A Human-Centered Approach to Monitoring Complex Dynamic Systems

DRAFT DRAFT DRAFT DRAFT DRAFT
FINAL REPORT
NASA Grant NCC2-1220
October, 2004

Daniel Shapiro
Dorrit Billman
Mei Marker
Pat Langley

Institute for the Study of Learning and Expertise
2164 Stauton Court, Palo Alto CA 94306 USA
{shapiro, billman, langley}@isle.org

Abstract

This paper reports on a human-centered approach to monitoring complex systems which offers the fidelity of methods that compare predictions against observations with the emphasis on higher-level user feedback found in more inferential techniques. In particular, we cast our work as a decision support tool and we assume that the monitoring system should reason over models that are natural to human users. This has led us to adopt several design tenets: the structure of the model should follow the componential and causal structure of the device; feedback from the system to the user should be tied to the device model; device behaviors should be represented as a set of processes with associated equations; and every process in the model should correspond to some behavior of the device. We claim that these commitments support an intuitive and effective design for a fault detection, isolation, and diagnosis tool. We discuss the knowledge representations, algorithms, and interface capabilities of this system, and present a scenario of its use that illustrate our claims in the context of monitoring the electric power system on board the International Space Station. We conclude by discussing a human evaluation study of the monitoring tool, and its actual effect on its users' diagnostic strategies.

1 Background and Motivation

As engineering technology grows, our ability to employ complex artifacts is limited by our inability to understand them. This difficulty arises for several reasons. First, complex devices - such as high performance aircraft, nuclear power plants, and space stations - perform complex functions. This requires many interacting parts, which makes it hard to organize knowledge about their structure. Second, complex devices interact with people and the environment to generate a wide variety of possible behaviors, which makes it difficult to discern correct from abnormal operation. Third, complex devices are typically instrumented to produce large quantities of data. This leads to an intricate diagnosis problem, as the information must be analyzed in order to synthesize an integrated picture of what the device is doing, whether it is functioning properly, and, if it is not, how and why it is misbehaving.

The International Space Station is a case in point, and we will use it as an example throughout this paper. Its complexity comes from the fact that it is simultaneously a self-contained habitat, a self-propelled vehicle, and an experimental laboratory. As a result it contains many subsystems, including power generation, power storage, power distribution, thermal control, atmospheric maintenance, navigation, propulsion, and attitude control, as well as a schedule of demands that often press the Space Station's performance limits. Its component systems also interact, meaning that a fault (say) in attitude control may show up in power storage. The evidence is contained in an extensive telemetry stream, as the electric power system alone generates approximxately 50,000 measurements every ten seconds. Specially trained staff members in Mission Control must examine these data in order to detect and isolate faults. Moreover, the ever-present pressure to reduce costs means that fewer, and less expert, personnel will perform this analysis in the future. NASA clearly needs tools to support the monitoring process.

The monitoring problem, broadly construed to include fault detection, isolation, and repair, has generated a large body of research within AI and related disciplines. Here we distinguish approaches that use background knowledge to infer a relation between observations and faults from approaches that operate by comparing observations with the expectations of a predictive simulation.

Inference-oriented approaches have grown out of the expert and knowledge-based systems movements. For example, Sermatech's TIGER system [2] applies production rules to detect anomalous conditions from observations, diagnose/identify the affected system, and abstract the results into a single fault conclusion. This system has been used to monitor motors, generators, turbines, transfer switches, and many other devices. In contrast, Larsson's [15] GoalArt takes a more model-driven perspective by describing devices as goal-driven machines and employing a model to map observations into a high-level analysis of system properties. GoalArt has been used to monitor patients in intensive care and is being applied to diagnose faults in nuclear power facilities. In general, inferential mechanisms offer a significant ability to supply analyses at a level of abstraction suitable for human users (i.e., by providing output in terms of lost capabilities vs. anomalous variable values) and their applications are typically cast as decision support tools.

Approaches that compare predictions against observations have emerged from a synthesis of ideas in qualitative physics and operations research. In general, these methods function by finding models that are consistent with observations and commands; they detect faults when the data are inconsistent with nominal models, and they diagnose problems when explicit fault models successfully predict the data. However, the task of matching models to data becomes computationally complex in the presence of multiple device components with multiple behavioral modes. In response, researchers have examined many types of predictive models that capture qualitative behavior [9] [7], semi-quantitative behavior [10] [24], and hybrid continuous/discreet dynamics [21] [22] [12]. These model-based mechanisms also extend to fault repair, viewed as the selection of control inputs that drive a system to a preferred mode. The Livingstone system [28] has had a great deal of success in this regard, as it performed fault detection, isolation, and repair on NASA's Deep Space One mission. While these predictive techniques provide a great deal of power, the research has focused historically on fully autonomous diagnostic systems. As a result, the underlying representations are instrumental for diagnosis and repair, but not for communicating analyses to human users. We will take this as the main point of departure for our work.

This paper describes a human-centered approach to monitoring complex systems that provides the fidelity of predictive methods as well as the high-level user feedback associated with inferential techniques. In particular, we cast our work as a decision support tool and we make the commitment, expressed in Langley et al. [14], that such systems should reason over models that are natural to human users. In response, we have adopted the following design principles:

- the structure of the model should follow the componential and causal structure of the device;
- feedback from the system to the user should be tied to the device model;
- device behaviors should be represented as a set of processes with associated equations; and
- every process in the model should correspond to some behavior of the device.

We claim that these commitments have led to an intuitive, and effective design for the fault detection and diagnosis tool that we present in the body of this paper.

1.1 Guide to Reading

We begin this report by introducing our approach in Section 2, then turn to our representational framework for modeling device structure and behavior in Section 3. Following this, Section 4 discusses our method for making model-based predictions using this framework, while Section 5 illustrates our system's fault detection and diagnosis capabilities. Given this background, Section 6 presents the graphical interface that gives users access to all monitoring functionality. We bring these capabilities together in Section 6.5, by walking through a scenario corresponding to an existing fault in the Space Station's electric power system.

In addition to building a monitoring system, we conducted an experiment aimed at evaluating its ability to support diagnostic reasoning in non-expert users. Section 7 discusses this experiment, together with our initial results.

We conclude this report by placing our work on human-centered monitoring in the context of related efforts in Section 8, and by summarizing the contributions and limitations of this research in Section 9.

2 An Overview of our Approach

Our approach to monitoring is straightforward. In abstract terms, it detects faults by comparing observations against the predictions of a device model and signaling an alert whenever a significant discrepancy occurs. Because we have gone to some length to construct accurate models, both manually and via a machine learning technique [3], we can attribute alerts to a fault in the device as opposed to a fault in the model. In addition, because we establish a close correspondence between the structure of the model and the structure of the device, we can localize a problem within the device by associating it with a specific component, subcomponent, property, and behavior within the device model.

In more detail, we model devices as hierarchical collections of systems, where each system is composed of processes which act on variables that may belong to further subsystems. Each process contains activation conditions and equations that express causal influences among variables and, since the mathematics supports both algebraic and differential equations, the model makes numeric predictions over time. Our modeling discipline associates device components with systems, device subcomponents with subsystems, device state with variables, and device behaviors with processes. Thus, processes are analogous to the modes of model-based diagnostic techniques[1] and, as the activation conditions gate behaviors, they capture qualitative transitions among modes.

As a whole, a model in this language represents the causal structure of a device and compiles into a directed graph connecting variables by processes. We use this internal form to support fault detection and diagnosis. In particular, a fault is a deviation between a predicted and observed value of a variable that exceeds some threshold. We localize faults by tracing the graph from symptoms towards root causes, and we isolate faults to particular processes (device behaviors) by comparing their expected influences (i.e., their input/output transformations) against the observed input/output relations where possible. We treat fault diagnosis as the task of fitting a model of a faulty component to the data stream.

Our user interface presents fault data via a graphical display that color codes affected elements of the model. The model's hierarchical structure provides a natural abstraction mechanism in this regard, since the presence, but not the nature, of a fault can be made visible in enclosing systems. Other user-interface operations support navigation through the hierarchical model, navigation through the time-history of a monitoring run, navigation through the causal graph connecting symptoms and causes, and projection of both nominal and fault models into the future, together with displays of predicted and observed variable values. The goal is to provide the user with the ability to detect problems, rapidly focus his or her attention, and delve into detail when those problems require special attention.

Before we enter into a detailed discussion of our system, we should point out that, unlike many of the model-based systems referenced earlier, it does not currently perform probabilistic mode identification or propose fault repairs. While this represents a limitation

[1]More exactly, sets of processes are analogous to modes.

with respect to fully automated techniques, there is reason to believe that attention to probabilistic effects might actually complicate the monitoring process in interactive settings, in comparison with a deterministic (but retractable) approach to prediction and analysis. In addition, our use of a deterministic framework has let us focus on the human-centered aspects of the monitoring problem.

3 Modeling Complex Artifacts

In order to model complex devices, we have designed a language for representing their behavior. For clarity, we divide our discussion into two parts that reflect the historical development of the language. The first subsection discusses the primitive elements of the language and their use in modeling simple device behavior. The second adds hierarchy to this picture and shows how to aggregate multiple behaviors into a depiction of abstract devices.

3.1 Quantitative Process Models

The primitive elements of our modeling language represent device behaviors in terms of *variables* and *processes* that act on those variables. A variable represents device state, like the charge on a battery. A process describes one or more causal relations between input and output variables, such as charging, which increases battery charge in response to an applied current. We state these relations in terms of differential equations (if the process describes change over time) or algebraic equations (if it describes instantaneous effects). A process may also include conditions, expressed as threshold tests on its input variables that identify when the process is active. A process model consists of a set of processes that link observable input variables with observable output variables, possibly through unobserved theoretical terms.

We use a model of the batteries onboard the International Space Station as a simple example. Figure 1 illustrates how an electrical engineer might conceptualize the batteries in a circuit diagram: the battery is a voltage source (V_{cb}) connected to a resistor in parallel (R_p) and in series (R_s). The parallel resistor models spontaneous discharge (as batteries slowly lose their charge over time). The series resistor models internal losses as charge is transferred into and out of the battery.

However, this diagram does not express all of the behavior we might want to capture. In particular, it lacks a quantitative description of currents and voltages in the circuit at any one time, in addition to any representation of the battery's dynamic behavior. For example, its state of charge varies over time and its ability to absorb or dispense charge varies with state of charge. We capture this behavior in the equations attached to processes.

Table 3.1 provides a more detailed process model of the battery in textual notation.[2] It begins by declaring the variables and parameters (constants) involved in the model, then lists the processes representing behavior with their associated conditions and equations. The first four variables - V_{cb}, V_t, soc, and i - describe physical properties of the battery. V_{cb} is a theoretical term corresponding to the voltage of the battery before any losses occur to internal resistances, while V_t represents the actual voltage at the battery terminals. The variable i

[2]We discuss a graphical representation of process models in Section 6.

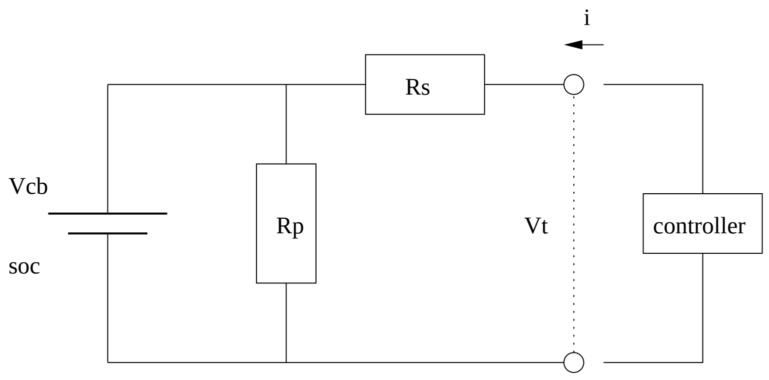


Figure 1: An abstract depiction of a battery model.

represents current, where positive values indicate flow into the battery by convention, and *soc* represents the total amount of electric charge stored in battery, expressed as a percentage of a nominally full battery. A fifth variable, *charging*, provides a control signal that selects among battery modes. (This battery is either charging or discharging, as the controller never connects it to power sources and loads at the same time.) The battery model also contains three parameters: R_p is the resistance in Ohms of the parallel resistor, R_s represents the battery's serial resistor, and R_{load} is the effective resistance applied by the controller during discharge.

The processes in Table 3.1 describe causal interactions between the variables of the battery model. SourceVoltage represents the behavior of the battery as a voltage source before any losses to internal resistances. This voltage is generally not constant in real power supplies. We model it as a linear function of state of charge, which provides good predictive performance when compared with actual telemetry data [3]. The ChargeTransfer process describes the effect of current flow on battery charge as a differential equation that integrates current flow across time periods (where the scaling constant converts Amperes into state-of-charge units). The process SelfDischarge represents spontaneous battery leakage; like ChargeTransfer, it also has *soc* as a dependent variable. Whenever two or more processes influence the same variable, we assume the effects are additive.

The last three processes in Table 3.1 represent the behavior of the battery under an applied load or power source. Here we define charge and discharge as mutually exclusive modes by testing the whether the control flag, *charging*, has value 1 or -1 within each condition clause. DischargeCurrent captures the batteries' ability to supply current under an applied (and internal) load, including an indirect dependence on state of charge (through V_{cb}). DischargeVoltage performs the analogous calculation for voltage. Finally, ChargeVoltage determines battery voltage under an applied current that is selected by the battery controller.

Taken together, these processes specify a causal model of battery function that represents its response to applied current or loads. This particular model has also been tuned to fit telemetry data taken from the batteries onboard the Space Station. The tuning mechanism involves a machine learning method that searches over a space of feasible battery models while balancing accuracy against complexity [3]. Although a discussion of that mechanism is beyond the scope of this paper, it results in a model that can be trusted to represent actual battery behavior for use in monitoring.

Table 1: A simple process model for a battery.

```
Model Battery; Variables soc, Vcb, Vt, i, charging;
Parameters Rp = 100, Rs = 0.214, Rload = 2.5

Process SourceVoltage;
   equations Vcb = 36.2 + 76.2 * soc;

Process ChargeTransfer;
   equations d[soc,t,1] = 10 * i;

Process SelfDischarge;
   equations d[soc,t,1] = - 10 * Vcb / Rp;

Process DischargeCurrent;
   conditions charging == -1;
   equations i = -Vcb / (Rs + Rload);

Process DischargeVoltage;
   Conditions charging == -1;
   equations Vt = (Vcb * Rload) / (Rs + Rload);

Process ChargeVoltage;
   Conditions charging == 1;
   equations Vt = Vcb + i * Rs;
```

3.2 Hierarchical Process Models

While the process modeling language can capture the behavior of simple devices, the ability to represent and monitor significantly more complex systems requires extensions to the language. In particular, we need methods for organizing our knowledge about devices so that we can appreciate their behavior in more abstract terms. At the same time, our design tenets demand that any language extensions support a mapping from elements of the model onto the structure and function of a device that engineers will find easy to understand.

We address this issue by introducing hierarchy into the representation presented in the previous section, to produce a hierarchical process modeling language, which we call HPML. Our approach follows from the observation that all complex devices have a clear functional decomposition (or else it would have been difficult to construct them). In response, we represent a device as a *system* that contains some number of other systems, *variables*, and *processes*, where the processes model interactions among the component systems. These subsystems recursively decompose into similar structures, until the hierarchy 'bottoms out' in process models of the form discussed in the previous section.

We illustrate this approach by outlining a model of the Space Station's electric power system (EPS), which is a self-contained solar power installation. It uses photovoltaic arrays to generate power, batteries to store it, and multiple circuits to deliver power to loads at various points on the Space Station. The EPS maintains these elements in careful balance by routing power; if the supply exceeds demand it stores power in the batteries; if demand exceeds supply (e.g., during eclipse) it takes power from them. The boundary cases concern extremes of supply and demand that require shutting down generation capacity, or shedding load (as in civic power systems).

Because the EPS is a complex device, we represent it via a hierarchy of models. At the top level it is a single system, called the *PowerChannel* (Table 3.2), which acts to govern power flow among three subsystems: power generation (*PowerGen*), power storage (*PowerStore*), and power usage (*Loads*). The *PowerChannel* makes decisions by altering the values of variables that represent targets for power production, power use, and power flow into (or out of) the batteries. These decisions are implemented by processes within the component models.

In more detail, we represent the abstract behavior of the PowerChannel via seven processes. The first two determine whether the batteries will act as a source, or sink for power. If the solar power system can supply more power than all loads combined, the *condition* field of *ChargeMode* will be true, that process will be active, and the variable *PowerStore.ChargingBat* will be set to 1 (the dot notation associates variables with systems). If demand for power exceeds maximum supply (e.g., during eclipse), *DischargeMode* will be active and the mode indicator set to -1 instead. Processes within the *PowerStore* model check this variable before activating behaviors that trickle charge, taper charge, fully charge, or discharge the batteries.

The next two processes instruct the *PowerGen* system to produce a specific level of solar power. *FullSolarPower* sets the actual output, *PowerGen.SolarPowerOut*, to its maximum possible level provided that the loads and the batteries can collectively absorb that much power. However, if there is an oversupply, *ShuntSolarPower* 'throttles down' the generation capacity to produce exactly as much as required. Processes within *PowerGen* calculate the maximum available solar power (*PowerGen.SolarMaxOut*) by tracking the process of pointing the solar panels at the sun as the Space Station moves along its orbit. Other processes implement production targets by reading the *SolarPowerOut* variable and activating or deactivating strings of photovoltaic cells to match the desired supply.

The processes for managing demand are symmetric with those that govern supply. If the power generation and storage systems can collectively support all loads, *ServiceLoad* sets *Loads.ServicedDemand* to the maximum demand. If there is a shortfall, *ShedLoad* forces demand into balance[3]. Processes within *PowerStore* determine how rapidly the battery can accept or produce current. Processes within *Loads* calculate the maximum demand by consulting a load schedule. Other processes implement load shedding via a uniform brownout policy (this is an abstraction; the Space Station actually sheds loads in priority order).

The last process within the *PowerChannel* system, *BalancePowerFlow*, determines

[3]EPS load schedulers work hard to avoid this condition, as it is highly disruptive.

the amount of power that must flow into, or out of, the *PowerStore* system. It inputs the final values of power supply and demand, and sets *PowerStore.ioBatPower* to their difference, where positive values denote inflow. Processes within *PowerStore* distribute this requirement across three component batteries, modeled as further subsystems containing dynamic equations for charge transfer, as in Table 3.1.

In summary, the *PowerChannel* system represents EPS behavior in abstract terms. It describes power flow instead of propagating currents and voltages, and it suppresses details about how that power is generated, stored, and used. The hierarchical process modeling language also lets us decompose the EPS along natural system boundaries, such that specific devices and more detailed behaviors appear at deeper levels within the model structure. Finally, the language lets us associate processes with recognized device behaviors, preserving the correspondence between the model and the object modeled. This correspondence facilitates monitoring.

We note, however, that HPML has several limitations. First, because the language characterizes devices as a casual graphs (where inputs flow through processes towards outputs), it cannot represent feedback within a given time period. This makes it difficult to model control systems (e.g., for pointing the solar panels) and to represent complex equilibria (e.g., the currents and voltages in real circuits), which are typically characterized by in-period feedback and the solution of simultaneous algebraic equations. Next, HPML imposes a reactive structure that predicts the next state from current state and observations, meaning that all past state must be propagated forward via differential equations. (Variables cannot be assigned at time 1, and recalled at time 4.) This reduces convenience, but it enhances the view that devices are simple causal structures. It also lets us define concise mechanisms for performing simulation, fault detection, and diagnosis. More broadly, we note that HPML supplies a deterministic vs a stochastic framework for modeling device behavior. This limits expressivity, but the simplification supports productive interaction with users, which is our immediate research goal.

4 Simulating Process Models

Given that we can represent a complex device with an HPML model, we can run the model as a simulation to predict values for its variables. Our approach to simulation is straightforward; we compile the model into a causal graph that connects input variables, through processes, to output variables, and we evaluate this graph once for each time period. Since the equations are either algebraic (describing an instantaneous relation among variables) or dynamic (describing a instantaneous rate of change across one time step), the computation is analogous to updating a spreadsheet. The simulator simply sweeps the graph and applies the equations associated with each process whose activation conditions are met. Over multiple time steps, this produces a time series of quantitative predictions for variables in the model, which we can then compare against observations. The next few subsections discuss the compilation and simulation of HPML models in more detail.

Table 2: The top-level of the electric power system model.

```
system PowerChannel;
  components PowerGen, PowerStore, Loads;

  process ChargeMode;
    conditions PowerGen.SolarMaxOut > Loads.MaxDemand;
    equations
      PowerStore.ChargingBat = 1;

  process DischargeMode;
    conditions PowerGen.SolarMaxOut <= Loads.MaxDemand;
    equations
      PowerStore.ChargingBat = -1;

# Generate full power if there is a place to put it
  process FullSolarPower;
    conditions PowerGen.SolarMaxOut <= Loads.MaxDemand + PowerStore.MaxBatPowerIn;
    equations
      PowerGen.SolarPowerOut = PowerGen.SolarMaxOut;

# Reduce power generation if there is nowhere to put it.
  process ShuntSolarPower;
    conditions PowerGen.SolarMaxOut > Loads.MaxDemand + PowerStore.MaxBatPowerIn;
    equations
      PowerGen.SolarPowerOut = Loads.MaxDemand + PowerStore.MaxBatPowerIn;

# Service all loads if the required power is available
  process ServiceLoad;
    conditions Loads.MaxDemand <= PowerGen.SolarMaxOut + PowerStore.MaxBatPowerOut;
    equations
      Loads.ServicedDemand = Loads.MaxDemand;

# Shed load if the required power is not available
  process ShedLoad;
    conditions Loads.MaxDemand > PowerGen.SolarMaxOut + PowerStore.MaxBatPowerOut;
    equations
      Loads.ServicedDemand = PowerGen.SolarMaxOut + PowerStore.MaxBatPowerOut;

  process BalancePowerFlow;
    equations
      PowerStore.ioBatPower = PowerGen.SolarPowerOut - Loads.ServicedDemand;
```

4.1 Generating Causal Graphs

As introduced in Section 3, every process within an HPML model contains equations that define a causal relationship among variables. Taken together, a set of such processes define a *causal graph*, which is a directed acyclic graph connecting variable and process nodes.

Figure 2 provides the causal graph associated with the battery model in Table 3.1. Here, ovals represent variables, rectangles represent processes, and links indicate data flow. An arc from a variable to a process implies the variable is an input to the process, whereas an arc from a process to a variable indicates that the variable is an output. Model parameters are not shown. In order to focus on causal relations, this representation also suppresses the control flow information contained in *conditions* fields. As a result, it appears that $ChargeVoltage$ and $DischargeVoltage$ simultaneously influence Vt, even though those processes are mutually exclusive. (In general, multiple influences on a single variable are added.)

This particular causal graph describes a dynamic system where the change in battery state of charge is a function of its current value. The simulator interprets this graph by integrating the equation for *dsoc/dt* forward between time steps, and adding the result to the value of *soc*. In light of these semantics, a differential quantity can only appear as the output of a process, and an integrated quantity can only be an input. Later figures will reduce the number of variables by mapping both quantities onto a single name (i.e., *soc*) and depicting differential equations as loops.

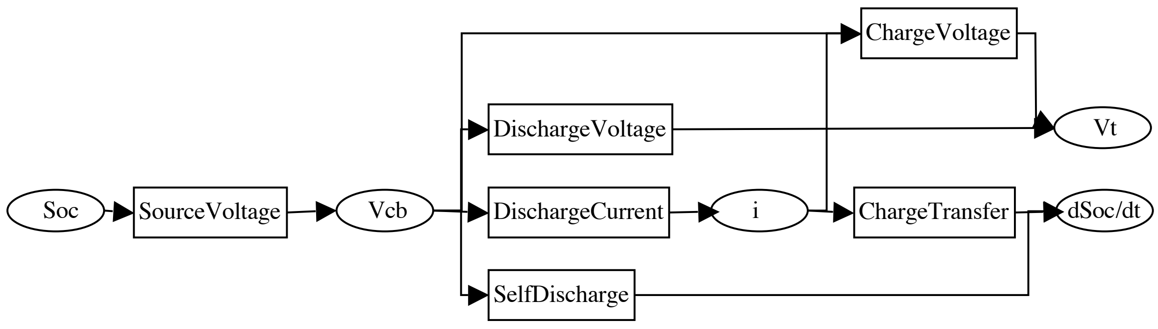


Figure 2: A causal graph for the simple battery model in Table 3.1.

The algorithm for assembling the causal graph is straightforward. It starts by finding *exogenous* variables that only appear as inputs to some process, and considers all processes not in the graph. It adds every process whose inputs are all present, together with its output variables (if they had not been added before). This iteration continues until the graph contains all processes and variables in the textual form of the model (success) or no process can be added (failure). Failure implies that the textual input specified a cyclic graph that has no meaning to the simulator.

Hierarchical process models also specify causal graphs, since the added structure simply partitions the data flow connecting processes and variables. As a result, HPML models compile into flat causal graphs exactly as described above, provided that the algorithm is modified to chase the input and output variables across system boundaries. This causal graph becomes an input to the simulation module.

4.2 Simulation using Causal Graphs

The simulator predicts the values of variables by simulating the causal graph forward in time. More exactly, it generates predictions for *endogenous* variables (intermediate and final outputs of the causal graph) from *exogenous* variables (inputs of the graph). This requires an observed value for every exogenous variable at every time step of the simulation, plus the size of a time step for use in integrating differential equations. Given these data, the simulator sweeps the causal graph once on each time step, from exogenous variables through processes towards final outputs, applying every equation belonging to a process whose activation conditions are met. If more than one process has the same output variable, the simulator adds the corresponding effects, reflecting a modeling commitment to view them as separable influences.

When the simulator encounters differential equations, it invokes a standard integration package [6] to compute the value of the integrated quantity for the next time step. Note that since a differential equation may depend upon the output of other processes, the rate of change is commonly a function of data observed on that time step.

It is important to notice that the simulator often has access to both a predicted and an observed value for endogenous variables, since observed values may be present in the telemetry stream. This creates a choice in source data. If the simulator uses predicted values to make further predictions, it generates estimates that only depend upon the exogenous data. If the simulator relies on observed values for intermediate variables instead, it generates more informed predictions but loses the causal dependency among predicted values. Each of these approaches has advantages. We rely on predicted vs observed values during fault detection, as this lends an intuitive consistency to the system's expectations on any single time step. However, we rely on observation over prediction during fault diagnosis, where the goal is to isolate problems to the affected processes. We also rely on observed data over predicted values to determine which processes are active, regardless of context.

5 Monitoring Complex Devices

We have used the ability to simulate device models to construct a decision aid for fault detection and diagnosis, called, simply, the *monitor*. This tool helps the user to observe the status of the monitored device, detect anomalies and track them to their source, and anticipate future hazards that are a consequence of current events. In overview, the monitor operates by comparing a time stream of observations from the monitored device against the simulator's predictions. Whenever there is a significant mismatch, it raises an alert and attempts to associate the fault with particular variables, processes, and systems throughout the model hierarchy. It presents the results to the user via a graphical interface that supplies operations for navigating the model hierarchy, navigating the underlying causal graph (to move from symptoms towards causes), navigating the history of monitoring system output, and for projecting the model into the future to supply an early warning facility.

This section discusses the monitor's primitive operations for detecting anomalous conditions in variables and faults in processes, for isolating faults, and for projecting model behavior into the future. Later sections will discuss the graphical interface, and the use of these capabilities to diagnose fault scenarios.

5.1 Detecting Anomalies in Variables

The monitoring system supports two methods of associating anomalies with variables. The first compares observed values against *absolute* limits and generates an alert if the value falls outside of an allowable range. This implements a standard threshold test. In the second method, the simulator generates a prediction based on exogenous inputs and the monitor compares it against the observed value. The monitor signals a fault if the error lies outside a predefined tolerance, expressed as a numeric range, or as a percentage of the observed value. We call this a *relative* threshold.

The monitoring system provides the user with a convenient checkbox-style interface for selecting variables to monitor and for setting both absolute and relative thresholds. For example, the user might set an absolute threshold that triggers an alarm if battery goes below a 65% state of charge, in recognition of the fact that there is a real safety risk if the Space Station enters the night side of its orbit with less charge. In contrast, the user might set a relative threshold on total power demand, reflecting an expectation that the scheduled demand will cover a fairly wide dynamic range, making an absolute threshold impractical.

5.2 Detecting Faults in Processes

Surprisingly, the monitor can detect faults in *processes* as well as anomalies in variable values. It does this by comparing the predicted input/output mapping of a process against the observed input/output relation and flagging any significant difference. If this difference exists, it is something of a smoking gun for fault isolation because it names a specific device behavior that is in error. However, the relevant data must be available. This only occurs if the process is *measurable* according to the definition:

> *A process P is measurable if and only if (i) all of its inputs are observable, (ii) at least one output is observable, and (iii) no other active processes contribute to the prediction of the observable output.*

This definition is straightforward with the exception of condition *(iii)*, which is designed to exclude situations where some other process affects the variable(s) in question, making it difficult to discover that P is in error. When the definition is satisfied, we can unambiguously declare that P is malfunctioning if $|o_{predicted} - o_{observed}| > k$, where k is a fixed threshold, o is a measurable output variable, and $o_{predicted}$ is computed by passing observed values of the input variables through P. At this point, we can take advantage of the fact that HPML fosters a close association between the model and the device to ascribe the process fault to a device behavior.

If multiple processes do affect a given output variable, o, we can sometimes treat the set of processes that influence o as a *compound process* and apply a relaxed version of the same definition. Here a compound process is measurable if its combined inputs are all observable, as well as at least one of its combined outputs. Again, no other processes can influence the chosen output variable. In this case, the compound process is malfunctioning when its predicted output (calculated from its input data) is sufficiently different from its observed output.

As an example, consider Figure 2 and assume that the variables *soc*, i, and *dsoc* are observable. When the processes $ChargeTransfer$ and $SelfDischarge$ are active, both influence *dsoc*, so neither process is measurable on its own. However, if we combine them with $SourceVoltage$ and $DischargeCurrent$, the resulting compound process *is* measurable. In particular, if the value predicted for *dsoc* as a function of *soc* in this time period is significantly different from its observed value, the compound process as a whole is malfunctioning. We cannot say which component process is at fault, but we have isolated the problem to a set of possible candidates, called an *ambiguity region.*

5.3 Isolating Faults

The monitor uses its ability to detect process faults to support a more general fault isolation capability. Here the task is to search the causal graph for the smallest *measurable* compound process that explains a variable fault. The algorithm for doing this starts with an observed variable, o, that violates a monitoring threshold (relative or absolute), and traces the causal graph backwards towards exogenous variables while growing the ambiguity region. The algorithm stops when it can observe all of the inputs to a connected subset of the causal graph. In the limit, the ambiguity region will contain all processes between the exogenous variables and o. At this point, the system can compute a prediction for o as a function of these observed inputs and compare it to the observed value of o. If the difference is outside of the tolerance,k, we say that the ambiguity region explains the fault.

We have found it necessary to restrict this algorithm to prevent it from generating overly large ambiguity regions. We do this by constraining the length of the longest causal chain in an ambiguity region to eight processes (this is a tunable parameter). This approach gives the monitor considerable freedom to trace problems from symptoms to causes, while restricting attention to problems the user can plausibly correct.

We note that a number of other systems cast fault isolation as a search through a causal graph [22] [1], but ours seems unusual in its ability to employ quantitative rather than qualitative predictions. This difference should increase the reliability of our isolation results, given an accurate predictive model.

5.4 Projecting Models into the Future

The monitor also has the capability to generate an early warning of possible faults by projecting current state in to the future. This should be contrasted with the ability to detect faults as they happen by performing tests on observations. The projection mechanism employs the simulation engine discussed in Section 4 to generate a deterministic prediction of future events, but since this process moves beyond the available data, we make one modification: we introduce models that posit observations for exogenous variables over the projection interval. This ‘closes’ the simulation, giving it the input data necessary to run.

The model developer has access to all of HPML to create predictors for exogenous variables, under the proviso that the new models are themselves closed. That is, its processes can define the value of an exogenous variable as a function of any variables in the rest of the model, specifically in the form of a differential equation (which is the only means in HPML of introducing dependencies across time periods). For example, the process:

```
process SetGimbalPosition;
  equations d[position, t, 1] = 4;
```

models an assumption that the gimbal responsible for pointing the solar panels towards the sun will advance at four degrees per minute (where its position formerly had to be observed as an exogenous variable). The projection system can then predict the consequences of this assumption for the behavior of the electric power system as a whole.

Since projection looks into the future, the monitor cannot compare predicted data against observed values. However, it can compare projected data against fixed thresholds to detect likely failures. Since the projections are quantitative, users can detect features such as the first moment at which battery charge will redline by dipping below 65 percent of its full capacity.

Any mechanism for projecting the current situation into the future will necessarily introduce assumptions, and thus expose issues of uncertainty and inaccuracy in the underlying model. As a result, the scope of the projections should be bounded in time. To this end, we introduce user-defined parameters that declare how many time steps to run a projection and how often to update that projection. We use one orbit (93 minutes) as the default value for this look ahead, and we update the projection twice in this interval, following our discussions with NASA personnel.[4]

5.5 Diagnosing Faults

Our capability for fault diagnosis builds on the projection facility described above. In particular, we let the user employ any model of a faulty system as the basis for projection, and compare that data against the actual observations as they arrive. For example, if the user suspects a failed NiH battery cell, he/she can select, and project the consequences of that specific fault model forward in time, creating predictions for many system variables. If later telemetry broadly agrees with those predictions, the diagnosis was correct.

This approach to fault diagnosis requires a means for finding and selecting appropriate fault models. One response (dating back to the 1980s [26]) is to develop a library of explicit fault models with tools that let users browse, combine, and apply such models. We are currently investigating a related approach that employs machine learning to automatically

[4]Alan Crocker, Electric Power System flight monitor, personal communication, 2002.

construct fault models that best fit the data. This mechanism operates by searching over plausible replacements for each process in the ambiguity region output by the fault isolation facility, and tuning its free parameters to the observed data. We have shown that this form of model revision is feasible in other domains [25] [4], and appears extensible to fault diagnosis.

In summary, our approach to monitoring complex devices employs causal process models to offer a variety of core capabilities, including mechanisms for anomaly detection in variables and in processes, fault isolation, projection, and fault diagnosis. The following section shows how we embed these capabilities within a graphical interface to provide users with a powerful tool for interactive monitoring.

6 The Graphical User Interface

The monitor's graphical interface provides users with a rapid way to understand the health of complex devices. Here, the key design challenge has been to maintain a sense of simplicity while supporting a daunting task characterized by a complex model, large volumes of data, multiple analysis methods, and potentially cascading alerts.

Our design relies on the hierarchical structure of HPML as the organizing principle. In particular, we tie all alerts and data displays to the graphical model to provide the user with a constant sense of orientation within the device, and we employ the model hierarchy as a means of hiding and summarizing detail. This implies that we are focusing on informed, but non-expert users who understand the basic function and organization of the device but still need some reminder of its structure. All totaled, the interface must display the model's structure, associate alerts with that structure, display time-varying data, and supply tools for navigating the data and the model, in addition to operators for invoking the various monitoring functions. We discuss each of these component capabilities below.

6.1 Graphical Model Display

The monitor displays HPML models via a two-dimensional layout that preserves both causal and hierarchical structure. We illustrate this in Figure 3, using the top-level model for the Space Station's electric power system (the *PowerChannel*). This display corresponds to the textual representation in Table 3.2. Here rectangular nodes denote processes, oval nodes denote variables, and long rectangles with rounded corners (and internal structure) denote subsystems. As before, an edge from a variable to a process indicates that the variable is an input to the process, while an edge from a process to a variable indicates that the variable is an output. Data flow goes from left to right in the diagram, while the subsystems are arranged vertically down the page.

This representation shows all of the variables and processes that belong to the enclosing system and all of the causal relations between the systems it contains. At the same time, it suppresses details associated with the structure of subsystems. However, the user can bring up those details by double-clicking on the subsystem's box (e.g., the *PowerStore* system) to open an analogous display in a separate window.

In addition to communicating system structure, the display also animates that structure as the user compares the model against telemetry. At each time step, active processes are fully displayed with incoming and outgoing edges, while the inactive processes are shaded and their corresponding edges are removed. For example, the display in Figure 3 corresponds to a state where the batteries are charging, and the system is servicing all loads while shunting some solar panels. This animation may change in the next time step, as motivated by the data.

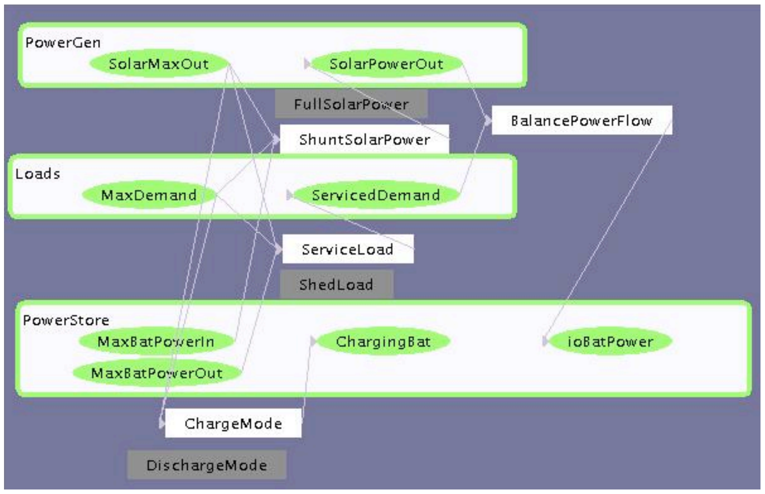


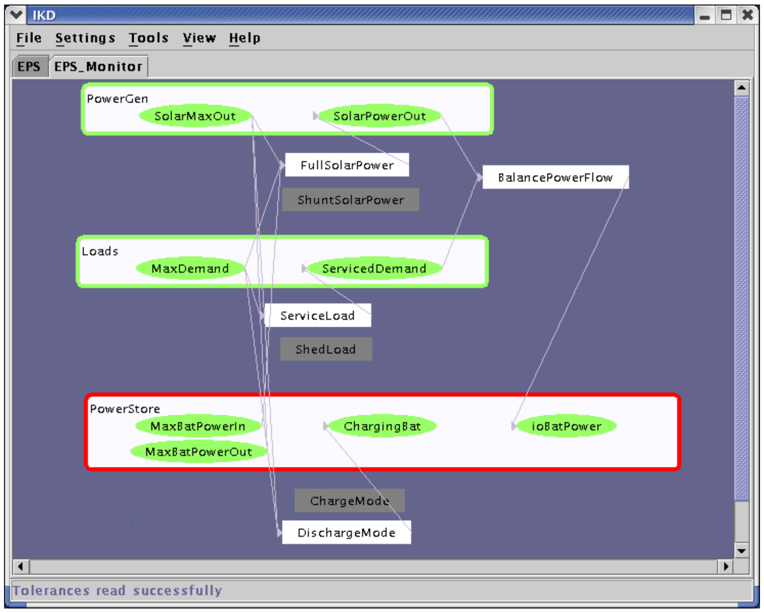
Figure 3: Graphical display for the top level of the electric power system.

6.2 Alert Propagation and Display

The monitoring system displays status by color-coding variables, processes, and systems. It uses green to signal OK, yellow for warnings, and red to indicate severe alerts in accordance with people's preconceptions. In particular, it displays the oval representing a variable in green if it satisfies all monitoring tolerances, yellow if the comparison between its observed and predicted values lies outside of a *relative* tolerance, and red if the value exceeds an *absolute* threshold (see Section 5.1). It displays the variable in a dull olive color if it has no attached monitoring tolerances (although note that the variable may still be observable in the telemetry stream). The system draws a process boundary in yellow if it is a part of an ambiguity region associated with a variable fault, as discussed in Section 5.2.

The monitor color codes component systems if they contain faults in variables and/or processes. This provides an abstraction principle: a system inherits the most severe color found in any of its components, recursively down the subsystem hierarchy. As a result, a system remains uncolored if it is fault free, it will be bounded in yellow if there are relative threshold violations or process faults (but not absolute threshold violations), and it will be bounded in red if it (or its subsystems) contain any absolute threshold violations.

Figure 4 illustrates this principle by showing the status of the *PowerChannel* system during a monitoring run. This display indicates that the *PowerGen* and *Loads* systems are fault-free, whereas *PowerStore* contains a severe fault at some location. If the user wants to see precisely where, he/she can drill down through the model hierarchy by double-clicking on systems. This leads to a display like that in Figure 5, showing that the state of charge in *Battery3* is out of bounds, where *Battery3* is a subsystem of *PowerStore*, which is a subsystem of *PowerChannel*.



Tolerances read successfully

Figure 4: The PowerChannel showing a fault during a monitoring run.

6.3 Data Displays

The monitoring system contains a graphing package that lets the user track the predicted and observed value of any variable over time (by double-clicking on its icon). Figure 5 provides a simple example concerning the state of charge on the battery. It illustrates that the observed value of *soc* deviates from the value predicted by the model by integrating current flow forward from time 0. The user can also view a third curve that projects the value of *soc* into the future under various fault models.

6.4 Navigation

The monitor provides several navigation tools to help the user detect and diagnose problems. These fall into three categories, for navigating the hierarchical device model, the time history of alerts, and the causal graph containing alert data. The operators for navigating the model are quite simple; they let the user open subsystems by double-clicking on their icons (creating a display of that system's contents in a separate window), choose open models (by clicking on windows), and move to any enclosing model from a point deeper in the tree.

The operators for navigating the time history of alerts treat that history as a sequence of snapshots depicting device health over time. In this framework, time zero is the beginning of the telemetry stream, *now* is the time index of the most recent telemetry, and any index later than *now* is in advance of the data, and thus solely associated with model projections. (In actuality, the data in our scenarios comes from a file rather than a real-time feed, so

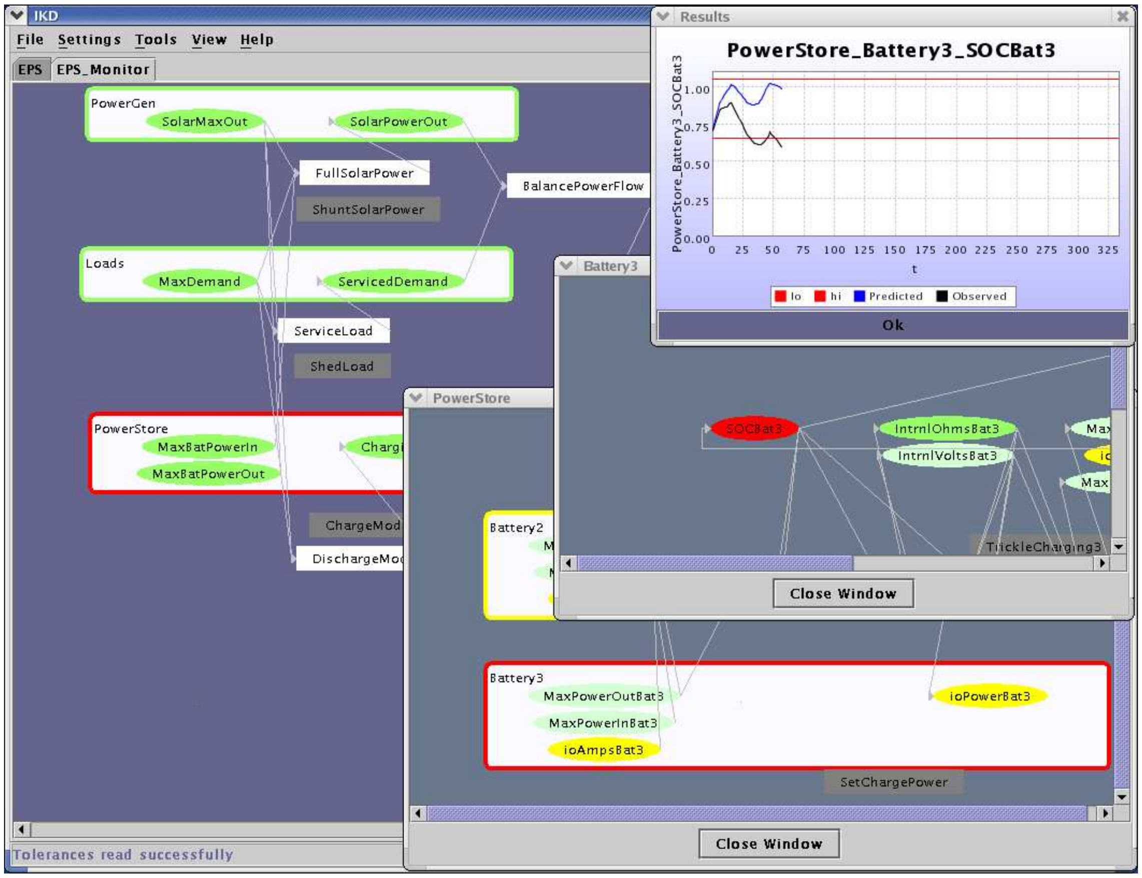


Figure 5: Drilling down through a model hierarchy to find a fault.

data is available at all time indices. However, we maintain the fiction that *now* corresponds to the most recent telemetry for the purpose of exposition.) With this background, our navigation metaphor follows the control functions of a DVD player. Reading left to right in the top corner of Figure 6, the user can play back the time-line (at three frames per second), move forward one timestep, move backward one timestep, pause, and stop the automated playback. The user can also drag the slider to any time of interest, in which case the variable *now* is set to that new time. In response, the monitor updates the graphs, and color coding within all open models to their appropriate values.

The operators for navigating the causal graph let the user shift attention to systems that are causally upstream or downstream of a fault within a given time index. In particular, the user can select the forward (or backward) causal navigation option by right-clicking on a variable. The monitor will respond by finding the the next (or previous) process, opening the model that contains it, and highlighting that process on the screen. An analogous operation navigates from processes to adjacent variables. Simple extensions to this facility will find variables and processes flagged with faults by searching over multiple links in the causal graph. This will provide a rapid means of managing fault cascades, for example, to locate the causally earliest evidence of a problem.

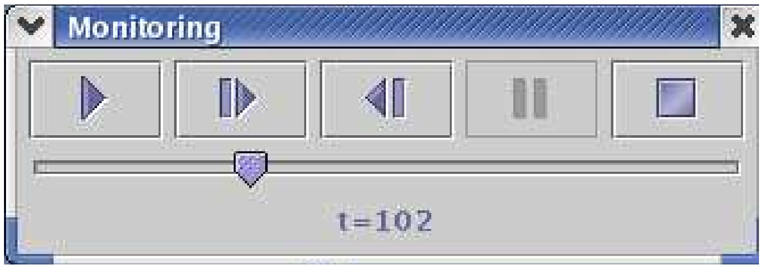


Figure 6: The control bar for navigating the time-series of monitoring data.

6.5 An Illustrated Monitoring Scenario

The monitor's palette of primitive capabilities should help people perform a variety of fault detection and isolation tasks. In particular, it should let users:

- detect problems,
- localize symptoms to components of the model,
- track those symptoms through time,
- trace symptoms to their causes within a given time, and
- identify the specific processes (and corresponding device behaviors) that appear to be malfunctioning.

We illustrate how these capabilities come together by walking through a hand-crafted scenario concerning a fault that has actually occurred on board the space station.

This scenario begins with the electric power system in some distress, at a time when two of its major components contain serious faults (*PowerGen* and *PowerStore*), and its third displays a warning. The left side of Figure 7 illustrates this case. Faced with this ambiguity, the user arbitrarily decides to drill down into the *PowerStore* system, producing the right side of Figure 7, and then to expand one of its three battery subsystems, all of which indicate faults. This generates the bottom illustration in Figure 7, which shows that *soc* is red-lined, but that all of its active processes appear to be functioning properly. This situation repeats in the other two batteries (not shown). The fact that all three batteries simultaneously contain variable faults suggests that these anomalies are symptoms, not causes. In response, the user traces the causal connections from *SOCBat*3 back towards the source of the problem.

The user can employ the monitor's causal navigation facility to trace the data flow from *SOCBat*3 through *NormalCharging*3, towards inputs of the *Battery*3 model (variables with no incoming arcs). Alternatively, the user can exploit the hierarchical structure to recognize that *ioPowerBat*3 represents power flow into, or out of *Battery*3, and that *PowerStore.ioBatPower* computes that quantity for the *PowerStore* subsystem as a whole. This chain of reasoning leads back to the *PowerChannel* model (the left side of Figure 7). Here, *ioBatPower* leads to *BalancePowerFlow*, *SolarPowerOut*, and ultimately, to the input variable, *SolarMaxOut*. We show the user employing causal navigation to find the predecessor of this variable.

This path leads into the *PowerGen* system, as it is causally upstream of power storage as a whole. Figure 8 shows the results. Here, the user has chased *NormalSupply* (the predecessor of *SolarMaxOut*) into the *BetaGimbal* system, which models the controller for the rotating joint that points the solar panels at the sun. The *BetaGimbal* system is the source of the error, as evidenced by the yellow highlight on the *GimbalResponse* process, which declares that its expected i/o mapping does not equal the observed behavior. Something is wrong with the gimbal joint; its position (integrated over time) is not responding appropriately to the applied current.

The user calls up three graphs to clarify the nature of the problem. The data in Figure 9 tells a convincing story that the gimbal joint is frozen in place. In particular, the predicted value for the supplied current is constant (corresponding to a constant rotation speed), while the observed current follows a sawtooth pattern, in lock step with the day/night cycle shown in the variable *SolarPowerOut*. This suggests that the gimbal controller is trying to rotate the panels in response to an increased pointing error, first one way, then another. Given that the battery discharged throughout the day cycle, the gimbal also froze with the panels at a dysfunctional angle to the sun.

7 Human Evaluation of the Monitor

While preceding scenario showed how the monitor's capabilities might be employed to diagnose a fault, we also studied its behavior in practice with our target audience of non-expert users. The key open questions were to determine (1) how the monitor's novel features affect diagnostic reasoning, and (2) which features supply advantage (or the opposite) relative to

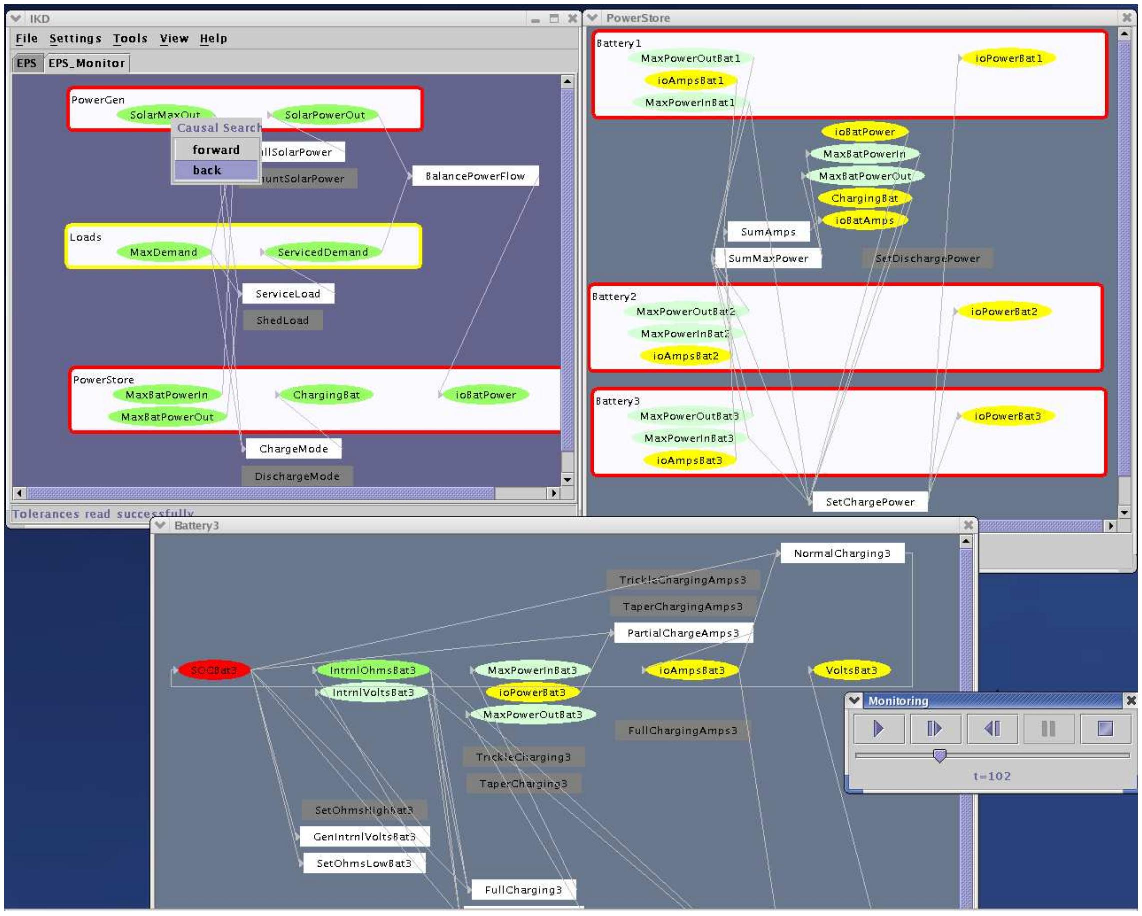


Figure 7: The PowerChannel system with simultaneous faults.

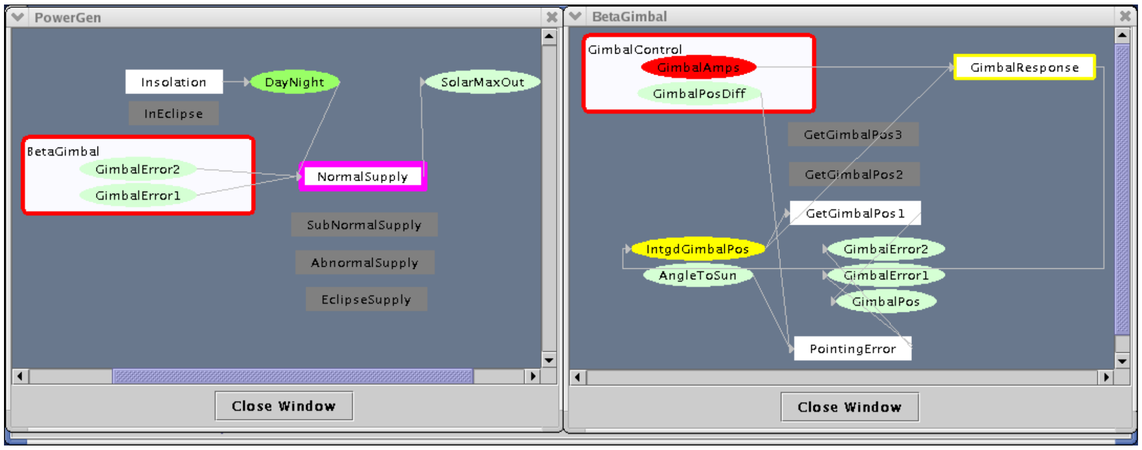


Figure 8: The gimbal controller, and gimbal systems.

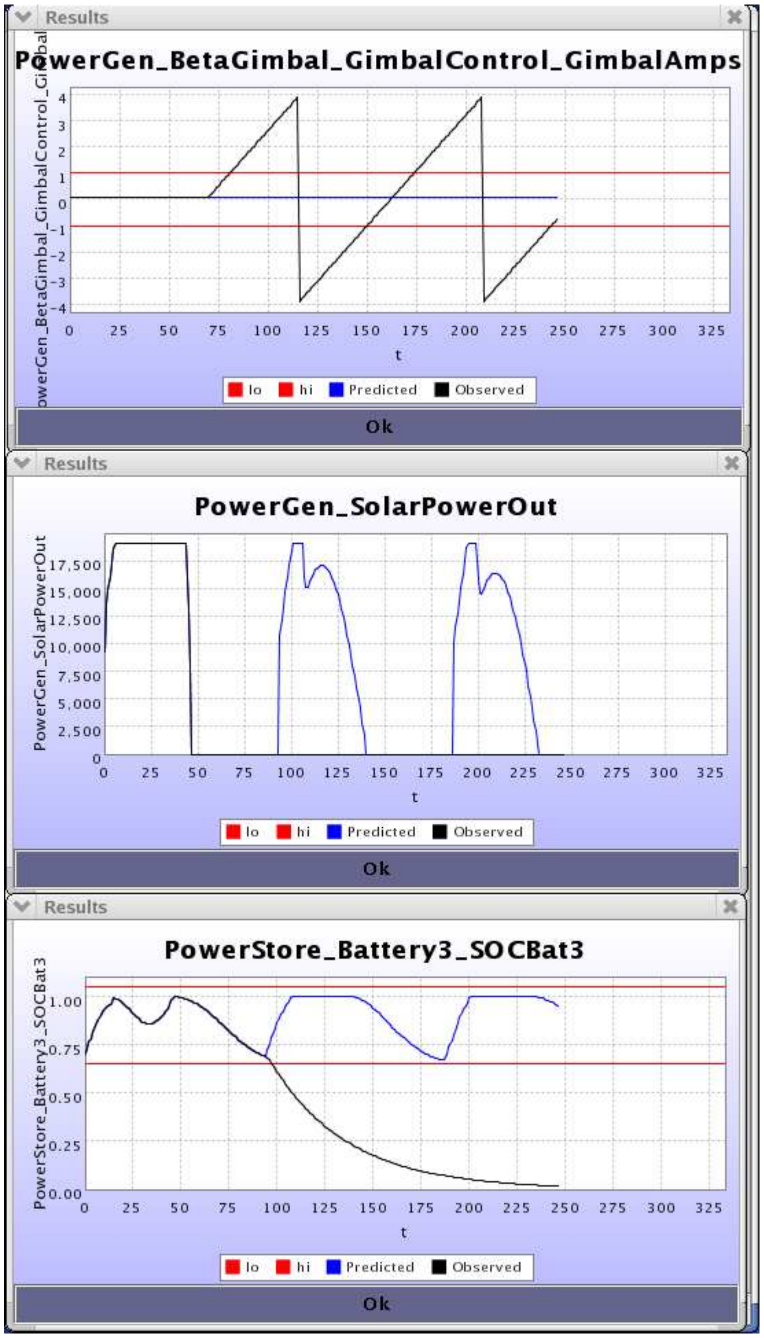


Figure 9: Data indicating a stuck gimbal fault.

existing interactive diagnostic techniques.

We focused on the first question, as we judged the second too difficult to address in the context of this project[5]. In particular, we conducted an exploratory study that recorded monitoring and diagnosis strategies in the presence or absence of hierarchical model structure. Then, we mined the results for observations about how people used the tool: which features they relied upon in both conditions, which they found frustrating or ignored, and whether the protocols indicated a difference in reasoning style or diagnostic accuracy across the hierarchical and non-hierarchical conditions.

We describe this study in some detail below, beginning with a description of the problem domain, and the diagnostic tasks we asked people to perform.

7.1 The Problem Domain

In order to study the impact of the monitor on diagnostic strategies, we needed a realistic model of a non-trivial device interacting with its environment, plus data describing a series of malfunctions. In response, we developed a rather complex model of the Space Station's electric power system; it contained 16 systems, 24 distinct variables, and 79 processes that reflected the components, observable features, and behaviors of the device. (The displays in this report are drawn from this model.) On average, the model hierarchy was four levels deep, and to enhance realism, some components had been tuned to fit Space Station telemetry data. However, most had not. As a result, the monitor could not predict actual telemetry values, and the scenarios employed synthetic data.

We also developed thirteen fault scenarios by altering processes within this model. These scenarios concerned anomalous conditions in the *Loads*, *PowerGen*, and *PowerStore* systems:

- A software error in the controller that rotates the gimbal.
- A stuck gimbal joint.
- A failed amp sensor on the gimbal motor.
- A recurrent shadow that reduces solar panel output.
- An aging battery that cannot hold as much charge as it once did.
- A NiH battery cell with an over-pressure fault.
- A battery that spontaneously discharges at an accelerated rate.
- A battery that cannot deliver or absorb energy at its normal rate.
- An inaccurate sensor for battery state of charge.
- A load that draws far too much power.

[5]The second question calls for a lesion experiment on a complex system, while measuring the performance of multiple users as they pursue a non-trivial cognitive task. Individual variations in diagnostic strategies would very likely govern the results, unless we ran a prohibitive number of subjects.

- A known, periodic load kicks in at the wrong time.
- An unscheduled load consisting of short duration periodic spikes.
- A planned, but unsatisfiable schedule of loads.

We employed a subset of these in our study.

7.2 Method

Our study involved 12 subjects, 6 fault scenarios, and 2 experimental conditions, corresponding to hierarchical and non-hierarchical presentations of the same electric power system model. (The non-hierarchical (or *flat*) model contained the same processes and variables, but without a supervening system/subsystem structure). We drew the subjects from a population of Stanford engineering graduates and undergraduates who could be expected to have some familiarity with the notion of models, and complex electrical devices. We felt this population offered a reasonable surrogate for our target audience of informed, but non-expert diagnosticians.

We divided these subjects into two groups of six, for the hierarchical and flat conditions. We provided each with a modicum of training in both the electric power system's organization, and in the use of the monitoring tool. Then, we asked each to perform the same six diagnostic tasks (down-selected from the thirteen identified above). We took videotape and encouraged the subjects to provide verbal protocols while conducting these tasks, and when they indicated they were done, we asked them to complete a questionnaire that investigated their understanding of the symptoms, consequences, and causes of the problem. We scored answers to this questionnaire relative to an expert's opinion (in this case, Dr. Shapiro, who composed the fault scenarios).

*** DORRIT, ADD WORDS ON OVERVIEW here. In overview, subjects took from *** to *** minutes to complete each task...

We describe our study in somewhat more detail, below.

7.2.1 Procedure

There were two conditions, between-subjects, one using the hierarchical layout and the other using the single-level layout. The users did six problems in the layout for their condition and then saw and worked with the opposite condition layout for a small amount of time at the end of their session. The session lasted 3 hours. It consisted of a tutorial running about an hour, the 6 diagnostic problems taking between 1.5-2 hours, and exploring the cross-condition layout as time was available, from just a few minutes to half an hour. Users took a standard break after the first or second problem (users choice), and were allowed to take a break after any problem. The Power Monitor is a very complex tool, simulating a very complex system. The tutorial was developed from considerable piloting. Piloting identified the information and activities which needed to be included in training to allow users to act effectively and to reason with the tools provided by the Monitor.

7.2.2 Training

After an introduction to the whole task session, the tutorial began with an overview of the electrical system of the space station. This was organized to cover broad information about the station and each of the three components: power generation, loads, and power storage. The experimenter read information to the user in four sections and asked the user to summarize and report back the main information from each. After this, the aspects of the PowerMonitor were introduced: 1) how structure of the electrical system is represented, variable name conventions, and navigating the structure by scrolling and causal links, 2) dynamics of the system through running the scenario and viewing variables over time, 3) alerts and information useful in diagnosis. A scenario in which all aspects of the station electrical system ran as normal was used in the tutorial, and the user practiced finding and viewing each aspect of the PowerMonitor as it was described. After all the information had been presented by the experimenter and explored by the user, the user explained and demonstrated the answers to a series of questions reviewing all the presented information.

In both these phases of training, users were questioned further if their first response did not include all the target information, and if needed the experimenter resupplied information which the user had not retained. In general, users did very well in answering the questions; some users realized after the first question that considerable detail was expected and shifted their strategy accordingly.

In the final part of the tutorial, the user solved a series of reasoning problems about normal operations, which gave the user practice interpreting graphs and using them to reason about the state of the system and about relations among variables. This section of the tutorial was most variable in time, in activity of the user, and amount of scaffolding by the experimenter. Our intent was to ensure all participants had a roughly comparable understanding of the system at the beginning of the task, while still keeping the user-experimenter interaction as standardized as possible.

7.2.3 Diagnostic Trials

The user was instructed that he or she would view a series of scenarios in which there were problems. For each, the user should detect, diagnose, and report on the problem. The user was told what counts as a problem (any thing not as predicted), what should be included in the diagnosis or explanation (the causes and effects generating the unpredicted and possibly damaging states), and how the report would be given. During the first two phases, users talked aloud; during the report phase they were asked to a) give their explanation of the problem b) state the root cause, c) state the effects of the problem, d) provide advice about actions to repair, compensate for, or prevent the problem, and e) rate its severity on a 1-5 scale. Users were asked to give their explanation first, as a relatively broad task. We found that many users continued to problem solve while giving their report and this first, open question allowed them to do that. We asked them to state the root cause and effects separately to get a cleaner, user-provided specification than produced if we had tried to extract this from the initial explanation. The final rating of severity was anchored to a 1 meaning that something was not as predicted but no negative impact and a 5 meaning possibly fatal. The user had access to the computer while making the report. Participants frequently used the system while making the intial explanation (from just using the cursor to point at variables, to opening systems and variables to test a hypothesis), Participants rarely (never?) used the computer while doing the remaining four items in the report.

After the user had provided the verbal report, the computer was moved away and the user was given a check list of candidate descriptions.

Users talked aloud as they detected and diagnosed the problems. The experimenter was sitting beside and watching the screen with the user, but made only three types on intervention. First, if the user had not said anything, or seemed inaudible the experimenter asked what are you thinking? or could you talk louder? Second, if the user seemed confused about the interface (e.g. not clicking fast enough to get a double click) the experimenter could clear up the problem. Third, since time spend before concluding varied considerably, the experimenter urged the participant to conclude if the user had been working more than 20/25 minutes on a problem. (This happened n times; in m of these the user was looking for more information that the system provided, or trying to go outside of the system instrumentation.)

All participants completed the 6 problems. Depending on the remaining time as many of these activities as possible were included: 1) payment and bebriefing, 2) view and comment on the alternative system, 3) instruction on different aspects, and problem solving of one or two problems on the alternative system. Problem solving on the alternative system used 2 novel problems. The procedure was the same as previously but no checklist was provided.

7.3 Results

*** DORRIT: you rule here.

8 Related Work

While this paper has shown how to employ model-based techniques to develop an interactive monitoring system, many other ideas in the fault detection, isolation and diagnosis arena relate to our work. In particular, we will discuss methods for handling uncertainty, for supplying goal-oriented feedback to users, for extending diagnosis into repair, and other languages for representing and simulating device models.

Although our work assumes that device modes are fully determined by available observations, many efforts in model-based monitoring take mode identification under uncertainty as the central issue. In this perspective, a device's behavioral mode changes in response to hidden state that must be inferred from uncertain measurements and control inputs over time. For example, Williams et al. [27] employ a Bayesian updating scheme to compute the most likely model consistent with observations and commands. They represent device behaviors and transitions with Hidden Markov Models, augment them with constraints that capture system dynamics within each mode, and then compute each modes' posterior likelihood from its prior distribution, the command input, and the available observations. This approach has the benefit that it naturally encompasses diagnosis (i.e., mode identification can select a fault mode), but it is computationally complex.

Researchers have developed a number of techniques for controlling the implied search, which, in principle, encompasses the space of all feasible modes for all device components to find the most likely vector of modes at each instant in time. Williams et al. consider modes in descending order of prior likelihood in the context of an anytime algorithm. Dearden and Clancy [8] employ particle filters to probabilistically sample possible modes, while being sure to include the most interesting ones. In contrast, Narasimhan et al. [22] [23] decompose the problem by employing quantitative data to track device behaviors across mode transitions (assuming a linear dynamic and compensating for noise with Kalman filters), qualitative reasoning to isolate faults to causally preceding modes, and quantitative parameter estimation techniques to find the candidate mode that best fits/explains the numeric observations. Other approaches encounter this search problem in different guises: Lawesson et al. [17] seek fault modes that entail other alerts by reasoning about models stated in a process algebra, Jones et al. [13] search for logical constraints that, when removed, restore consistency between predictions and observed behavior. This relates to the seminal work by de Kleer et al. [7] who describe observations and component behavior in first order logic and map diagnosis onto the problem of assigning truth values (OK/Not OK) to component descriptions.

In summary, the task of mode identification in the presence of uncertainty entails unavoidable search problems. As our research expands to consider this task, we will encounter the same issues but in an interactive setting. This suggests alternative strategies that rely on the user to prioritize candidate modes or that present only the most probable fault hypothesis to the user. If the user rejects it, the system can shift attention to the next most likely interpretation. In general, the notion of presenting a single, but retractable, hypothesis may provide an effective tool for encapsulating the uncertainty of mode identification when interacting with people.

Systems that infer failures from behavioral measures can supply users with a highly desirable form of abstract feedback about device faults. While expert systems can fulfill this role [2], Larsson's [16] work on diagnosis (recently incorporated into the commercial system, GoalArt [15]) performs this function in a particularly interesting way. He takes advantage of two properties of multi-level flow models (MFM), which represent device behavior in terms of a efforts and flows (much like hybrid bond graphs [20]). First, the primitive elements of MFM models support causal analysis of stylized faults, such as, "a *low capacity* alert on a *storage* object will cause a *low flow* alert in a downstream *transport*". A set of similar rules suffice to restrict alarm cascades and isolate primal faults. Second, MFM models link these functional descriptions into a hierarchy of goals and subgoals, as in saying that a cooling system's *purpose* is to maintain desired water flow and that its ability to transport water through a reactor vessel depends upon the subgoals of having electrical supply and a cooled pump. This produces a direct connection from observations to device goals, enabling high level feedback from primitive faults.

There are several interesting points of comparison between this work and our own. First, where GoalArt exploits the semantics of specific functional primitives to reduce alarm cascades, we trace faults through the causal graph. It is not yet clear if either mechanism is more powerful. However, GoalArt can use expectations from its primitives to question sensor values, while we must take sensor data as ground truth. A second point is that our work supplies more detail on the relation between device structure and function, while GoalArt adds a connection between function and goals. The implication is that we can improve the quality of our system's output by incorporating teleological structure into our representation language, so that we can simultaneously localize faults to processes and functional subsystems, and identify failed goals.

Although our work does not extend into fault repair, it is important to note that other monitoring systems often do. For example, many deployed systems employ fault trees to map a branching series of yes/no questions into repair actions stored in the leaves. A more sophisticated approach views repair in control theoretic terms. For example, Williams and Nayak [28] describe a combinatorial optimization technique that drives a state transition model to a desired configuration, as applied in the context of NASA's Deep Space One mission. Williams et al. [27] elevate this merger of monitoring and control into a proposal for developing fault-aware systems. Here reactive control programs set mode configuration targets, while fault detection, diagnosis, and repair methods implement that behavior plan.

Looking beyond other monitoring techniques, we should acknowledge the relation between our framework and other representational languages. In particular, the core notion of describing device behavior in terms of processes that are active under qualitative conditions comes from Forbus' [11] work on qualitative process theory, although we incorporate more quantitative predictions. Work by Bobrow et al. [5] is also quite relevant, as they develop a compositional modeling language that exhibits many of the same features found in our work. It supports a part-of hierarchy, typed variables, and processes with activation conditions whose behavior can be described by both static and differential equations. However, this work emphasizes declarative knowledge representation issues, as opposed to a causal account of behavior. In addition, there is less concern about establishing a tight coupling between elements of the model and the modeled system or device. In other words, while the languages overlap, our research follows a very different theme.

Finally, we note that the CONFIG system [18] [19] is specifically intended as a graphical environment for composing and simulating models of complex devices. It supplies a discrete-event simulation with capabilities for continuous-time models, where the models are expressed as deterministic, state transition systems. CONFIG has been employed as the central engine in a number of NASA simulation tests. Relative to our language, CONFIG offers more diverse model types and a richer capability for simulating models forward, but with less attention to hierarchical model structure.

9 Concluding Discussion

In closing, we should consider some general issues that have emerged from our research on interactive monitoring. One surprising result is that a single representation of knowledge – hierarchically organized sets of quantitative process models – can support not only anomaly detection, but also fault isolation and even diagnosis. This indicates that process models have substantial practical power and suggests they should be considered for other tasks as well. We believe the source of this power lies in the direct mapping between model structure and device structure, which leads directly to the formalism's transparency and to the monitoring framework's simplicity.

A second result is that we have been able to generate high-level user feedback on the basis of a functional description of device behavior, while such feedback is typically associated with goal-oriented methods. Rather than view these two approaches as a dichotomy, our work suggests that they can be productively merged. In particular, if the abstract vocabulary for describing faults is what carries value to users, we should consider augmenting our hierarchical, but mechanical perspective with a layer of modeling that describes artifacts in purposive terms, and use the new layer to interpret monitoring alerts. More broadly, we may be able to represent both physical and man-made systems as process models, with functional and teleological interpretations.

Despite the advantages of hierarchical process models for monitoring tasks, there remain many ways in which we can extend the framework to improve it further. One limitation of the current scheme is that it cannot encode directly when a device or subsystem is operating in different modes. We can mimic mode behavior by placing conditions on processes, but some modes can activate or deactivate entire subsystems, and operating modes play such a central role in engineering that they may deserve special treatment.

Another drawback is the current framework's inability to handle uncertainty. Even when accurate deterministic models are available, noise in sensors can produce uncertain predictions, and even well-established models have only limited accuracy. This suggests that we extend the modeling framework to support confidence intervals on both process parameters and sensor values, which will require more sophisticated methods for simulation to take this ambiguity into account. Such extensions will bring it closer to the capabilities of other model-based monitoring systems, but still retain the interpretability that is one of its greatest strengths.

As mentioned earlier, we have done initial work on using machine learning to revise our model of the electric power grid, but we hope to expand on this idea in our future efforts. Here we plan to extend the environment to include libraries of generic processes and subsystems that can serve as the building blocks of new models. We will also develop methods for revising hand-crafted process models from the knowledge in these libraries and from observations. We envision not automated techniques for model revision, but rather ones that let the user direct high-level aspects of the search, with the system handling low-level details like eliminating inconsistent candidates and estimating parameters from the data.

In parallel with these extensions, we should expand upon our efforts to evaluate the monitoring system with human test subjects. Earlier, we hypothesized that the system's hierarchical display and suppport for causal analyses would improve user performance on diagnostic tasks. However, we need to test these predictions in controlled experiments, ideally via a lesion study that measures diagnostic accuracy and speed as system capabilities are removed. It would also be useful to extend that comparison to include a system like the one currently employed by NASA to monitor the electric power system on board the Space Station (which provides very few cues about EPS structure, as it is intended for expert users). If these studies produce encouraging results, we should consider an application roll-out to mature, and transition our human-centered approach to monitoring into operational use.

Acknowledgements
We thank Rick Alena at NASA-Ames for many helpful conversations and for supplying a great deal of background knowledge concerning the electric power system onboard the International Space Station. We also thank Daryl Fletcher, Charles Lee, and Stephen Bay for their assistance in examining ISS telemetry. This work was funded by NASA grant NCC 2-1220, and we gratefully acknowledge NASA's support.

References

[1] G. Aaseng, K. Cavenaugh, and S. Deb. An intelligent remote monitoring solution for the international space station. In *Proceedings of the IEEE Aero Conference*, 2002.

[2] Sermatech Intelligent Applications. Tiger: Gas turbine condition monitoring, http://www.intapp.co.uk.

[3] S. Bay, D. Shapiro, and P. Langley. Revising engineering models: Combining computational discovery with knowledge. In *Proceedings of the Thirteenth European Conference on Machine Learning*, 2002.

[4] S. Bay, J. Shrager, A. Pohorille, and P. Langley. Revising regulatory networks: From expression data to linear causal models. *Journal of Biomedical Informatics*, 35:289–297, 2003.

[5] D. Bobrow, B. Falkenhainer, A. Farquhar, R. Fikes, K. Forbus, T. Gruber, Y. Iwasaki, and B. Kuipers. A compositional modeling language. In *Proceedings of The Tenth International Workshop on Qualitative Reasoning*, 1996.

[6] Scott Cohen. Cvode, a stiff/nonstiff ode solver in c, http://citeseer.nj.nec.com/1230.html.

[7] de Kleer. J., A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992.

[8] R. Dearden and D. Clancy. Particle filters for real-time fault detection in planetary rovers. In *Proceedings of the 13th International Workshop on Principles of Diagnosis (DX-2002).* http://www.dbai.tuwien.ac.at/user/dx2002/, 2002.

[9] R.J. Doyle, S.M. Sellers, and D.J. Atkinson. A focused, context-sensitive approach to monitoring. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1989.

[10] D. Dvorak. Monitoring and diagnosis of continuous dynamic systems using semiquantitative simulation. *PhD thesis, University of Texas, Austin, TX*, 1992.

[11] K. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[12] M. Hofbaur and B. Williams. Mode estimation of probabilistic hybrid systems. In *International Conference on Hybrid Systems: Computation and Control*, 2002.

[13] C. Jones, G. Bond, and P. Lawrence. Consistency-based fault isolation for uncertain systems with applications to quantitative dynamic models. In *Proceedings of the 13th International Workshop on Principles of Diagnosis (DX-2002).* http://www.dbai.tuwien.ac.at/user/dx2002/, 2002.

[14] P. Langley, J. Sanchez, L. Todorovski, and S. Dzeroski. Inducing process models from continuous data. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 347–354. Morgan Kaufmann, 2002.

[15] J. Larrson. Goalart, http://www.goalart.com.

[16] J. Larsson. Diagnosis based on explicit means-end models. *Artificial Intelligence*, 80(1):29–83, 1996.

[17] D. Lawesson, U. Nilsson, and I. Klein. Fault isolation using process algebra models. In *Proceedings of the 13th International Workshop on Principles of Diagnosis (DX-2002).* http://www.dbai.tuwien.ac.at/user/dx2002/, 2002.

[18] J. Malin. Using hybrid modeling for testing intelligent software for lunar-mars closed life support. *JOM-e*, 51(9), http://www.tms.org/pubs/journals/JOM/9909/Malin/Malin-9909.html, 1999.

[19] J. Malin, D. Ryan, and L. Fleming. Configintegrated engineering of systems and their operation. In *Proceedings of the Fourth National Technology Transfer Conference (NASA Conference Publication CP-3249)*, pages 97 – 104, 1993.

[20] P. Mosterman. Hybrsim - a modelling and simulation environment for hybrid bond graphs. *Proceedings of the I MECH E Part I Journal of Systems & Control Engineering*, 216:35–46, 2002.

[21] P. Mosterman and G. Biswas. Monitoring, prediction, and fault isolation in dynamic physical systems. In *Proceedings of the American Association for Artificial Intelligence AAAI*, 1997.

[22] S. Narasimhan, B. Biswas, G. Karsai, and T. Szemethy. Hybrid modeling and diagnosis in the real world: A case study. In *Proceedings of the 13th International Workshop on Principles of Diagnosis (DX-2002).* http://www.dbai.tuwien.ac.at/user/dx2002/, 2002.

[23] S. Narasimhan, G. Biswas, and G. Karsai. An integrated approach to diagnosis of complex hybrid systems. In *15th Annual International Symposium on AeroSense (Component and Systems Diagnostics, Prognosis, and Health Management)*, pages 275–286, 2001.

[24] B. Rinner and B. Kuipers. Monitoring piecewise continuous behaviors by refining semi-quantitative trackers. In *Proceedings of the Sixteenth International Joint Conference on Aritifical Intelligence*, 1999.

[25] G. Saito, P. Langley, S. Bay, and K. Arrigo. Discovering ecosystem models from time-series data. In *Proceedings of the Sixth International Conference on Discovery Science*, in press.

[26] D. Shapiro. Sniffer: A system that understands bugs. *Master's Thesis: Massachusetts Institute of Technology, Artificial Intelligence Laboratory. AI Memo 638*, 1981.

[27] B. Williams, M. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. Sullivan. Model-based programming of fault-aware systems. *AI-Magazine*, Fall, 2003.

[28] B. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the American Association for Artificial Intelligence*, pages 971–978, 1996.