N91-22780

Representing Functions/Procedures and Processes/Structures for Analysis of Effects of Failures on Functions and Operations

Jane T. Malin
Automation and Robotics Division - ER2
NASA Lyndon B. Johnson Space Center
Houston, Texas 77058

Daniel B. Leifker
The MITRE Corporation
1120 NASA Road One
Houston, Texas 77058

Abstract

Current qualitative device and process models represent only the structure and behavior of physical systems. However, systems in the real world include goal-oriented activities that generally cannot be easily represented using current modeling techniques. We propose an extension of a qualitative modeling system, known as *functional modeling*, which captures goal-oriented activities explicitly, and we show how they may be used to support intelligent automation and fault management.

Background

Artificial intelligence (AI) technology for intelligent automation of monitoring, control, and fault management of space systems will result in significant reductions in operational costs of manned and unmanned systems, as well as increased capability to carry out new types of unmanned missions. In addition, more robust fault management performance will reduce costs for space system maintenance and repair and can potentially reduce risks from undetected failures.

An important goal of work in AI is to produce software that can respond constructively to a wide class of problem scenarios. At the same time, however, the software should operate in ways that reflect human thought and reasoning patterns. Representations have therefore been developed that either correspond to or mesh with conventional human perspectives of the problem domain. Understandable rule-based and object-based systems have been successfully developed and used by flight controllers in the Space Shuttle program (Muratore, 1990).

Additional types of representations are needed to capture key concepts and strategies that are used for fault management by mission controllers, system designers, and safety personnel. They use mental models of the function and structure of designed systems and their interrelated components. These models require more expressive power and an enlarged scope before they can be added to the set of AI representations that are now successfully used.

Modeling Structure, Behavior, and Function

Computer simulations and models are used to represent, to any desired level of detail, the structures and actions of physical systems. Modeling is done primarily to understand certain dynamic principles related to the given system that cannot be analyzed in closed form and cannot be studied in the system itself without great cost, danger, or inconvenience.

A number of researchers have developed a rich simulation theory known as qualitative modeling (see, for example, Forbus, 1984; Davis and Hamscher, 1988). The concept we present here, functional modeling, is an outgrowth of recent attempts to merge qualitative modeling and discrete event simulation into a single unified paradigm. Malin, Basham, and Harris have implemented this paradigm in a system known as CONFIG (Malin et al., 1990), a modeling and simulation tool prototype for analyzing normal and faulty qualitative behaviors of engineered systems. Like other device modeling systems, CONFIG has relied on component structures and processes to represent real-world systems.

However, current structure and process approaches rarely provide a way to represent system functionality. In fact, there have been admonitions to not mix function into a model of system structure and behavior. Nevertheless, since devices are systems designed to be used in goal-oriented activity, functionality is important to model and analyze. Modelers need representations of functionality to evaluate the success of a design, that is, to analyze how well the device will perform its functions. We argue that the "functionality" of the device is a concept that captures how activities (the operations/acts of the device/structure in time) produce a set of effects that are goals the device is designed to achieve. The structure of the device is a set of components and their interrelationships. The behavior of the device consists of what it does, and can include internal processes that cause changes in itself or in things it operates on. Thus, to model functionality, one must model structure, behaviors, goals and time -- the principal constituents of a goal-oriented activity. Since procedures are used by human and machine controllers to sequence and structure goal-oriented activities, representing function also provides the basis for representing procedures and controllers. Goals reside in controllers and in designers of devices and procedures, and are not ordinarily part of the physical device itself. Nevertheless, they must be combined with representations of device structure and behavior to model and analyze the functionality of the design.

The Space Shuttle Remote Manipulator System (RMS) can be used to illustrate this. Its structure, as shown in Figure 1, is made up of components such as the manipulator arm, manipulator retention latches (MRL), manipulator positioning mechanisms (MPM), and payload capturing subsystems. Its behavior consists of various movements and capture and release operations. The system has been designed to perform functions such as deploying a payload while avoiding collisions and remaining operational. These serve goals of safety and transport.

Clearly, fault management systems can be made more useful if they embody models that represent the entire scope and range of the monitored physical systems. Such a system for the RMS, for example, would be totally integrated with RMS procedures (which, being goal-oriented, are not a part of RMS designed structure and behavior) and thus support RMS fault management activities at new levels of operation. To make these ideas more precise, we turn now to a detailed discussion of real-world systems and how they may be modeled in terms of function as well as behavior.

Functional Systems

Figure 2 depicts the organization of a real-world designed system and its relationship to real-world goals. It is tempting at first to view the *entire* designed system as a physical device with a given structure and behavior. Using the RMS example, however, it is easy to see that this is not the case: the entire RMS designed system encompasses not only the physical device but also a collection of related procedures that govern the (presumably judicious) use of the RMS device to achieve goals. A procedure is an information structure whose purpose is to help a controller successfully carry out and monitor plans. A procedure also contains information to support impact assessment or even fault management replanning and recovery.

The bridge between the procedure and the device is the controller, which (a) consults the procedure and decides on an action, (b) determines the true state of the device by requesting specific monitoring information,

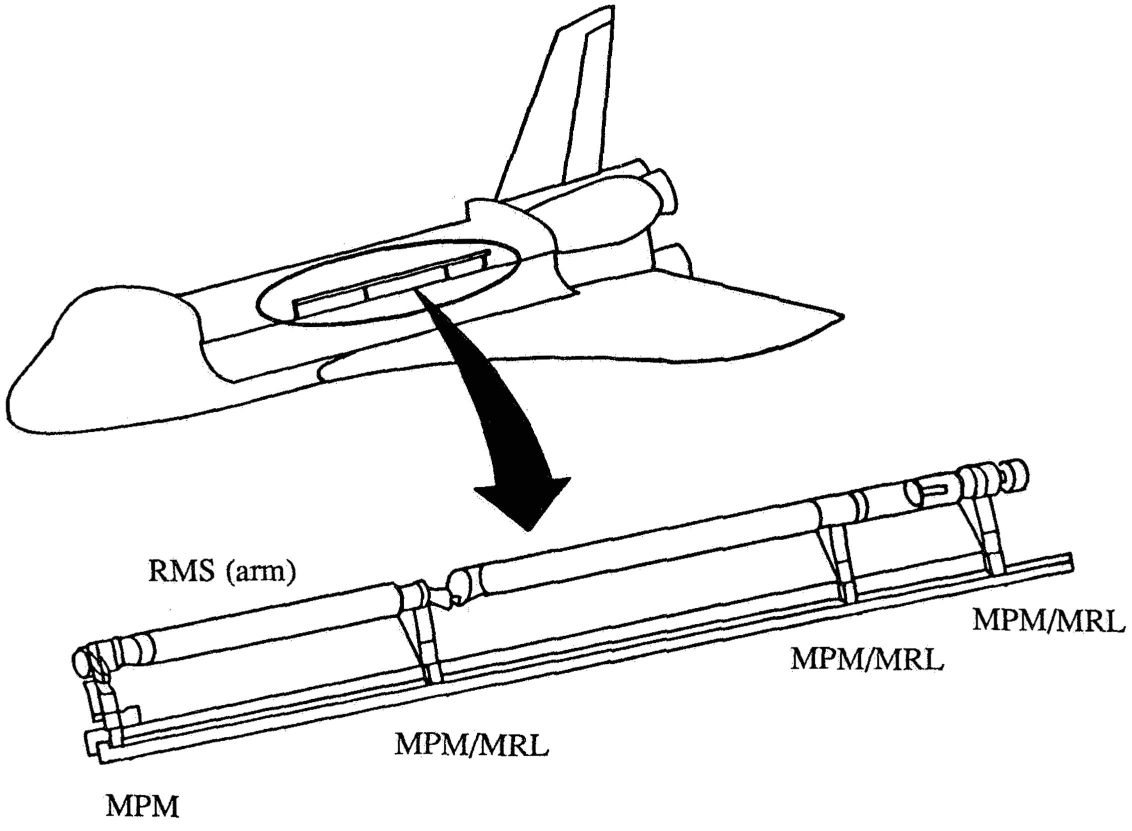

Figure 1: Manipulator Arm

(c) decides if the true state of the device is appropriate for the desired action, (d) executes the action by issuing appropriate commands to the device, and (e) confirms success of the action. This is the normal sequence for the careful monitoring of physical systems, and we will revisit this sequence later. In the meantime, we use the term *functional system* to refer to the combination of a designed system and its goals in the real world.

Several things should be noted from Figure 2. First, the notion of controller of quite general; in the real world it could be a human or an expert system. Second, we use the term *procedure*[1] rather loosely, and it should not be confused with or compared to an algorithm. As will be described later, procedures are realized as networks of goal-oriented activities that have no counterparts in conventional algorithms. Third, the goals of the functional system are not a part of the designed system. Some goals may be clearly implied by the procedures of a designed system, but they have a separate and detached existence. There is obviously a close relationship between the two in the sense that one can, to some extent, imply or describe the other, but they are distinct entities. In fact, two different designed systems may have identical goals.

Behavioral Models

As shown in Figure 3, most device models contain interconnected *components* that simulate the structure of a target device. For example, an RMS latch is modeled by a software "latch" whose possible operational

[1] Terms such as "Shuttle Operations Procedure" illustrate our usage of *procedure*.

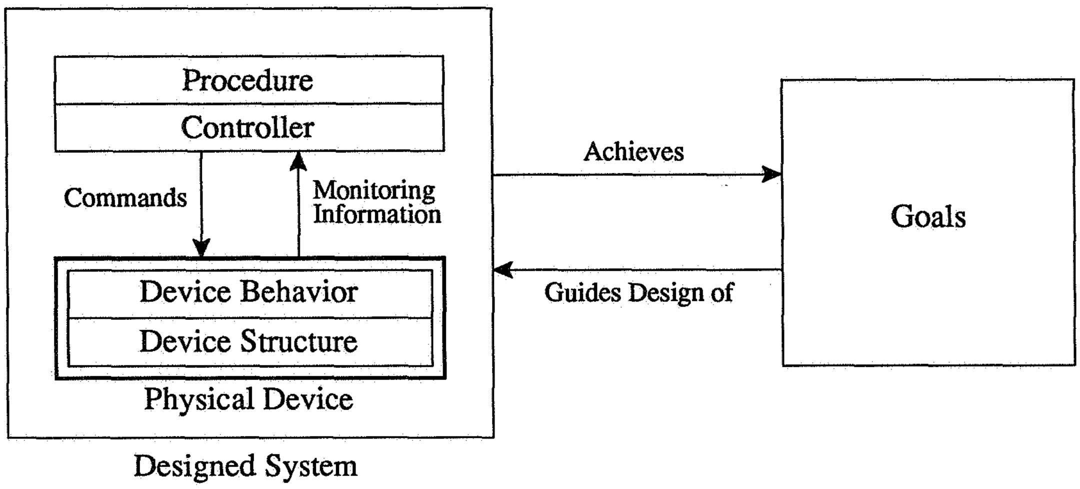

Figure 2: Real-World Functional System

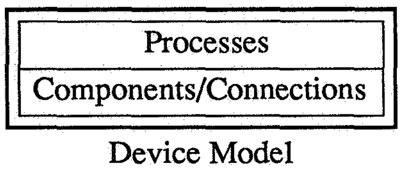

Figure 3: Behavioral Model

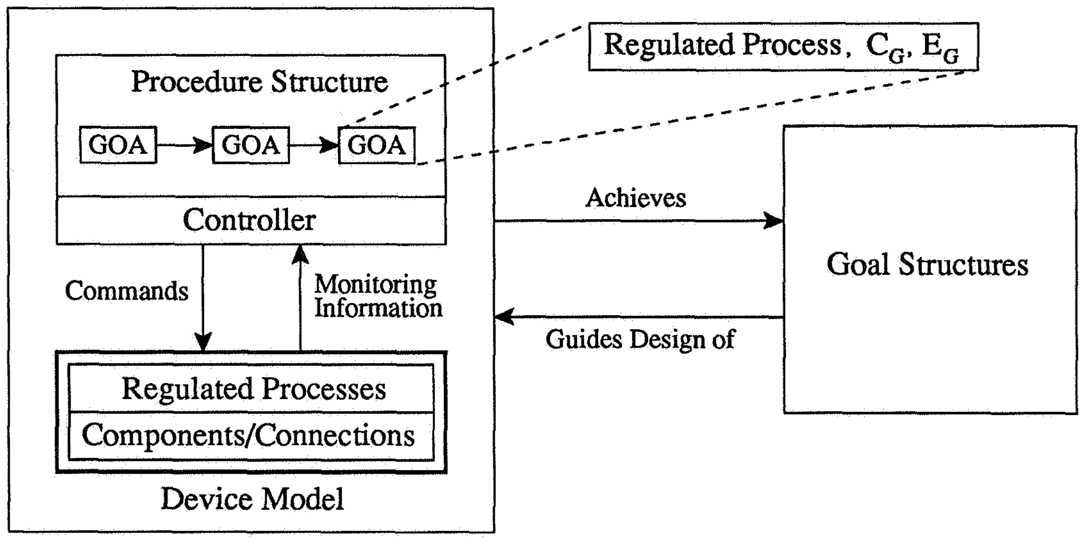

Figure 4: Functional Model

modes mimic the modes of the physical latch (open, closed, or in transition). Hence, during simulation, a component must have some sort of memory to distinguish its current mode from its potential modes. Such a memory is typically implemented as a collection of *state variables*. They may be used to remember not only component modes but also any relevant information that supports the purpose of the simulation. Conventional models could manage state variables that represent things such as temperatures, pressures, rates, oscillations, and even acceleration.

Similarly, the behavior of the target device is represented by *processes*. Each component has associated with it a set of processes that simulate device dynamics by changing the values of state variables in the model. A change in a state variable may activate one or more processes, which themselves can change other state variables. In this fashion the entire structure and behavior of a device can be simulated efficiently. Of course, depending on the exact combinations of state variable values, a model may rest in any one of a very large number of possible states. However, most of these states can be grouped into configurations that denote the general condition of the system (e.g., deployed, not deployed, nominal, off-nominal, etc).

Such models, which we call *behavioral models*, only simulate the device portion of the functional system and thus ignore the issues of commanded behavior, goal-oriented activities, procedures, and goals. Having no explicit links to outside regulation or intention, the behavioral model is primarily used to generate potential outcomes that would occur by starting the device from a given state. It can be started, halted, analyzed in detail, and then restarted as often as desired, but, in the absence of constant manual intervention, it remains largely insulated from outside influences. Thus users of behavioral models are more observers than controllers.

CONFIG is an example of a behavioral model. Qualitative component models are defined in terms of normal and faulty modes and processes, which are defined by invocation statements and effect statements with time delays. System models are contructed graphically by using instances of components and relations from object-oriented model libraries. System failure syndromes are simulated in CONFIG using a modified form of discrete event simulation. Yet despite these capabilities, CONFIG only models device structure and behavior. Extending CONFIG to model goals as well is the chief purpose of our research into functional modeling.

Functional Models

Figure 4 shows how an entire functional system may be modeled by what we term a *functional model*. Corresponding to the physical device is a device model with components and processes, just as with behavioral models. In addition, however, the functional model contains *procedure structures* and *goal structures* that correspond to procedures and goals in the real world. A controller executes the procedure structure on the device model. Goal structures are important for modeling complete functional systems. However, since they are separate from the designed system, goal structures will not be considered further in this paper.

There is an important difference between a functional device model and a behavioral device model: the functional device model is constantly interacting with the controller. Explicit connections are required to pass commands from the controller to the device model and to transmit data from the model to the controller for use in monitoring.

Hence processes in functional device models are extensions of processes in behavioral device models. We call the resulting extension a *regulated process*. The functional model also contains a *procedure structure* to model procedures in the functional system. These procedures consist of networks of *goal-oriented activities*. These three concepts are key features of functional modeling, and each will now be discussed in detail.

Regulated Processes

Regulated process are models of device behavior in real-world functional systems. They are analogous to ordinary processes in behavioral models such as CONFIG except that they have explicit mechanisms for communicating with controllers. This communication may be from the controller to the regulated process (commands) or from the regulated process to the controller (control information). This section will describe these mechanisms in detail.

The general layout of a regulated process is given in Figure 5. There four parts: the regulator, the invocation, the outcome, and the effector.

Regulator (command from controller)
- Start Switch
  - Inhibit On (default)
  - Inhibit Off
- Terminate Switch
  - Halt at Completion (default)
  - Abort Immediately

Invocation (conjunction of conditions)
- Component (self)
  - Mode
  - State variable values
- System
  - Other system components and their modes
  - Other system state variable values
  - Component connections
- Subsystem
  - Subcomponent and their modes
  - Submodel state variable value
  - Subcomponent connections
- Supersystem
  - Supercomponent modes
  - Supermodel state variable values
  - Supercomponent connections

Outcome (list of effects)
- Results of process (changes to mode and state variables)
- Delay

Effector (means of achieving effects)
- Pointer to procedure in submodel

Figure 5: The Regulated Process

The *regulator* may be regarded as two optional switches that control when the process starts and terminates.

The start switch, if present, has two modes: inhibit on and inhibit off. The regulated process may not begin execution if the start switch is inhibit on. The default mode is inhibit on. There are thus two scenarios for initiating a regulated process: (1) there is no start switch present, in which case the regulated process proceeds normally whenever the simulation triggers it; and (2) there is a start switch, in which case the regulated process emerges in the simulation in a "waiting" state, and the controller explicitly commands a "start!" by toggling the start switch from inhibit on to inhibit off. (Switching back to inhibit on at this point has no effect whatsoever on the running process.)

The terminate switch, if present, also has two modes: halt at completion and abort immediately. The default is halt at completion. The two scenarios for use are as follows: (1) there is no terminate switch present, in which case the process runs as expected and terminates when its outcome is completed; and (2) there is a terminate switch, in which case the process halts as soon as the controller an abort (or runs until expected completion if the controller never switches to abort).

The *invocation* is a conjunction of state expressions[2] required for the process to begin. An invocation (denoted C) may be formally decomposed as (c_1 **AND** c_2 **AND** ... **AND** c_n) where c_i is a state expression called a *condition*. It is stressed that conditions merely denote the requirements necessary

[2] Device models, both in behavioral and functional models, rely on state variables. It is assumed that these variables are visible throughout the entire system and that their values may be examined and tested at any time. We define a *state expression* as any Boolean expression built from these state variables. The only requirement for state expressions is that they be well-defined (i.e., ultimately evaluate to true or false) at all times. Thus state expressions may be as simple as a comparision between a state variable and some value, or it may be a very complex expression involving many of these comparisons or nested subexpressions joined by Boolean operators.

for the process to be physically executed and they imply nothing else. For example, the MRL process "release arm" would have as part of its invocation a condition that the motors are operating nominally, and a condition that the terminating microswitches are operational, among other things. For convenience, these conditions are grouped according to their "location" within the model hierarchy. A given regulated process is associated with a specific component. Conditions using the modes and state variables of that component are given first, followed by conditions using modes, connections, and state variables of other components at the current level within the model. But some conditions may also use modes and state variables from a subsystem or a supersystem of this component, and they appear last. This grouping, or "invocation typing," is merely a form of semantic clustering and is done solely to mirror human perspectives.

The *outcome* is a list of state expressions which become true during the process or when the process terminates. An outcome (denoted E) may be formally decomposed as $(e_1{:}d_1, e_2{:}d_2, e_3{:}d_3, ..., e_n{:}d_n)$ where e_i is a state expression called an *effect* and d_i is the delay, or the time lapse between the start of the process and the point when the effect becomes true. Effects are counterparts to conditions in the invocation. However, instead of specifying physical requirements for process execution, they represent expected states of the model that result during or after process execution.

The *effector*, another optional part of the regulated process, names the procedure(s) in the submodel which actually implement the outcome. A given component *S* may consist of *n* subcomponents $s_1, s_2, ..., s_n$. A regulated process *R* associated with *S* achieves its outcome simply by resetting *S*'s state variables and ignoring *S*'s subcomponents. However, the controller may wish to explicitly simulate the activities of all subcomponents s_i and not just assume that they will produce the outcome in *R*. In this case, a procedure must be created at the submodel level (the s_i level) to achieve these goals explicitly. Whenever this occurs, that procedure is named by the effector of *R*.

Hence controllers may issue commands to the regulated process by setting the regulator switches, and they may retrieve control information from the regulated process simply by referencing the appropriate state variables.

Goal-Oriented Activities

As controllers, humans can never monitor every facet of a real-world functional system at every moment. We are forced to monitor only a subset of the system and simply to make assumptions about the unmonitored parts. Unfortunately, a real-world functional system is not obligated to obey our assumptions, and disasters can occur whenever it does not. Hence good controllers are fundamentally suspicious when following plans. They do not blindly execute the specified steps, but instead will view each step as a separate goal with distinct conditions and expected effects. Furthermore, whenever possible, they will independently prove selected conditions and confirm the effects even if the system gives no hint of a failure.

Goals such as these must be reduced to formal objects before they can be manipulated by computer-based systems. We next show how goals and regulated processes are related.

Balkanski (Balkanski, 1990) defines *activity* as a triple containing an act-type, an agent which performs the act-type, and the time interval over which the act-type is performed. This formalism is useful in modeling collaborative activities, but it also coincides with regulated processes in the sense that a regulated process embodies an activity of the type Balkanski describes. We extend this notion to capture the concepts of monitoring and fault management. A *goal-oriented activity (GOA) with respect to device model M* is a a triple (R, C_G, E_G) where

- R is a regulated process in M (with an invocation C and an outcome E)

- C_G, the *goal conditions*, is a conjunction of conditions contained in C

- E_G, the *goal effects*, is a list of effects contained in E

Note that C_G may equal C, it may be a "subset" of C, or it may be null. A similar relationship holds between E_G and E.

An example helps to clarify these ideas. As will be described later, the designer of a procedure usually expresses the steps of the procedure as distinct GOAs and not as simple commands. The device model contains a library of regulated processes[3], each with preset invocations and outcomes. Using the outcomes as a guide, the designer selects a regulated process which best accomplishes the intended purpose. It is important to note that a given regulated process may have many effects in its outcome, but not all of them may be intended effects of the designed procedure. The designer must therefore select a set of intended effects, E_G, which deserve special attention during procedure execution.

Although all the conditions must be true to begin execution of the regulated process, the designer may not wish to "suspect" them all for monitoring and fault management purposes. Analogously, the designer constructs a set of critical "contended" conditions, C_G, which must be reconfirmed as true (even if the model gives no indication to the contrary) before the regulated process is started.

Thus a goal-oriented activity is a "cross section" of the invocation and outcome of a selected regulated process. More precisely, a GOA is simply a view of a regulated process. This feature lets the system designer create general regulated processes that can be customized by a GOA in a procedure.

After the system has been designed, the controller encounters a GOA while using the system (i.e., executing a designed procedure). The controller notes the contended conditions C_G and takes steps to confirm their truth before attempting to initiate the regulated process. If any conditions in C_G are proved false, then there is a discrepancy between the controller's assumptions and the true state of the world. At this point, executing the regulated process would fail. A comparison with behavioral models will illustrate this critical concept. In a behavioral model, the process simply fails with no indication of why. In a functional model, the failure is likely to be intercepted before it occurs, and the controller, having tested C_G explicitly, now enjoys considerable insight into the reasons for the averted failure.

Procedure Structures

We have shown that procedures are used to achieve goals using devices whose current states are only assumed by the controller. For this reason, procedures cannot be expressed in terms of conventional algorithms; they must be expressed in terms of more abstract goal-oriented activities. This section describes the methods for constructing procedures out of GOAs.

The most straightforward approach is to implement the procedure as a sequence of independent GOAs. The controller is never forced to choose which GOA to execute next, and, furthermore, each GOA can ignore the results of previous GOAs in the procedures. This may suffice for simple procedures, but in general it seems clear that (1) procedures must have a memory, and (2) mechanisms must exist for combining GOAs in complex ways.

The first requirement may be satisfied easily by introducing data stores, called *procedure variables*, whose purpose is identical to variables in conventional computer programs. They may be set, reset, and

[3] Strictly speaking, a process with neither a start switch nor a termination switch is not truly regulated, but we still refer to them as "regulated" because the switches can be installed at any time.

tested by the procedure. Their purpose is simply to remember whatever the procedure chooses to remember, and they assist the controller by making available the results of previous steps in the execution of the procedure.

The second requirement is more difficult. We adapt Kant's proposals (Kant, 1988) for algorithmic control of goal-like entities. This approach allows GOAs to be chained in *control networks* that support classical programming control structures such as branching and iteration.

The general layout of a procedure structure is given in Figure 6.

Preamble

- Processes that point to this procedure
- Required resources
- Time estimate
- Global preconditions
- Procedure variable declarations
- Names of GOAs

Results

- Intermediate effects (temporary)
    - Primary
    - Side
- Final effects (permanent)
    - Primary
    - Side

Control

- GOA control network

Figure 6: Procedure Structure

Procedures consist of three parts. The *preamble* contains documentation information such as: the names of the regulated processes (presumably from one level higher in the model) that point to this procedure, a summary estimate of resources required to execute the procedure (e.g., fuel), a summary estimate of how long the procedure takes to execute, a set of global preconditions that describe the required general state of the system before the procedure may be executed, the declarations for procedure variables, and the names of all goal-oriented activities contained in the procedure. The *results* document the effects of the procedure and are classified by time (intermediate or final) and by intention (desired effect or side effect). Finally, the *control* section contains the control network of GOAs described above.

Having described regulated processes, goal-oriented activities, and procedure, we now present an example of how they might be used.

Sample RMS Application

Preparation of the Shuttle RMS arm for mission tasks involves deploying the arm away from the Shuttle Payload Bay to its operational position. This includes the RMS Powerup and Deploy procedure, which uses the Manipulator Positioning Mechanisms (MPM) to swing the arm outboard from the Shuttle, uses the Manipulator Retention Latches (MRL) to unlatch the arm from the MPM, and then uses normal joint-driving commands to move the arm away from the MPM and its latches.

The first portion of this procedure consists of the following sequence of goal-oriented activities:

1. Select RMS: Supply power to the RMS control sensors and actuators
2. Configure power: Supply power to the RMS control sensors and actuators
3. MPM Deploy
4. MRL Release
5. Configure Power: Deactivate power supply to MPM/MRL motors

A more detailed representation of the MRL Release GOA (number 4) is provided. This unlatching process includes the following detail:

1. Check whether the MRL drive motors are operational
2. Check that the Shuttle Digital Autopilot is in free drift or the vernier navigation jets are selected
3. Start the MRL release of the latches

		Regulating Command	
		Start	Abort
Regulating Information	Goal Conditions	Subsystem At least one operational drive motor per latch Shuttle Digital autopilot mode free drift or vernier jet selected	Subsystem NOT (At least one operational drive motor per latch) Shuttle NOT (Digital autopilot mode free drift or vernier jet selected)
	Goal Effects	Self NOT (MRL Released)	Self MRL released

Figure 7: Controller commands for the MRL Release Process

4. Monitor the MRL release, and abort the release if it has not terminated properly after 18 seconds

This procedure step exhibits much of the detail that the GOA view of a regulated process is designed to capture. There are two types of regulation commands used: start! (set inhibit off) and abort! (set abort switch on).

Figure 7 shows how information about selected conditions and effects of the MRL Release process determines whether each command should be issued by the controller. This information is at various levels of detail within the shuttle system, from the operational status of motors that are subcomponents of the MRL subsystem to modes of the Shuttle guidance system. Note that much of the information about the changes that occur in an MRL release process is not captured in this GOA (e.g., power consumption effects are not of interest).

Conclusions

Functional models extend the power of behavioral models because they can express system goals as well as structure and behavior. Their properties make them especially useful for supporting the development and validation of fault management procedures. Three points deserve special emphasis.

First, types of conditions in the regulated process correspond to some parts of the qualitative process definition of Forbus. Individuals, preconditions, and quantity conditions of a Forbus process correspond to system components, connections, and state variables that are conditions in the regulated process. Influences and qualitative proportionality relations in a Forbus process will correspond to modulators in the regulated process, but this is an area for future research.

Second, the concept of a controller as a verifier of contended conditions and intended goals sheds new light on concepts of coordinated action among teams, gained from analysis of the heterogeneous human-machine team in a designed system. Here, the "suspicion" is not an adversarial one about whether goals are shared between the human and the device, as described by (Levesque, 1990). Instead we provide a general framework for representing monitoring of goal-oriented activities based on selected conditions and outcomes.

Finally, functional models can support goal-directed simulations with explicit mechanisms to react to changing events. Nilsson (Nilsson, 1989) calls this *teleoreactivity* and uses it to make "smart" processes whose preconditions include the negated goal (e.g., if a process has a goal *g*, then ~*g* must be true for the process to begin). Functional modeling extends this notion to conditions as well as effects and does so in a fault management environment.

References

Balkanski, Cecile T. (1990) *Modeling Act-Type Relations in Collaborative Activity*, TR-23-90, Harvard University Center for Research in Computing Technology.

Davis, Randall, and Walter Hamscher (1988). "Model-based Reasoning: Troubleshooting." *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence* (H. Shrobe, ed.) Morgan Kaufman Publishers, San Mateo, California, pp. 297-346.

Forbus, Kenneth D. (1984). "Qualitative Process Theory." *Artificial Intelligence*, 24 (1984) pp. 85-168.

Kant, Elaine (1988). "Interactive Problem Solving Using Task Configuration and Control," *IEEE Expert*, Winter 1988, pp. 36-49.

Levesque, H. J., P. Kohen, and J. Nunes, (1990). "On Acting Together." *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 94-99.

Malin, J. T., Bryan D. Basham, and Richard A. Harris (1990) "Use of Qualitative Models in Discrete Event Simulation for Analysis of Malfunctions in Continuous Processing Systems." *Artificial Intelligence in Process Engineering* (M. Mavrovouniotis, ed.), Academic Press, pp. 37-79.

Muratore, J., T. Heindl, T. Murphy, A. Rasmussem, and R. McFarland (1990). "Real-Time Data Acquisition at Mission Control." *Communications of the ACM*, December 1990, Volume 33, number 12, pp. 19-31.

Nilsson, N. J. (1989) *Teleo-Reactive Agents*. Computer Science Department, Stanford University (forthcoming).