

An architecture for real-time reasoning and system control

Félix Ingrand, M.P. Georgeff, A.S. Rao

► **To cite this version:**

Félix Ingrand, M.P. Georgeff, A.S. Rao. An architecture for real-time reasoning and system control. IEEE Expert, Institute of Electrical and Electronics Engineers (IEEE), 1992, 7 (6), pp.34-44. 10.1109/64.180407 . hal-01980099

HAL Id: hal-01980099

<https://hal.laas.fr/hal-01980099>

Submitted on 15 Jan 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Architecture for Real-Time Reasoning and System Control*

François F. Ingrand^{†‡}

LAAS/CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex, France
E-mail: felix@laas.fr

Michael P. Georgeff

Australian Artificial Intelligence Institute
1 Grattan Street,
Carlton, Victoria 3053, Australia
E-mail: georgeff@aaii.oz.au

Anand S. Rao

Australian Artificial Intelligence Institute
1 Grattan Street,
Carlton, Victoria 3053, Australia
E-mail: anand@aaii.oz.au

Abstract

The development of systems capable of handling and diagnosing malfunctions in real time has long been of considerable practical importance. This paper describes the architecture of such a system, called the Procedural Reasoning System (PRS). PRS is based on the notion of a rational agent that can reason and plan under possibly stringent constraints on both time and information. This approach provides the system with the ability to reason in complex ways about dynamic processes, while still maintaining the reactivity required to ensure appropriate responsiveness and control. By considering two large-scale applications in aerospace and telecommunications, it is shown how PRS meets many of the critical requirements for real-time malfunction-handling and diagnostic systems. Finally, PRS is compared with a number of other real-time reasoning and knowledge-based architectures that have been used in similar applications.

1 Introduction

In the process control industries, computer systems for diagnosing and handling plant and process malfunctions are becoming increasingly important. Often, the manner in which these tasks are best carried out does not fall within the domain of conventional control techniques. Instead, they require the application of certain rules or procedures that are specific to particular situations or circumstances. These rules and procedures may contain knowledge about both the diagnosis of the malfunction and possible corrective actions and can be arbitrarily complex.

Systems that reason about and perform these kinds of tasks are called *situated reasoning systems*. A situated reasoning system must be capable of receiving information from a variety of sources (or sensors) in an asynchronous fashion and, from this information and previously acquired knowledge, must assess whether the process under its control is behaving normally. If an abnormality occurs, the system should isolate the fault and determine, if possible, the cause of the problem. It must take appropriate corrective actions to either rectify or contain the problem and must continue to monitor the process to ascertain the effects of its actions.

One of the most difficult—and least considered—problems facing the designers of such systems is how to manage the execution of these rules and procedures under the stringent real-time constraints typical of many process control applications. Where the rules and procedures are relatively simple and independent of one another, execution can be effected in some straightforward manner, such as first-come-first-served or in parallel. However, when these rules and procedures can take an arbitrarily long time to execute, and possibly involve other rules or procedures, it is essential to be able to reason about the *management* of these tasks. This may require reasoning about which tasks need to be performed to realize other tasks (so-called means-ends reasoning), the criticality or urgency of tasks (task utilities), the potential interactions among tasks, the order in which tasks should be performed, and which tasks need to be suspended or resumed given the current state of the system. Most importantly, all this must be done while continuing to attend to the process under control and reacting appropriately to changed circumstances.

One approach to this problem is to base the architecture of the system on the notion of a *rational agent*. Such systems perform execution-time reasoning and planning using explicit representations of the cognitive attitudes usually

*A revised version of this paper has been published in IEEE Expert 7(6) 33-44, December, 1992.

[†]This author is also working at ACS Technologies, Village d'Entreprises, Voie 5, BP 556, 31324 Labège Innopole Cedex, FRANCE

[‡]Part of this work was done while this author was at SRI International, AI Center, 333, Ravenswood Avenue, Menlo Park, CA 94025, USA

associated with human rationality; i.e., attitudes such as belief, desire, and intention. The question for the system designer is to determine the role these attitudes play in governing the rational (effective) behavior for a system required to act in a dynamic world and subject to constraints on both time and information. In particular, the reasoning and planning performed by the system must all be carried out in a continuously changing world, requiring that the system appropriately balance the time taken thinking against the time needed for acting.

In Section 2 we discuss the major requirements for a situated reasoning system to be used effectively in process control applications. In Section 3, we provide an overview of the Procedural Reasoning System (PRS), a generic reasoning system based on the rational-agent architecture described above. Although this reasoning system has been applied to a number of different applications, we choose herein two that are of most relevance to the process industries. The first of these applications involves malfunction handling for the Reaction Control System (RCS) of NASA's Space Shuttle, described in Section 4. The second, described in Section 5, concerns diagnosis and control of failures and overloads in a telecommunications network. In Section 6, we compare our work with other architectures for situated reasoning systems. We conclude in Section 7 by revisiting the general requirements of process control applications and showing how they are met by PRS in the two applications.

It is important to note that the focus of this paper is primarily on the *representation and management* of malfunction-handling procedures and tasks. In particular, we are not concerned with the derivation of these procedures (possibly from reasoning about the causal properties of the domain), nor in distinguishing diagnostic actions from corrective actions. However these procedures are produced (either by engineers, system designers, or automated mechanisms), at the end of the day one is still left with the problem of organizing their execution. It is this problem that we address here.

2 Requirements for the Design of Situated Reasoning Systems

In this section, we describe some of the characteristics that are essential in the design of situated reasoning systems for diagnosis and malfunction handling in process control applications. These characteristics are extensions of those given by Laffey *et. al.* in their evaluation of real-time expert systems [13].

- **Asynchronous event handling:** Given a continuous mode of operation, as in process control, it is unreasonable to assume that events occur at designated times, or that data will be sent to the system at regular intervals of time.
- **Guaranteed reaction and response times:** Because the diagnostic systems for process control are embedded in a continuously changing environment, there are stringent real-time constraints to be met. Important events must be noticed in a timely manner and appropriate actions have to be taken before it is too late. As a consequence, the reasoning system has to provide a guaranteed reaction time (the time taken to react to a situation) and must ensure that the response time (the time taken to respond adequately to a situation) is short enough to allow the response to be properly effected.
- **Procedural representation of knowledge:** In most process control applications, the knowledge required to isolate faults and provide corrective actions is usually represented as procedures or plans. Most maintenance and operation manuals are filled with procedures encoding the different steps to follow in various situations. It is therefore preferable to represent this knowledge as situation-specific *procedures* rather than as a collection of *rules*. This makes the creation and modification of the domain knowledge of the system easier, as well as allowing the system to reason and plan in a timely manner.
- **Handling of multiple problems:** In process control, the occurrence of multiple problems, related or unrelated, is quite common. It is thus necessary that the system be able to handle multiple problems concurrently, possibly by suspending or reorganizing the execution of other ongoing diagnostic or corrective tasks.
- **Reactive and goal-directed behavior:** A situated reasoning system needs to perform its actions with a well-defined goal or purpose, such as isolating a fault or taking a corrective action. At the same time, it also has to be capable of responding to exceptional circumstances (such as alarms) in a timely manner.
- **Focus of attention:** In order to complete specific tasks under the real-time demands of most process control applications, it is necessary that the system be able to focus its attention on these tasks and not be unduly distracted by other events.
- **Reflective reasoning capabilities:** Although focus of attention is important, it is equally important that the system be able to change its focus in response to changes in its environment. This cannot be achieved if the system is not capable of reflecting on its own status, such as the importance of the activity it is currently performing. The system must be able to manipulate the procedures it is currently executing by changing their priority, and suspending and reinvoking them as necessary. In other words, a situated reasoning system needs procedures for controlling the execution of other procedures.

- **Continuous embedded operation:** Unlike diagnostic systems that are used in static domains, systems for process control need to operate continuously.
- **Handling of incomplete or inaccurate data:** As a situated reasoning system will be connected to a variety of different sensors, it is highly likely that the data received from different sources may be inconsistent, inaccurate or incomplete. The system must therefore have some mechanisms for being able to handle such data.

There are many other capabilities that are desirable in process control applications, including:

- **Handling of transients:** Process control systems are required to continuously measure and monitor different control parameters with different response characteristics. Under such situations, transients are common and have to be accommodated.
- **Modeling delayed feedback:** The actions carried out by a situated reasoning system may not take effect immediately; i.e., the feedback from the external environment to the system is often delayed. The system needs to be able to model this delayed feedback by recognizing that its actions will take time to be effected in the real world and at the same time recognizing the need to take alternative remedial actions if its original actions have not had their desired effect.
- **Operator control:** While a situated reasoning system must be capable of autonomous operation, it must also be capable of being overridden by a human operator. Moreover, to enable the operator and the reasoning system to cooperate, the system needs to provide the operator with intelligible trace and explanation facilities and should be capable of accepting useful advice from the operator.

These features are not “requirements” in the sense that they are not essential for all process control applications. Nevertheless, any general-purpose tool suited to process control and malfunction handling should support them.

In the next section we describe the Procedural Reasoning System (PRS), which has been designed to satisfy most of the requirements mentioned above.

3 Description of PRS

The Procedural Reasoning System (PRS) [3, 5] is a generic architecture for representing and reasoning about actions and procedures in a dynamic domain. It has been implemented and applied to various tasks with real-time demands, including malfunction monitoring for different subsystems of NASA’s space shuttle [5], the diagnosis, monitoring and control of telecommunications networks [15, 21], the control of a mobile robot [7], system control for a surveillance aircraft [10], and air-traffic management [14].

The architecture of a PRS module or agent consists of (1) a database containing the system’s current beliefs about the world; (2) a set of current goals; (3) a library of plans (or procedures), called Knowledge Areas (KAs), which describe particular sequences of actions and tests that may be performed to achieve given goals or to react to certain situations; and (4) an intention structure, consisting of a [partially] ordered set of all plans chosen for execution at run-time. An interpreter (inference mechanism) manipulates these components, selecting an appropriate plan (KA) based on system beliefs and goals, placing those selected KAs on the intention structure, and finally executing them.

PRS interacts with its environment both through its database, which acquires new beliefs in response to changes in the environment, and through the actions that it performs as it carries out its intentions. Different instances of PRS, running asynchronously, can be used in an application that requires the cooperation of more than one subsystem.

The contents of the PRS database may be viewed as representing the current beliefs of the system. Some of these beliefs are provided initially by the system user. Typically, these will include facts about static properties of the application domain, such as the structure of some subsystem or the physical laws that must be obeyed by certain mechanical components. Other beliefs are derived by PRS itself as it executes its KAs. These will typically be current observations about the world or conclusions derived by the system from these observations, and these may change over time. For example, at some times PRS may believe that the pressure of an oxidizer tank is within acceptable operating limits, at other times not. Updates to the database therefore necessitate the use of consistency maintenance techniques.

In PRS, the goals are descriptions of desired tasks or behaviors. In the logic used by PRS, the goal to achieve a certain condition C is written as $(! C)$; to test for the condition is written as $(? C)$; to wait until the condition is true is written as $(\wedge C)$; to maintain C is written as $(\# C)$; to assert the condition C is written as $(\Rightarrow C)$; and to retract the condition C is written as $(\sim > C)$. For example, the goal to close valve $v1$ could be represented as $(! (\text{position } v1 \text{ } c1))$, and to test for it being closed as $(? (\text{position } v1 \text{ } c1))$.

Knowledge about how to accomplish given goals or react to certain situations is represented in PRS by declarative procedure specifications called *Knowledge Areas* (KAs). Each KA consists of a *body*, which describes the steps of the procedure, and an *invocation condition*, which specifies under what situations the KA is useful. Together, the

Leak Isolation

INVOCATION:
 (*GOAL (I (LEAK-ISOLATED \$P-SYS)))

CONTEXT:
 (AND (*FACT (MANIFOLD-1 \$P-SYS \$MANF1))
 (*FACT (MANIFOLD-2 \$P-SYS \$MANF2))
 (*FACT (MANIFOLD-3 \$P-SYS \$MANF3))
 (*FACT (MANIFOLD-4 \$P-SYS \$MANF4))
 (*FACT (MANIFOLD-5 \$P-SYS \$MANF5))
 (*FACT (TYPE HE-TANK \$HE-TK))
 (*FACT (PART-OF \$P-SYS \$HE-TK))
 (*FACT (TYPE PROPELLANT-TANK \$PROP-TK))
 (*FACT (PART-OF \$P-SYS \$PROP-TK))
 (*FACT (CONNECTS \$V1 \$PROP-TK \$12-TANK-LEG \$MANF1))
 (*FACT (CONNECTS \$V2 \$12-TANK-LEG \$MANF1))
 (*FACT (CONNECTS \$V3 \$PROP-TK \$345-TANK-LEG))
 (*FACT (CONNECTS \$V4 \$345-TANK-LEG \$MANF3))
 (*FACT (CONNECTS \$V5 \$HE-LEG \$PROP-TK)))

GOAL ACHIEVER?:
 T

EFFECTS:
 NIL

PROPERTIES:
 NIL

DOCUMENTATION:
 "This KA is used to detect the location of a system leak. The system is first secured. One then tries to isolate the leak by testing for either very low or decreasing pressures."

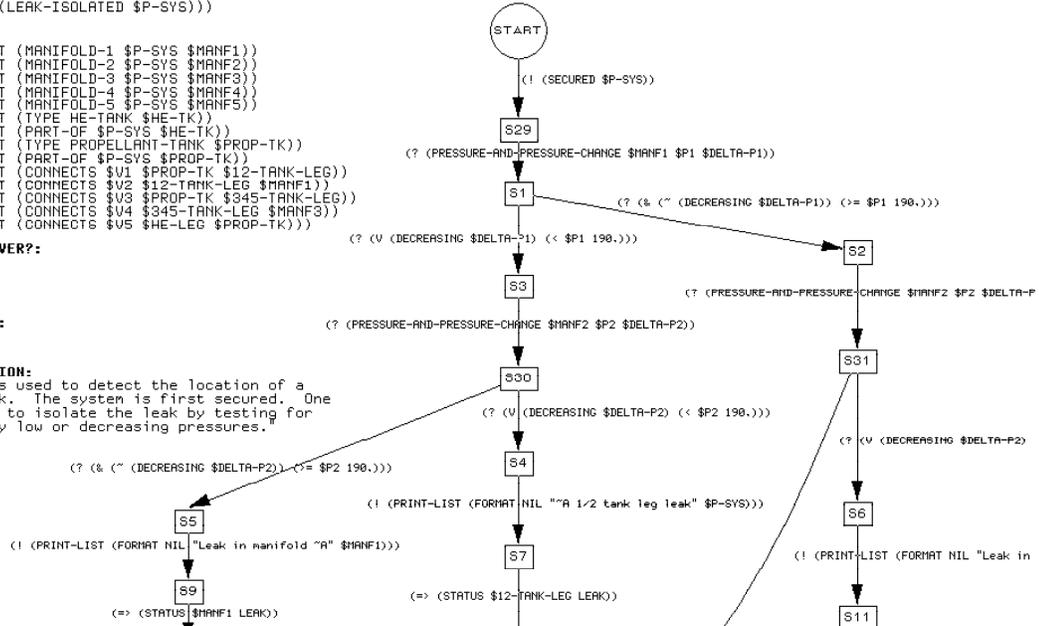


Figure 1: Portion of a KA for Leak Isolation

invocation condition and body of a KA express a declarative fact about the results and utility of performing certain sequences of actions under certain conditions [6].

The body of a KA is represented as a graphic network and can be viewed as a plan or plan schema. Each arc of the network is labeled with a goal to be achieved by the system. The invocation condition has two components: a *triggering part* and a *context part*. Both must be satisfied for the KA to be invoked. The triggering part is a logical expression describing the *events* that must occur for the KA to be executed. Usually, these consist of some *change* in system goals (in which case, the KA is invoked in a goal-directed fashion) or system beliefs (resulting in data-directed or reactive invocation), and may involve both. The context part is a logical expression specifying those conditions that must be true of the current state for the KA to be executed.

A typical example of part of a KA is given in Figure 1. It describes a procedure to isolate a leak in the Reaction Control System of the space shuttle. The invocation part describes under what conditions this KA is useful. In this case, the KA is considered useful whenever the system acquires the goal to isolate a leak in the RCS ($\$p\text{-sys}$), provided the various type and structural facts given in the context part are true. (In determining the truth value of the invocation part, some of the variables appearing in the invocation part will be bound to specific identifiers. Indeed, in this case, all the variables will be so bound.)

The KA body describes what to do if the KA is chosen for execution. Execution begins at the **start** node in the network, and proceeds by following arcs through the network. Execution completes if execution reaches a finish node (a node with no exiting arcs). If more than one arc emanates from a given node, any one of the arcs emanating from that node may be traversed. To traverse an arc, the system must either (1) determine from the database that the goal has already been achieved or (2) find a KA (procedure) that achieves the goal labeling that arc. For example, to traverse the arc emanating from the start node requires either that the system be already secured or that some KA for securing the RCS be found and successfully executed. Similarly, to transit the next arc requires that some KA be found for determining the pressure change ($\$delta\text{-}p1$) in the manifold $\$manf1$. If the system fails to traverse an arc emanating from some node, other arcs emanating from that node may be tried. If, however, the system fails to achieve any of the goals on arcs emanating from the node, the KA as a whole will fail. For example, since only one arc emanates from the start node in Figure 1, if all attempts to secure the RCS fail, this procedure for isolating a leak in the system will also fail. The full KA for this procedure consists of over 45 nodes and is the largest in the RCS application.

Important properties of the KA are represented in the slots on the left side of the KA structure. For example, the **goal achiever** slot is set to **T** (true), representing the fact that, upon successfully completing this KA, the goal that triggered execution will have been achieved.

Some KAs have no bodies. These are the primitive KAs of the system and have associated with them some primitive action that is directly performable by the system. Clearly, execution of any KA must eventually reduce to the execution of sequences of primitive KAs (unless, of course, each of the subgoals of the KA has already been achieved).

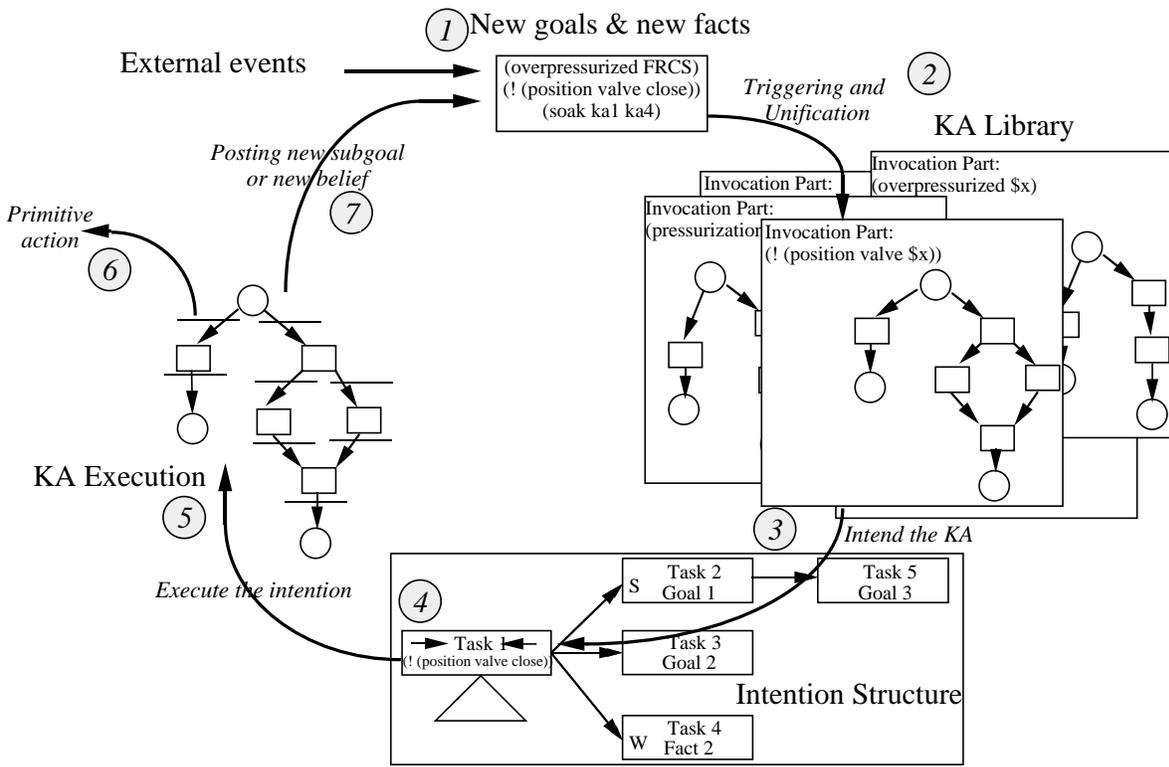


Figure 2: Procedural Reasoning System main loop

The set of KAs in a PRS application system not only consists of procedural knowledge about a specific domain, but also includes *metalevel* KAs — that is, information about the manipulation of the beliefs, goals, and intentions of PRS itself. For example, typical *metalevel* KAs encode various methods for choosing among multiple applicable KAs, determining how to achieve a conjunction or disjunction of goals, and computing the amount of additional reasoning that can be undertaken, given the real-time constraints of the problem domain. In achieving this, these *metalevel* KAs make use of information about KAs that is contained in the system database or in the property slots of the KA structures.

The PRS interpreter (Figure 2) runs the entire system. From a conceptual standpoint, it operates in a relatively simple way. At any particular time, certain goals are established and certain events occur that alter the beliefs held in the system database (1). These changes in the system’s goals and beliefs trigger (invoke) various KAs (2). One or more of these applicable KAs will then be chosen and placed on the intention structure (3). Finally, PRS selects a task (intention) from the root of the intention structure (4) and executes *one step* of that task (5). This will result either in the performance of a primitive action (6), the establishment of a new subgoal, or the conclusion of some new belief (7).

At this point the interpreter cycle begins again: the newly established goals and beliefs (if any) trigger new KAs, one or more of these are selected and placed on the intention structure, and again an intention is selected from that structure and partially executed.

It is important to note that that each intention on the intention structure (appearing as a task box within the Intention Structure shown in Figure 1) represents an entire stack of invoked KAs (procedures). In particular, as each KA is executed, it establishes certain subgoals. These subgoals, in turn, invoke other KAs, and so on. All the KAs so invoked form a run-time procedure stack, much like the run-time stack of called subroutines in conventional programming languages. Where the system has only one task to perform, there is only one such stack, and consequently one task box. But where the system needs to perform multiple tasks, it spawns multiple run time stacks, executing, suspending, and resuming these in much the same manner as processes are handled in an operating system.

PRS has several features that make it particularly powerful as a situated reasoning system, including: (1) The semantics of its plan (procedure) representation, which is important for verification and maintenance; (2) Its ability to construct and act upon partial (rather than complete) plans; (3) Its ability to pursue goal-directed tasks while at the same time being responsive to changing patterns of events in bounded time; (4) Its facilities for managing multiple tasks in real-time; (5) Its default mechanisms for handling stringent real-time demands of its environment; and (6) Its *metalevel* (or reflective) reasoning capabilities. Some of these features have been discussed in earlier reports and papers [3, 4, 5, 6, 16, 17, 18].

We now consider two large-scale applications of PRS, one concerned with malfunction handling on the space shuttle

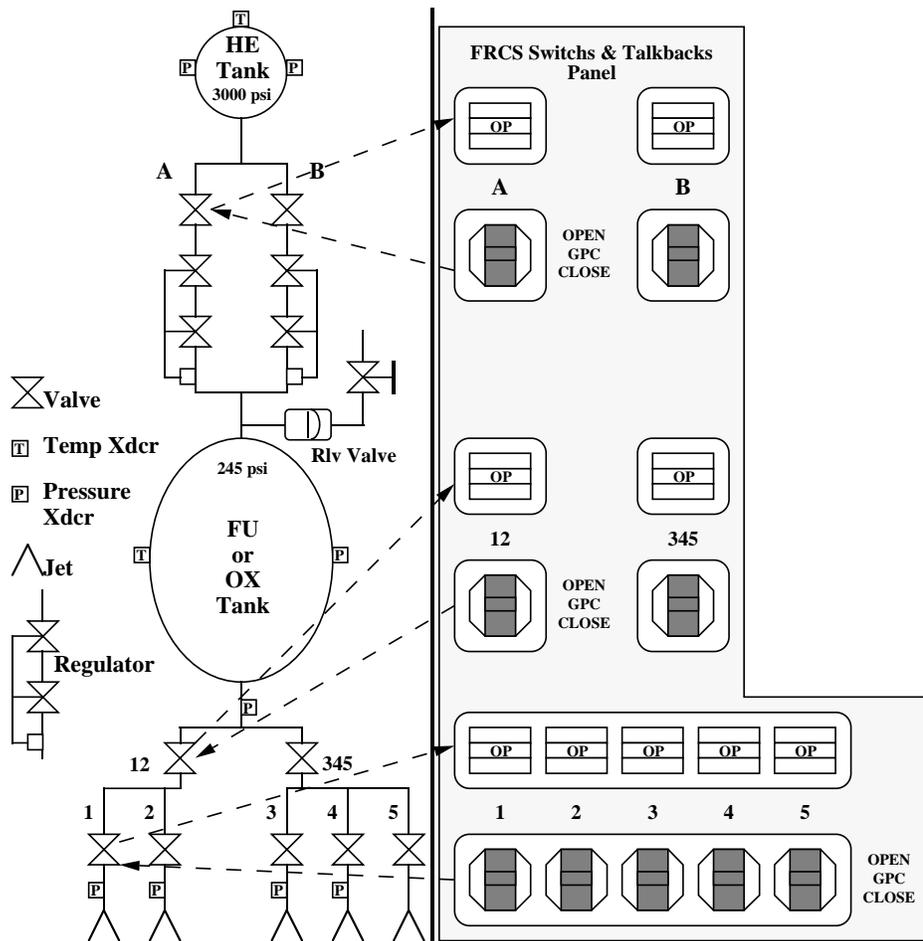


Figure 3: System Schematic for the RCS

and the other with management of a telecommunications network. Although these applications are not usually viewed as “process control,” they exhibit most of the characteristics described above.

4 The RCS Application

In this section we describe a PRS-based system that was developed for handling malfunctions in the Reaction Control System (RCS) of NASA’s space shuttle. This application typifies some of the problems involved in developing diagnostic systems for process control applications. The RCS structure is depicted in the schematic of Figure 3 (left part).

The RCS provides propulsive forces from a collection of jet thrusters to control the attitude of the space shuttle. There are three RCS modules, two aft and one forward. Each module contains a collection of primary and vernier jets, a fuel tank, an oxidizer tank, and two helium tanks, along with associated feedlines, manifolds, and other supporting equipment. Propellant flow, both fuel and oxidizer, is normally maintained by pressurizing the propellant tanks with helium.

The helium supply is fed to its associated propellant tank through two redundant lines, designated A and B. The pressure in the helium tanks is normally about 3000 psi; this is reduced to about 245 psi by regulators that are situated between each helium tank and its corresponding propellant tank. A number of pressure and temperature transducers are attached at various parts of the system to allow monitoring.

Each RCS module receives all commands (both manual and automatic) via the space shuttle flight computer software. This software resides on five general purpose computers (GPCs). Up to four of these computers contain redundant sets of the Primary Avionics Software System (PASS) and the fifth contains the software for the Backup Flight System (BFS). All of the GPCs can provide information to the crew by means of CRT displays.

The various valves in an RCS module are controlled from a panel of switches and talkbacks (Figure 3, right part). Each switch moves associated valves in *both* the fuel subsystem and the oxidizer subsystem.¹ Switches can be set to OPEN, CLOSE, or GPC, the last providing the GPCs with control over the valve position.

¹Because the two propellant subsystems are identical, only one system is represented in the left part of the figure.

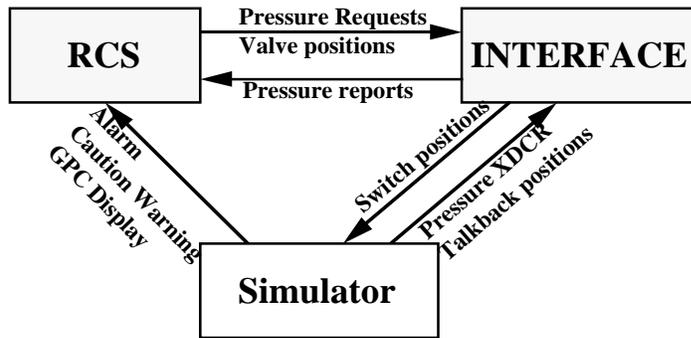


Figure 4: System Configuration

The talkbacks provide feedback on the position of their associated valves. A talkback reading may not correspond to the position of the actual valves if a valve has jammed or if the control or feedback circuitry is faulty. Under some conditions, such as when the corresponding valves in both the fuel and oxidizer subsystems do not move in unison, the talkback displays a “barberpole” warning.

As with most dynamic systems, transients are common. For example, in the process of changing switch position, there will be a short time (about 2 seconds) when the positions of the talkback and the switch will differ from one another. This is because it takes this amount of time for the actual valve to change its position. Furthermore, during this transition, the talkback will also pass through the barberpole position. Thus, a mismatched talkback and switch position or a barberpole reading does not always indicate a system fault.

4.1 System Configuration

Two instances of PRS were set up to handle the RCS application. One, called **INTERFACE**, handles most of the low level transducer readings, effector control and feedback, and checks for faulty transducers and effectors. The other, called **RCS-CONTROLLER**², contains most of the high-level malfunction procedures, much as they appear in the malfunction handling manuals for the shuttle. To test the system, a simulator for the actual RCS was constructed.

The complete system configuration is shown in Figure 4. Each of these parts is described in the following sections.

During operation, the simulator sends transducer readings and feedback from various effectors (primarily valves) to **INTERFACE** and communicates alarm messages as they appear on the shuttle system displays to **RCS-CONTROLLER**. The simulator, in turn, responds appropriately to changes in valve switch positions as requested by **INTERFACE**. The simulator can be set to model a variety of fault conditions, including misreading transducers, stuck valves, system leaks, and regulator failures.

The top-level PRS instantiation, **RCS-CONTROLLER**, contains most of the malfunction handling procedures as they appear in the operational manuals for the space shuttle. The **RCS-CONTROLLER** takes an abstract view of the domain: it deals in pressures and valve positions, and does not know about transducers, switches, or talkbacks.

The PRS instantiation **INTERFACE** handles all information concerning transducer readings, valve switches, and valve talkbacks. It handles requests from **RCS-CONTROLLER** for information on the pressures in various parts of the system and for rates of change of these values. Determination of this information can require examination of a variety of transducers, as readings depend on the status of individual transducers, their location relative to the region whose pressure is to be measured, and the connectivity of the system via open valves.

In the application described herein, over 100 object-level KAs were used together with about 25 metalevel KAs. Most of the KAs specific to RCS operations were written by space shuttle mission controllers. The database contains over 650 facts for the forward RCS alone, approximately half of which are being continuously updated during simulation.

4.2 Sample Interactions

In this section, we examine different scenarios illustrating the capabilities of PRS.

Handling Transients. The following example illustrates the capacity of the system to handle transients and reason about more than one task at a time. Consider the situation where **INTERFACE** gets a request from **RCS-CONTROLLER** to close some valve, say `frcs-ox-tk-isol-12-valve` (forward rcs, oxidizer tank, one-two isolation valve). **RCS-CONTROLLER** achieves this by sending **INTERFACE** the message `(request RCS-CONTROLLER (! (position frcs-ox-tk-isol-12-valve`

²For simplicity, **RCS-CONTROLLER** is named **RCS** in the KAs and intention structures appearing in this paper.

Switch Dilemma (Barberpole)

INVOCATION:
(*FACT (POSITION \$T BP))

CONTEXT:
(*FACT (ASSOCIATED-TALKBACK \$S \$T))

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL

DOCUMENTATION:
"This KA is called whenever there is a talkback showing a barberpole position. If after 10 seconds, the tb is still in bp, a warning is send to the astronauts."

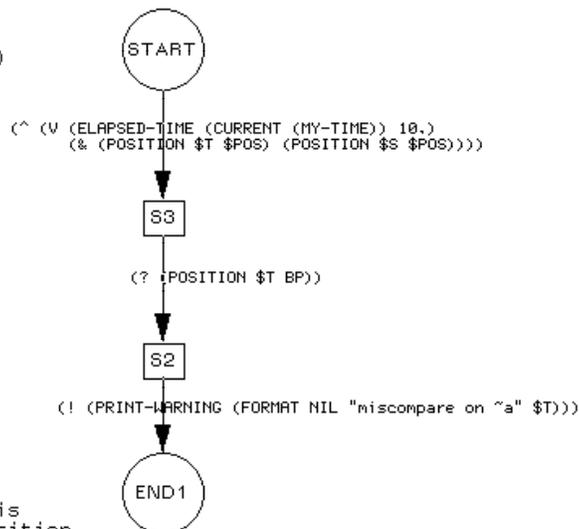


Figure 5: KA for checking the barberpole position on a talkback

c1))). Responding to this request, **INTERFACE** calls a KA that, in turn, asks the astronaut to place the switch corresponding to this valve in the closed position. Once the astronaut has done this, **INTERFACE** will wait until the talkback shows the requested position and will then advise **RCS-CONTROLLER** that the valve has indeed been closed.

However, while this is taking place, **INTERFACE** will also notice that, just after the switch is moved to the closed position, there is a mismatch with the talkback indicator (which will still be showing open, because of the normal delay in valve movement). Furthermore, a fraction of a second later, the talkback will move into the barberpole position, another indication that things could be wrong with the valves.

Each of these events will trigger a KA and thus initiate execution of a task (intention) that seeks to confirm that the talkback moves to its correct position within a reasonable time (ten seconds). At this point, the system is dealing with three different tasks, one responsible for answering the request, one checking the miscomparison between the switch and the talkback, and one checking the barberpole reading. Each of these last two tasks immediately suspend themselves (using the “wait-until” (^) operator, see Figure 5) while awaiting for the talkback to move to the correct position or until ten seconds has elapsed (in which case an error is reported to the astronauts).

Notice that the KAs that respond to the request from **RCS-CONTROLLER** to change the valve position, that monitor for possible switch dilemmas, and that check the barberpole reading are all established as different intentions at some stage during this process. Various metalevel KAs must therefore be called, not only to establish these intentions, but to decide which of the active ones to work on next.

A typical state of the intention structure is shown in Figure 6. It shows a number of intentions in the system **INTERFACE**, ordered for execution as indicated by the arrows. The intention labeled **Meta Selector** is a metalevel KA. The other intentions include two that are checking potential switch problems (**Switch Dilemma (Barberpole)** and **Switch Dilemma (Closed)**) and one that is responding to the request to close the valve (**Open or Close Valve**). The metalevel intention, in this case, is the one currently executing. Although not clear from the figure, it has just created and ordered the new intentions resulting from the miscomparison and barberpole problems.

In this example, we have shown how PRS handles multiple tasks. All these tasks are started upon reception of asynchronous events: the changes of the talkback reading. Moreover, two of these tasks deal with transients that are potentially dangerous.

Handling Inaccurate Data. In this scenario, we show how two PRS agents cooperate and control the execution of their intentions so as to handle faulty transducers and the resulting false warning alarms.

We will assume that one of the transducer on the oxidizer tank (**frcs-ox-tk-out-p-xdcr**) fails and remains jammed at a reading of 170 psi. This causes a number of things to happen. First, it causes a low-pressure alarm to

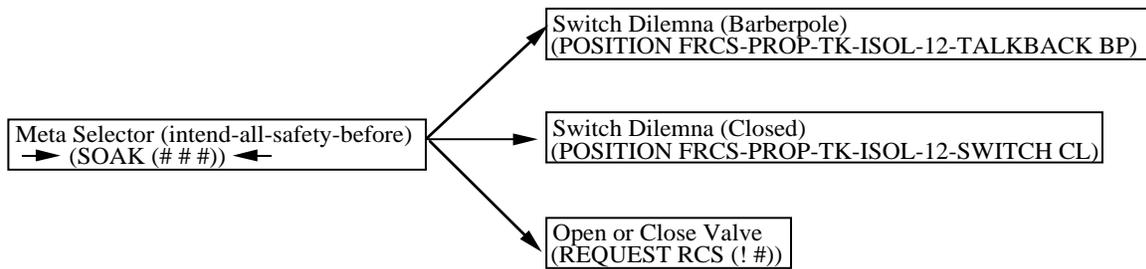


Figure 6: Intention Structure during Switch Operation

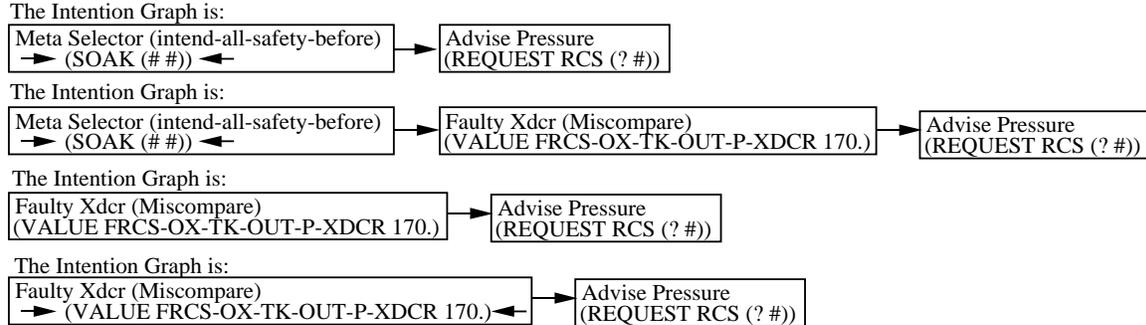


Figure 7: Intention Structure Development

be activated. This will be noticed by the PRS instantiation `RCS-CONTROLLER`, which will immediately respond to the alarm by initiating execution of the KA `Pressurization Alarm (Propellant Tank)`. This KA will, in turn, request a pressure reading from `INTERFACE` to ensure that the alarm is valid.

While this is happening, `INTERFACE` by itself has noticed that the two transducers on the oxidizer tank disagree with one another (in this case, the other transducer is reading the nominal value of 245 psi). This invokes a KA that attempts to determine which of the two transducers is faulty. It does this by first waiting a few seconds to ensure that the mismatch is not simply a transient, and then testing to see if one of the readings is outside normal limits. If so, it assumes this is the faulty transducer, (this is indeed the procedure used by astronauts and mission controllers).

Notice what could happen here if one is not careful. Having more than one thing to do, `INTERFACE` could decide to service the request for a pressure reading for the suspect tank. If it does so, it will simply average the values of the two transducer readings (yielding 207 psi) and advise `RCS-CONTROLLER` accordingly. Clearly, this is not what we want to happen: any suspect parameter readings should be attended to before servicing requests that depend on them, and this implemented in metalevel KAs. When `INTERFACE` eventually gets around to servicing the request from `RCS-CONTROLLER`, it disregards the faulty transducer reading and thus advises `RCS-CONTROLLER` that the pressure is 245 psi. `RCS-CONTROLLER` then determines that the alarm was activated in error and that the pressure is within normal operating range.

Of course, the specific technique used here for recognizing the faulty transducer is relatively simple—in general, one would want to draw on other, more sophisticated techniques for determining sensor or effector failures. However, these other techniques are not difficult to encode in PRS; the essential component is not the recognition procedure itself, but rather the role played by the metalevel in appropriately controlling the order of task execution.

Even with all this going on, other things are happening within the `INTERFACE` system. For example, the fact that the transducer is determined to be bad, together with the fact that it is the very transducer that informs the shuttle computers of overpressurization problems, causes the invocation of another KA. This KA reflects a flight rule that states that overpressurization protection is lost while the transducer is inoperative.

As before, metalevel KAs are invoked to determine which KAs to adopt as intentions and how to order them on the intention structure. The development of the intention structure during the early stage of this process is shown in Figure 7.

This example shows how important it is for a situated reasoning system to be able to change its focus of attention. Moreover, it demonstrates the need for sophisticated reasoning capabilities to ensure that tasks are executed in an appropriate order.

Diagnosis and Malfunction Handling. This last example illustrates how PRS manages both reactive and goal-directed reasoning. We first examine the case in which the regulator on the feed line between the helium tank and its associated propellant tank fails in position “open”. In this example, we will assume that the `frcs-fu-he-tk-A-reg`

has failed. We will focus primarily on **RCS-CONTROLLER** (**INTERFACE** is, of course, working away during this process as discussed above).

The first thing that happens when the regulator fails is that pressures throughout the fuel subsystem begin to rise. When they exceed the upper limit of 300 psi, certain caution-warning (cw) alarms are activated. These events trigger the execution of a KA that attempts to confirm that the system is indeed overpressurized.

Note that this process is more complicated than it first appears. The high transducer readings that gave rise to the caution-warning alarm will also trigger KAs in the PRS system **INTERFACE**. These KAs will proceed to verify that the corresponding transducers are not faulty (as described in Section 4.2); that is, that the reading of the transducers is indeed accurate. While doing this, or after doing this, **INTERFACE** will get a request from **RCS-CONTROLLER** to advise the latest pressure readings. If **INTERFACE** is in the process of checking the transducers, it will defer answering this request until it has completed its evaluation of transducer status. But eventually it will return to answering the request and, in the case we are considering, advise that the pressure is indeed above 300 psi.

On concluding that the system is overpressurized, another KA (**Overpressurized Propellant Tank**) is activated and this, eventually, concludes that the A regulator has failed. Note that this KA establishes parallel subgoals to close both the A valve and the B valve at the same time, as there are cases when both are open. For the A valve, this involves a request to **INTERFACE** as discussed above. However, for the B valve, the system notices that the B valve is already closed. Thus, its goal is directly achieved without the necessity to perform any action or request.

The final goal of this KA activates another KA that opens the valve of the alternate regulator (B). Having opened the valve, it is desirable to then place it under the control of the on-board computers. However, this cannot be done until the pressure in the system drops below 300 psi, as otherwise the GPC will automatically shut the valve again. Thus, the malfunction handling procedures specify that the astronaut should wait until this condition is achieved before proceeding to place the valve switch in the GPC position. **RCS-CONTROLLER** achieves this by asking **INTERFACE** to monitor the pressure and advise it when it drops below 300 psi. While waiting for an answer, the task is suspended, and **RCS-CONTROLLER** gets on with whatever else it considers important.

When the pressure eventually drops below the threshold (because the astronauts are firing the jets), the task (intention) is awakened, and execution continued. Thus, the valve switch is finally placed in the GPC position and the overpressurization problem resolved.

This example illustrates both the reactive and goal-directed behavior of PRS; that is, the system is reactive in its initial response to the overpressurization alarm, and goal-directed in its activities towards repressurizing the system.

5 The IRTNMS Application

Most countries in the world are experiencing a widespread growth in their telecommunications networks. Although the networks are designed to carry normal traffic, with alternative routes for average peak-hour demands, problems like congestion, overload, or failure in parts of the network occur frequently. These problems are due to events such as natural disasters (e.g., bush-fires, earthquakes, floods), phone-in polls, a high number of callers on certain days (e.g., Christmas Day and New Year), or failure of switching systems and transmission links.

All these events can lead to a high level of congestion as the effects of excessive unsuccessful call attempts quickly spread throughout the entire network. This results in poor service to the customer and loss of revenue. To solve these problems, Network Traffic Management (NTM) is being introduced in many centers around the world. NTM is the function of monitoring the status and performance of the network in real time and, when necessary, taking action to control the flow of traffic to ensure the maximum utilization of the network capacity [20].

This section describes a PRS-based system for diagnosing, controlling, and monitoring a telecommunications network. The system, called the Interactive Real-Time Telecommunications Network Management System (IRTNMS), was developed for Telecom Australia.

5.1 System Configuration

Three instances of PRS were set up in IRTNMS to handle the network management task: **DIAGNOSIS**, **CONTROLS**, and **MONITOR**. The **DIAGNOSIS** module receives the route data from the network and diagnoses the different problems occurring. These problems are passed to the **CONTROLS** module, which then suggests the different control actions that could be taken to alleviate the problems; the controls are then sent to the exchanges. The **MONITOR** module, working in parallel with the above two modules, continuously monitors the controls that have already been taken, modifying and deactivating them as necessary. In the current prototype, a **SIMULATOR** replaces the actual network. The interactions among the various modules involved in the overall system are shown in Figure 8.

The system architecture with the above three PRS modules mirrors the natural division of network management tasks; namely, diagnosing problems, taking initial control actions, and monitoring them continuously. The multi-agent architecture of PRS allows these modules to be run concurrently as three different processes. This also means that the tasks of diagnosis, control selection, and monitoring can be done asynchronously, i.e., the **DIAGNOSIS** can be working on

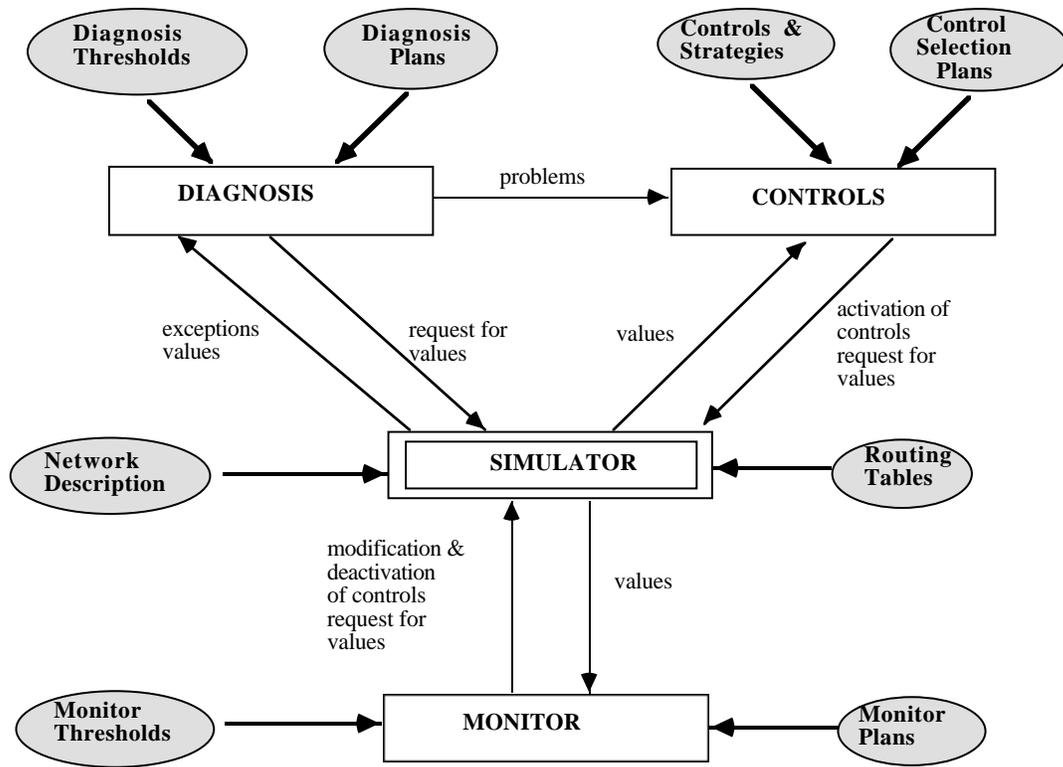


Figure 8: IRTNMS Architecture

current data, while **CONTROLS** can be solving problems based on the previous data, and **MONITOR** can be monitoring the controls imposed for solving previous problems.

The network simulator simulates the normal flow of traffic in the network as well as some of the more commonly occurring problems. The simulator sends exception data to **DIAGNOSIS** in fixed cycles of three or six minutes and alarms as and when they occur (i.e., asynchronously). The simulator also receives commands from **CONTROLS** and **MONITOR** to activate, deactivate or modify controls.

DIAGNOSIS is initiated by route exceptions, exchange exceptions and route alarms received from the simulator. Its main task is to diagnose the different types of problems in the network. It also diagnoses the cause of these problems. For example, it not only isolates a congestion problem in a particular link, but also identifies its cause as being due either to a single link (i.e., a congestion problem due to a first-offered route) or to multiple links (i.e., a general congestion problem).

CONTROLS contains the strategies for various network problems and the controls applicable under each strategy. It is activated by the problems it receives from **DIAGNOSIS** and is responsible for selecting controls or corrective measures for the diagnosed problems. It has to perform a variety of checks on the network status and choose a number of different controls to alleviate the problems. The controls chosen fall into two broad categories: expansive controls, which re-route traffic via alternative routes, and protective controls, which block a certain percentage of the network traffic.

MONITOR is activated as soon as a control is issued to the network, and remains active until there are no longer any controls to monitor. It is responsible for continuously checking the status of various network parameters, modifying the controls (by increasing or decreasing the percentage of calls re-routed) and finally removing them when they are no longer required. It handles transients in the network by waiting for the network to stabilize before suggesting the removal of controls.

All three modules communicate with each other by sending messages. Given the natural division of network management tasks among the three modules, communication between modules is kept to a minimum. Also, the amount of communication depends only on the actual number of problems in the network and the number of controls activated—it does not depend on the size of the network.

5.2 Sample Interactions

In this section, we examine a sample run of the system and illustrate how the various instances of the system interact in diagnosing, controlling and monitoring problems.

In the following sample run, the simulator sends exception data every 180 seconds (which reflects the current

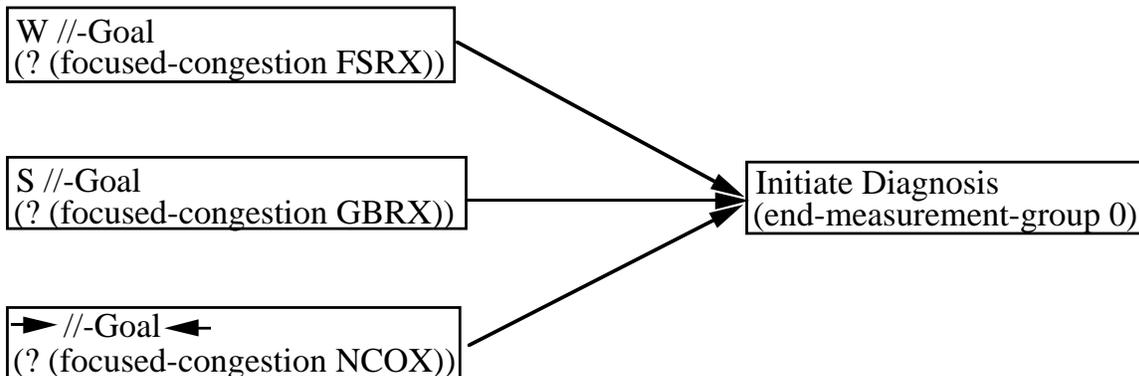


Figure 9: Parallel Diagnosis of Problems

operating conditions of Telecom’s major networks) and sends alarms asynchronously. However, the simulator could send data in a shorter or longer cycle as required.

Diagnosing problems in the network. In this example, the simulator, at the end of first 180 seconds, sends a set of exception data to **DIAGNOSIS**. The format of this data is as follows:

```
(route-exception 180 RPOFL Brunswick Northcote 12)
(route-exception 180 RPOFL Brunswick Batman 47)
```

The first item in the above list states that there is a route-exception at time 180 from the node Brunswick (BRUX) to Northcote (NCOX) whose route percentage overflow is 12%.

DIAGNOSIS receives only route, exchange and code exceptions; all other values it needs are received on a demand basis from the simulator. As the system may have to wait for the simulator to send these data, all problems of a particular type are handled in parallel. This ensures that, while one problem is waiting for the data to arrive from the simulator, the other problems are being attended to. This point is illustrated in Figure 9, where the diagnosis of three different code congestion problems are handled in parallel; one of them is currently active (indicated by arrows); one of them is suspended waiting for data from the simulator (indicated by S), and the other has just been awakened (indicated by W). Although we have shown only three tasks executing in parallel, in reality a particular regional network would have around 40 to 50 exchanges; in this case, executing the diagnostic tasks in parallel would provide a significant increase in speed.

For the above situation, **DIAGNOSIS** makes use of its procedures to diagnose the following problem in the network:

- (1) **First-offered congestion from Brunswick to Northcote due to Batman at time 180 (ACTIVE).**

The above display states that there is a final choice congestion from Brunswick to Northcote at time 180 caused by the first-offered route Brunswick to Batman.

Having diagnosed the problems, **DIAGNOSIS** sends messages to **CONTROLS** indicating the type of the problem, its cause, and other details. **DIAGNOSIS** is then free to work on the next set of data while **CONTROLS** can, in parallel, take the corrective measures for the problems identified so far.

This example illustrates how the system can execute multiple tasks in parallel and thereby operate more efficiently. It also shows how one can exploit the multi-agent architecture of PRS by partitioning modularly the network management task.

Taking corrective measures. Following the above actions, **CONTROLS** solves the observed problem by first examining expansive controls. It makes use of different procedures to re-route the traffic on the first-offered route that is causing the problem. If this does not solve the problem, it tries to re-route traffic from the final-choice route and then all the other first-offered routes that are overflowing onto the final-choice route.

If there is no opportunity to take expansive controls, **CONTROLS** tries protective controls. Protective controls re-route a percentage of calls onto the Recorded Voice Announcement (RVA) or congestion tone.

In this case, the system suggests the following solution; namely, to re-route the first-offered traffic from Brunswick to Batman via Footscray:

```
First-offered congestion Brunswick-Northcote (180):
(1) Activate BRUX102: Reroute Batman overflow traffic via Footscray
    33 calls at 60%
```

The number of calls (or percentage of calls) to be re-routed (shown on the last line above) is based on the number of calls that are overflowing and the spare capacity available on the new route.

The above example illustrates the complex nature of planning corrective measures and the goal-directed behavior of **CONTROLS**. It would be extremely difficult to express the procedures for planning corrective measures in the more conventional rule-based form.

Delayed Feedback. In the next 180 second time period, the following problems are notified:

- (1) Code focused congestion at 311 code of Footscray at 360 (ACTIVE).
- (2) First-offered congestion from Brunswick to Northcote due to Batman at time 360 (SUSPEND)

Note that although the operator has solved the first-offered congestion from Brunswick to Northcote due to Batman by taking appropriate controls, the problem has reappeared at time 360. This is because of delayed feedback, i.e., there is a time lag between activating a control and the control taking effect on the real network. The system recognizes this fact and suspends the problem. If the problem does not disappear in the next cycle, the system warns the operator that the control activated two cycles ago is not taking effect.

Conventional expert systems have difficulty recognising and reacting to such delayed feedback. PRS achieves this capability by suspending intentions and then waiting for a specified period of time (or for some other critical event) before reactivating them.

Multiple controls. So far we have only shown the system solving a single problem and producing a single control for that particular problem. However, the system is capable of solving multiple problems and producing multiple controls for each problem. Given that the network is continuously changing, **CONTROLS** has to be very careful in allocating spare capacity to the multiple controls. We illustrate below a situation where multiple controls are required to solve a single problem.

If the operator chooses to solve the code congestion problem (as given above), the system suggests the following code controls. These controls selectively block the 311 code traffic from various originating nodes.

Code focused congestion Footscray-311 (360):

- (1) Activate BATX501: Reroute 311 overflow traffic at Batman A to RVA
140 calls at 50%
- (2) Activate BRUX502: Reroute 311 overflow traffic at Brunswick to RVA
102 calls at 50%
- (3) Activate GBRX503: Reroute 311 overflow traffic at Greensborough to RVA
80 calls at 50%
- (4) Activate NCOX504: Reroute 311 overflow traffic at Northcote to RVA
220 calls at 50%

The percentages above are based on the heuristic of re-directing half the overflow traffic to RVA, followed by continuous monitoring to increase or decrease this value. A deteriorating situation will cause **MONITOR** to re-direct half the total calls offered (rather than the overflow) to RVA.

The above example illustrates how the system reasons about the problem, traces its way back to the original nodes causing the problem, and then places protective controls on each one of these originating nodes.

Monitoring of controls. As the system continues running, we find that, at time 540 seconds, there are no more active problems in the network. Thus **CONTROLS** will be idle. However, as described earlier, **MONITOR** will be monitoring all the controls activated for the final-choice congestion and code-congestion problems.

At time 540, **MONITOR** determines that there is enough idle capacity on the final-choice route Brunswick to Northcote to carry all overflowing calls. Hence, the expansive control activated at time 180 can be deactivated. However, **MONITOR** waits for a fixed period of time for the network to stabilize before suggesting that the control be deactivated. If the network stabilizes within the wait period, the following message is displayed:

First-offered congestion Brunswick-Northcote (180):

- (1) Deactivate BRUX102: Reroute Brunswick overflow traffic via Footscray.

MONITOR can also suggest a reduction or increase in the percentage of calls that are to be re-routed. **MONITOR** and **CONTROLS** are in constant communication, informing each other of the presence of controls and their activation levels. This communication is important for the system to make best use of the available network capacity by using as many controls as necessary and reducing or removing controls as soon as possible.

The above scenario illustrates the continuous monitoring of all the activated controls. It also shows how the system models transients in the network by waiting until the network stabilizes.

6 Comparison with Other Architectures

PRS is based on the notion of a rational agent, or what is conventionally called a Belief-Desire-Intention (BDI) architecture. In this section, we briefly review some other approaches to the design of situated reasoning systems.

Architectures for situated reasoning systems can be considered to be of the following paradigms:

- *Rule-Based Architectures*: This paradigm is an extension of the more traditional expert-system to meet the real-time constraints of process control applications. The domain knowledge is encoded declaratively as rules that can be triggered (invoked) by external events.
- *Blackboard Architectures*: The blackboard model is very similar to a data-driven expert system, except that multiple processes are active and share a common database (blackboard).
- *Situated Automata*: This paradigm closely resembles conventional real-time software. In this approach, the responses of the system to all anticipated situations is hard-wired into the system.
- *BDI Architectures*: This architecture represents a system or agent in terms of its beliefs, desires (or goals), and intentions. Knowledge is encoded as plans or procedures. Based on its current beliefs and goals, the system adopts one or more plans as its intentions, which it is then committed to carrying out.

Each of the above paradigms has its own advantages and disadvantages. The exact choice depends on a number of factors, including desired reaction time, ease of modification, modularity, and the nature or complexity of the domain knowledge.

As mentioned above, PRS is based on the BDI architecture. Other BDI systems, such as RAPS [2], also invoke tasks on the basis of system beliefs about the current world situation and expand these tasks hierarchically. However, these systems do not include many of the real-time capabilities of PRS and have not been widely applied.

Real-time expert systems like G2 and PICON [13] are extensions of expert systems' technology for real-time use. Although both G2 and PICON have many of the real-time features of PRS, the domain knowledge is encoded as rules. While G2 has some schemes for expressing procedural knowledge, the representation (a la Pascal) is less expressive (no goal and plan representation) than that used in PRS. More importantly, however, is that G2 has very weak mechanisms for handling procedure execution: procedures cannot be interrupted or suspended,³ there is no simple means of capturing task or procedure dependencies, and there are no generic reflective capabilities for manipulating priorities of multiple tasks, maintaining and changing focus of attention, and conditionally suspending tasks.

A situated reasoning system can also be built using systems such as BB1 [8] and RT-1 [1], which are based on the blackboard paradigm. BB1 has been applied to medical monitoring and RT-1 to a pilot's associate for fighter aircraft. These systems use a global database for storing information and repeatedly execute a cycle consisting of task invocation, selection (or scheduling), and execution. Tasks are invoked by changes to the blackboard or the occurrence of specific interprocess communications.

These systems have many features in common with PRS. However, there are important differences. BB1 and RT-1 do not have a convenient representation for procedures nor very powerful mechanisms for reasoning about task scheduling. Moreover, in current blackboard systems, the actions carried on by the system are not interruptible. As these procedures can be quite complex, this can pose a serious problem for providing real-time behavior. Keeping the blackboard consistent when knowledge sources are asynchronous is a serious problem that has yet to be addressed; most blackboard architectures use an agenda of pending knowledge sources (tasks) that are run serially. The agenda manager runs with considerable overhead, since it is invoked in each cycle, and for each task present on the agenda.

GAPPS (Goals as Parallel Program Specifications) [12] and Rex [11, 19], which are based on the situated automata paradigm, can also be used as a situated reasoning system. GAPPS takes as input the top-level goals of the system and a set of goal-reduction rules. It transforms these into the description of a circuit that exhibits situation-specific response. Although a situated reasoning system implemented using this paradigm can be expected to have a better measure of reactivity than the other paradigms, it is unclear whether it would have the flexibility needed for complex process-control applications. In particular, it does not provide a language for directly expressing procedural information and has no means for expressing information about task management and control.

In addition, there are two further important differences between all the above systems and PRS. First, none provide sufficiently powerful mechanisms for balancing the decision-making requirements against the constraints on time and information that are typical of complex domains. Second, with the exception of GAPPS, none attempt to provide a semantics for their knowledge representation, which is critical for proper system verification and maintenance.

7 Conclusion

In this paper, we have attempted to indicate that architectures based on the notion of a rational agent provide one possible solution to the problem of controlling and managing complex processes. In particular, they provide a means

³Except with the explicit use in the procedure of instructions enabling the scheduling of other procedures or rules

for balancing long-term goal-directed reasoning with short-term situation-determined reaction to unusual or important events.

The two examples presented in this paper exhibit all the characteristics of applications identified in Section 2. We have shown that PRS addresses most of them well.

- **Asynchronous event handling.** In both applications, various sensor data, alarms, and messages are sent asynchronously to the different PRS agents. The interruptible procedure execution of PRS allows for asynchronous communication.
- **Procedural representation of knowledge.** By design, the basic knowledge representation used by PRS is procedural. In both applications, this representation has proven to be more effective than a rule-based representation.
- **Handling of multiple problems.** The parallel execution of tasks provided by PRS offers an elegant way of solving multiple problems. The PRS metalevel reasoning feature provides a flexible way of prioritizing these problems and then solving them. A more detailed description of the metalevel reasoning capabilities of PRS can be found in [9].
- **Reactive and goal-directed behavior.** We have shown that PRS can be reactive when responding to events such as alarms and network congestion, while it can be goal directed when it is working on specific tasks, such as the re-routing of network traffic.
- **Focus of attention.** By using the appropriate metalevel KAs, PRS can react to new information and change its current task to focus on new problems.
- **Reflective reasoning capabilities.** The features mentioned above, such as changing the focus of attention and guaranteed response time, can only be realized when one has a sophisticated metalevel reasoning mechanism. PRS offers such capabilities using the same syntax as the application procedures, making it simple for the user to encode metalevel strategies.
- **Continuous embedded operation.** Both applications have been exercised with sophisticated simulators under continuous operations. PRS is currently being reimplemented in C to provide more robust operational capabilities.
- **Handling inaccurate data.** The example of the failed sensor in the RCS application demonstrates how the system can handle inaccurate data.
- **Handling transients and delayed feedback.** Both of these problems are handled in PRS by conditionally suspending and reinvoking tasks.
- **Operator control.** The IRTNMS system is capable of operating in different modes – manual, semi-automatic, and automatic. The reactive behavior and flexible prioritization of tasks using the metalevel reasoning feature enables PRS to hand over the ultimate control of the system to the human operator under all the operating modes.

For situated reasoning systems, it is also important that there be a guaranteed bound on reaction time and on certain response times. Although this is not exhibited in the above examples, we show elsewhere [5] that an appropriate set of metalevel KAs can guarantee a bounded reaction time. A bound on response time is application specific and cannot be guaranteed a priori. One needs to make an in-depth analysis of the KAs used in the system to ensure the completion of critical KAs within desired hard time limits.

PRS does not include as standard features some of the mechanisms found in conventional monitoring and control systems. For example, PRS does not provide as standard any mechanism that takes account of goal deadlines. The main reason for this absence is that PRS has been built as a generic tool to be used in a wide range of applications, and no specially coded routines have been built for particular application domains. However, such mechanisms can be encoded by the user as metalevel KAs as required.

There still remains considerable work to be done in developing this approach further. For example, the current system relies on the user constructing appropriate metalevel KAs for recognizing potentially harmful interactions among procedures executing in parallel. Clearly, it would be preferable if the system could determine such interactions automatically. In addition, we need to develop a much better understanding of deliberation processes as represented by metalevel KAs, and under which conditions it is appropriate to invoke them. These and other related issues are the subject of current research.

References

- [1] R. Dodhiawala, N. S. Sridharan, P. Raulefs, and C. Pickering. Real-time AI systems: A definition and an architecture. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 256–261, Detroit, Michigan, U.S.A, August 1989.
- [2] R. J. Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 202–206, Seattle, Washington, U.S.A, 1987.
- [3] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 972–978, Detroit, Michigan, U.S.A, 1989.
- [4] M. P. Georgeff and F. F. Ingrand. Real-time reasoning: The monitoring and control of spacecraft systems. In *Proceedings of the Sixth IEEE Conference on Artificial Intelligence Applications*, Santa Barbara, California, U.S.A, March 1990.
- [5] M. P. Georgeff and F. F. Ingrand. Research on procedural reasoning systems. Final Report, Phase 2, for NASA Ames Research Center, Moffet Field, California, U.S.A, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, June 1990.
- [6] M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74:1383–1398, 1986.
- [7] M. P. Georgeff, A. L. Lansky, 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, U.S.A, 1987.
- [8] B. Hayes-Roth, R. Washington, R. Hewett, M. Hewett, and A. Seiver. Intelligent monitoring and control. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 243–249, Detroit, Michigan, U.S.A, August 1989.
- [9] F. F. Ingrand and M. P. Georgeff. Managing deliberation and reasoning in real-time ai systems. In *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning*, Santa Diego, California, U.S.A, November 1990.
- [10] F. F. Ingrand, J. Goldberg, and J. D. Lee. SRI/Grumman Crew Members’ Associate Program: Development of an Authority Manager. Final Report, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1989.
- [11] L. P. Kaelbling. An architecture for intelligent reactive systems. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 395–410. Morgan Kaufmann, Los Altos, California, U.S.A, 1987.
- [12] L. P. Kaelbling. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 60–65, Saint Paul, Minnesota, U.S.A, 1988.
- [13] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read. Real-time knowledge-based systems. *AI Magazine*, 9(1):27–45, 1988.
- [14] M. Ljungberg. The OASIS Air Traffic Management System. Technical note 28, Australian Artificial Intelligence Institute, Carlton, Australia, 1992.
- [15] A. S. Rao and M. P. Georgeff. Intelligent real-time network management. In *Avignon: Expert Systems and their Applications*, Avignon, France, 1990.
- [16] A. S. Rao and M. P. Georgeff. Asymmetry thesis and side-effect problems in linear time and branching time intention logics. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, 1991.
- [17] A. S. Rao and M. P. Georgeff. Deliberation and its role in the formation of intentions. In *Proceedings of the Seventh Conference on Uncertainty in Artificial Intelligence (UAI-91)*, 1991.
- [18] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, San Mateo, 1991.

- [19] S. J. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In J. Y. Halpern, editor, *Proceedings of the First Conference on Theoretical Aspects of Reasoning about Knowledge*, San Mateo, California, U.S.A, 1986. Morgan Kaufmann Publishers, Inc.
- [20] J. L. Seamons. Network Traffic Management in Telecom Australia. *Telecommunication Journal of Australia*, 38:15–24, 1988.
- [21] L. Wesley, F. F. Ingrand, J. D. Lee., J. Rushby, and J. Garcia Luna. Application of PRS to network management system. Final Report, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1991.