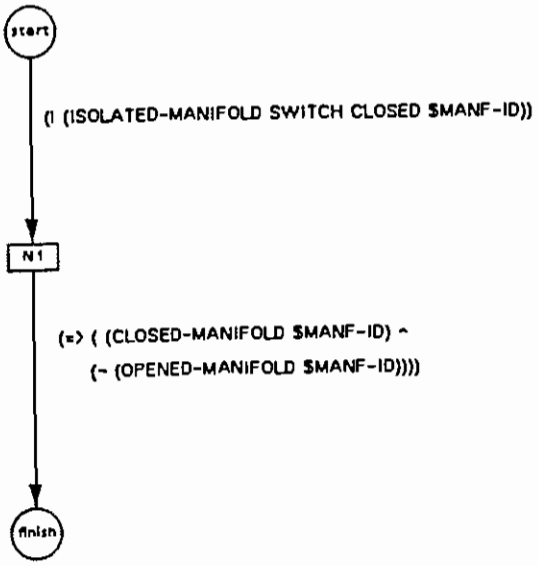


Figure 9: JET-FAIL-ON Process

Precondition: (TYPE MANF-ISOL-VALVE $N $MANF-ID) ∧ (≠ $N 5)

Effect: (! (CLOSED-MANIFOLD $MANF-ID))

CLOSED-MANIFOLD



Precondition: (TYPE MANF-ISOL-VALVE 5 $MANF-ID)

Effect: (! (CLOSED-MANIFOLD $MANF-ID))

CLOSED-MANIFOLD-VERNIER

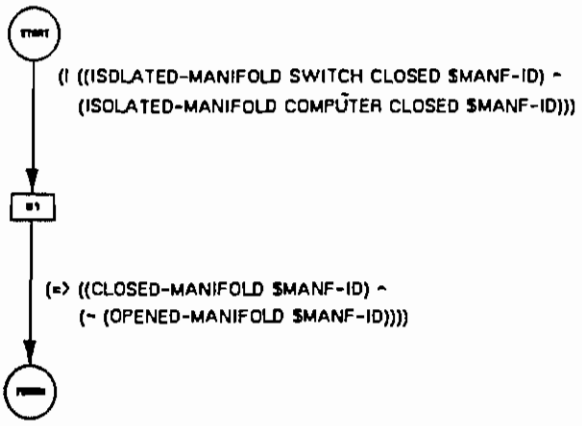


Figure 10: Processes For Closing A Manifold

```
(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 RCS subsystem, jet, and manifold. The system will then react and apply the JET-FAIL-ON process.

Starting at its START node, JET-FAIL-ON will proceed and try to traverse its first edge, labeled with the goal expression (! (CLOSED-MANIFOLD $manf-id)). 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 8, which reads: "Affected MANF ISOL - CL (tb-CL), then GPC if MANF 5." Notice how we have abstracted the overall *goal* or *intent* of this step (to close the manifold) from a particular instruction in the malfunction book which only states *how* to achieve the goal.

In this case, there are actually two different ways of achieving a closed manifold: for all manifolds, a talk-back switch is set to the closed position, but for vernier manifolds (type 5 manifolds), 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 procedures shown in Figure 10, CLOSED-MANIFOLD and CLOSED-MANIFOLD-VERNIER. Each responds to a goal of the form (! (CLOSED-MANIFOLD $manf-id)). However, their preconditions 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 situations in which more than one processes is truly applicable to a given goal, metalevel processes are used to resolve which is most useful.

Of course, given the semantics of process assertions, there is yet another way to achieve (! (CLOSED-MANIFOLD $manf-id)). In particular, a goal of the form (! P) will automatically be achieved if the system already believes that P is true. For this case, if the system already has in its data base a fact of the form (CLOSED-MANIFOLD miv.1.1.1), a goal of the form (! (CLOSED-MANIFOLD miv.1.1.1)) will automatically succeed – no executions of the processes CLOSED-MANIFOLD and CLOSED-MANIFOLD-VERNIER need be undertaken.

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

We continue now with one more step in the execution of the JET-FAIL-ON process. If the goal to close the manifold 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., our goal is to determine whether there is *not* high usage in the affected RCS. To handle a goal of this form the system will first check to see

if there are any data-base facts or processes that match this goal precisely. 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 (for the sake of argument we have assumed that $rcs-id is bound to rcs.1). Similarly, there may be a process 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 process is found, then the system will attempt to satisfy the goal in these ways. However, if no such fact or matching process is present, the system will try to achieve the goal using any other means at its disposal.

For goals composed of certain negated predicates, a metalevel process is available which tries to achieve the goal using the "negation as failure" rule [15]. In other words, for a goal of form (! (¬ P)) or (? (¬ P)), the metalevel process will try to achieve (! P) (or (? P)), and if it fails to do so, will assume that the original negated goal has succeeded. In our current system, this is precisely how the goal (? (¬ (HIGH-USAGE $rcs-id))) is handled. Other metalevel processes also exist for achieving a conjunct of goals or a disjunct of goals.

8.2 Autonomous Robot

A second interesting application of PRS is in the control of autonomous robots. Our experimentation in this domain is being done using SRI International's new robot, Flakey. To effectively tackle this problem, we had to use multiple, concurrently active PRS modules, each consisting of its own data base, goal stack, and processes that monitor and control different aspects of the robot's activity.

As our objective, we envisaged the robot in a space station acting as an astronaut's assistant. When asked to get a wrench, for example, the robot works out where the wrench is kept, plans a route to get it, and goes there. If the wrench is not there the robot reasons further about how to obtain information on its whereabouts, and finally returns to the astronaut with the wrench or explains why it could not be retrieved. In another scenario, the robot may be in the process of retrieving the wrench when it notices a malfunction light for one of the jets in a RCS module of the space station. It reasons that this is of higher priority than retrieving a wrench and sets about diagnosing the fault and correcting it. After having done this, it continues with its original task, finally telling the astronaut what has happened.

To accomplish these tasks the robot must not only be able to create and execute plans, but must be willing to interrupt or abandon a plan when circumstances demand it. Since other agents can move obstacles and issue demands even as the robot is planning, and since its view of the world can change as fast as the robot itself is moving, performance of the task requires a robot which is perceptive and highly reactive as well as goal directed.

The way we have structured the processes for this domain has actually conformed somewhat to Brooks' notion [3] of a *vertical* decomposition of robot functions (in contrast to the traditional *horizontal* decomposition into functional modules). The top level robot module is used to perform higher level cognitive functions: overall route planning and high-level guidance. The lower the level of a module, the more primitive its function. Below the highest level are modules which put together sonar sensory data and figure out where "walls"

and "doors" are. Even lower level modules reactively monitor the more rudimentary aspects of the navigation process – reacting to obstacles, maintaining a parallel bearing to the wall, getting back on course when veering takes place, etc.

Our present version of the robot application system is more fully described elsewhere [10]. Currently, the robot's model of the external world is particularly simple: apart from toplogical knowledge about hallways and rooms, the beliefs of the robot consist solely of its most recent sonar readings, various velocities and accelerations, and some indicators regarding the status of simulated external systems (such as the RCS module). Of course, any realistic application of the system would require that the robot be capable of building and storing much more complex models of the world around it.

9 Conclusions

We have presented a model for action and a means for representing knowledge about procedures. The importance of reasoning about *processes* rather than simple histories or state sequences was stressed. In particular, we have indicated the role that process *failure* plays in practical reasoning.

A declarative semantics for the representation was provided that allows a user to specify facts about processes and their behaviors. This semantics is important for providing a model-theoretic basis to the knowledge representation. 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. A critical feature of the interpreter, and one that distinguishes it in kind from most existing AI planners, is that it is situated in an environment with which it interacts during the reasoning process. We consider the partial hierarchical planning that results to be an essential component of effective practical reasoning.

The knowledge representation we have described can also be used for symbolic planning in the traditional sense, although we would need to provide additional axioms stating under what conditions primitive processes would be successful. Indeed, the operators of many standard planning systems (such as NOAH [22], DEVISER [26], and SIPE [27]) can be viewed as restricted forms of process assertions.

Our formalism 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 declarative semantics allows *facts* about the behavior of the system to be stated and verified independently. 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.

We have described a practical implementation of a system based on this model, and have shown how it can be applied for fault diagnosis and in the control of autonomous robots in highly dynamic situations. Although we have used parallel instances of PRSs within our implementation, we have yet to extend our formal model to deal with it. Some work in this direction is described by Georgeff [8], and work on synchronizing the activities of multiple agents has been done by Lansky [14] and Stuart [24].

Acknowledgements

There have been a number of people involved in the design, implementation, and testing of PRS, including Pierre Bessiere, Marcel Schoppers, Joshua Singer, and Mabry Tyson. We are most grateful for their contributions.

References

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

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

[3] R. A. Brooks. *A Robust Layered Control System for a Mobile Robot.* Technical Report 864, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1985.

[4] K. M. Chandy and J. Misra. How processes learn. In *Proceedings of the Fourth ACM Symposium on Principles of Distributed Computing*, 1985.

[5] W. F. Clocksin and C. S. Mellish. Programming in prolog. Springer-Verlag, Berlin, 1984.

[6] D. Davidson. *Actions and Events.* Clarendon Press, Oxford, England, 1980.

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

[8] A. L. Lansky Georgeff, M. P. and M. Schoppers. *Reasoning and Planning in Dynamic Domains: An Experiment with a Mobile Robot.* Technical Note 380, Artificial Intelligence Center, SRI International, Menlo Park, California, 1987.

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

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

[11] M. P. Georgeff and A. L. Lansky. *A System for Reasoning in Dynamic Domains: Fault Diagnosis on the Space Shuttle.* Technical Note 375, Artificial Intelligence Center, SRI International, Menlo Park, California, 1986.

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

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

[14] A. L. Lansky. *Localized Representation and Planning Methods for Parallel Domains.* Technical Note 401, Artificial Intelligence Center, SRI International, Menlo Park, California, 1986.

[15] J. W. Lloyd. *Foundations of Logic Programming. Symbolic Computation Series*, Springer-Verlag, Berlin, 1984.

[16] J. McCarthy. Programs with common sense. In M. Minsky, editor, *Semantic Information Processing*, MIT Press, Cambridge, Massachusetts, 1968.

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

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

[19] V. Nguyes and K. J. Perry. *Do We Really Know Knowledge Is.* Technical Note, IBM T. J. Watson Research Center, Yorktown Heights, New York, 1986.

[20] S. J. Rosenschein. *Formal Theories of Knowledge in AI and Robotics.* Technical Report, Artificial Intelligence Center, SRI International, Menlo Park, California, 1985.

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

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

[23] M. Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16(2):111–140, 1981.

[24] C. J. Stuart. *Synchronization of Multiagent Plans Using A Temporal Logic Theorem Prover.* Technical Note 350, Artificial Intelligence Center, SRI International, Menlo Park, California, 1985.

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

[26] S. Vere. Planning in time: windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3):246–267, May 1983.

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

[28] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13, 1970.