Presented at: European Conference on Industrial Applications of Knowledge-Based Diagnosis, Segrate (Milan), Italy, October 17-18, 1991.

USING THE G2 DIAGNOSTIC ASSISTANT FOR REAL-TIME FAULT DIAGNOSIS

F. Eric Finch, Gregory M. Stanley* and Steven P. Fraleigh
Gensym Corporation, 125 CambridgePark Dr., Cambridge, MA, 02140 (USA)

Keywords: Expert systems, Fault diagnosis, Real time computer systems, Supervisory control, Knowledge engineering, Artificial intelligence, Automation

Abstract

This paper presents an overview of real-time fault diagnosis and describes essential features of a software environment for developing real-time fault diagnosis systems. The G2 Diagnostic Assistant is an environment for real-time fault diagnosis, created using the G2 Real-Time Expert System. The principle component of the Diagnostic Assistant is a graphical language for representing diagnostic knowledge called GDL. GDL contains tools for common diagnostic problems such as malfunction detection, alarm filtering, intelligent information display, and fault recovery. Some of the features that make GDL well suited for diagnosis of dynamic systems are described and applications areas are discussed.

Real-Time Fault Diagnosis

The objective of fault diagnosis is to pinpoint and correct problems that occur in dynamic systems. The objective of real-time fault diagnosis is to perform these tasks within a time frame that allows continued, safe operation of the system. Economic incentives for real-time fault diagnosis include product quality, equipment protection, and environmental protection. The time-frame for real-time diagnosis (TFD) can vary dramatically depending on the system being diagnosed. For many electronic or power systems, TFD may be one second or less. For some chemical or biological systems, TFD may be several minutes or hours.

Automated computer systems have many advantages for real-time fault diagnosis [1]. Automated systems are capable of continuously monitoring more variables with greater accuracy and faster response time than human beings. To fully realize the advantages, however, an automated diagnosis system must be able to:

1. access real-time data,
2. intelligently interpret data,
3. communicate with human decisionmakers, and
4. initiate corrective actions.

Items 1 and 4 can usually be accomplished by integrating the diagnosis system with existing automation systems. Data is usually available from a real-time database that serves as a repository for data collected from remote sensors. Corrective actions can usually be initiated by accessing the same supervisory control systems used by human operators. This integration can be accomplished by existing network technologies. Items 2 and 3 are the key features that need to be addressed by advanced software development environments.

Specification of a Real-Time Diagnosis Environment

Before a diagnosis system can intelligently interpret data, knowledge must be added to the system by a human domain expert. This knowledge can include models of system behavior and interactions (models), and human experience and interpretations (heuristics). Collectively, this knowledge is referred to as a *knowledge-base*. Knowledge-based diagnosis systems can achieve greater sophistication, sensitivity, and flexibility than can be achieved with hardware protection, safety interlocks, or simple alarming systems alone [1 - 4].

Probably the most difficult task in developing an automated diagnosis system is creation of the knowledge-base; therefore, a software environment for real-time diagnosis must have a user interface tailored for the knowledge-base developer. The developer's interface should support rapid, incremental development, so the developer can quickly build, test, and modify diagnosis strategies. The developer's interface should represent knowledge in a way that is logical, understandable, and closely matches the domain expert's mental models, so that other domain experts can immediately comprehend the knowledge-base. Lastly, the developer's interface should be sufficiently intuitive and robust to allow the domain expert to construct the knowledge-base without the aid of a knowledge engineer. Knowledge engineers are experts in a particular software environment who can translate the domain expert's knowledge into a form the software can utilize. Knowledge acquisition and translation are inherent bottlenecks in the construction of a knowledge-base.

One approach to interface design that eliminates the need for a knowledge engineer is to provide as part of the environment a set of prebuilt tools that the domain expert can use without programming. The tool set should be capable of performing the majority of common functions demanded by the domain expert. A complete environment for real-time fault diagnosis should be capable of performing the following functions:

1. filtering and statistically analyzing noisy data,
2. detecting fault symptoms,
3. identifying root causes,
4. generating and managing alarms,
5. planning and executing tests,
6. giving advice,
7. explaining conclusions,
8. recognizing recurring problems, and
9. determining appropriate corrective actions.

To communicate with human decisionmakers, the software environment should have a second user interface tailored for end-users. Unlike the developer's interface, targeted for knowledge-base construction, the end-user's interface should concentrate on display of information. The end-user's interface should provide facilities to quickly browse the knowledge-base so that the basis for diagnostic conclusions can be understood. If the conclusions and advice of the diagnosis system are to be believed, it is important that the system not operate as an inscrutable "black box". The end-user's interface should include features that highlight critical information while de-emphasizing non-critical information. Information filtering and prioritization are crucial to avoid overloading and distracting the human decisionmaker. Information should be presented primarily in summary form with details available upon request. Information display density should be kept high to minimize the number of displays that need to be visible at any one time. The system should allow human confirmation or override of conclusions and actions, and the interface should support creation of a log containing all system conclusions, explanations, and advice, and all user comments and inputs.

Knowledge-based Expert Systems for Diagnosis

Knowledge-based expert systems (KBES) satisfy many of the requirements of a real-time diagnosis environment. Modern KBES combine the features of object-oriented systems and rule-based expert systems to provide many options for knowledge representation. A recent trend has been to incorporate in KBES features that support creation of real-time systems, such as task schedulers for concurrent operations, time stamping and validity intervals for data, history-keeping, and real-time data interfaces [5].

A knowledge-base is comprised of several different types of knowledge, such as

1. facts,
2. associations,
3. conditionals,
4. procedures, and
5. equations.

Object-oriented systems [6,7] define classes of objects to represent facts or behaviors. Each class serves as a template for organizing data by defining the number and type of attributes that distinguish one type of object from another. Associations among facts are maintained by the structure of the object system, typically hierarchical, and by relations between objects.

Rule-based expert system shells [8] provide a convenient mechanism for representing conditional knowledge. Rules are typically of the form

IF <a set of conditions> THEN <a set of conclusions>

An *inference engine* is provided that searches for and executes pertinent rules. Because the rules are separate from the inference engine, this style of knowledge representation in intrinsically declarative. In more advanced shells, rules are structured to resemble natural language, making the knowledge accessible to domain experts who are not familiar with software environment. Almost all shells also provide access to a more conventional procedural language that can be used to represent procedural knowledge or write functions and formulas (equations). Sometimes, the procedural language is merely the underlying language in which the shell is written, such as LISP or C.

Graphical Knowledge Representation

Now that graphics workstations and personal computers with advanced graphic capabilities are widely available, graphic user interfaces (GUI) are common in software environments. KBES are no exception. In a graphics-oriented KBES, objects are represented as graphic icons that can be moved and arranged by the user on a workspace; relationships between objects can be represented as graphic connections; and using a pointing device, a dialog box can be used to view or modify object attributes. In a graphics-oriented KBES, information can be communicated to the users via colors, pictures, and animation.

Graphics-oriented knowledge representation has many advantages over text-based representation. GUI are language independent and can gain broad acceptance internationally. More information can be displayed in a given display area using graphics than can be achieved using text. Perhaps most importantly, graphics may be the most natural form to represent certain types of knowledge. Common forms of graphical knowledge representation are maps, system schematics, program flowcharts, organizational charts, fault trees, decision trees, project management schedules, and so on.

GDL -- An Integrated Graphical Language for Diagnosis

The G2 Diagnostic Assistant[1] is a software environment for real-time fault diagnosis. The Diagnostic Assistant has been developed using the G2[1] Real-Time Expert System, a real-time, graphics-oriented KBES that has been used in a variety of online applications [9 - 11]. The primary developer's interface in the Diagnostic Assistant is the Graphical Diagnostic Language (GDL). GDL allows the domain expert to encode diagnostic knowledge in a series of connected block diagrams. The functional goal of GDL is to provide an environment that includes the basic tools (blocks) for real-time fault diagnosis. The implementational goal of GDL is to provide an environment that follows simple conventions, is easy to use, is rich in visual feedback, and can be freely extended.

GDL Overview

The basic component of GDL is a *block*. Following the object-oriented programming approach, GDL defines a variety of different block *classes*, each of which can have an unlimited number of block *instances*. Depending on how its class is defined, a block can have zero, one, or multiple inputs and outputs (IO). IO can be analog values, discrete values, logical states (TRUE,FALSE,UNKNOWN), or program control signals. Different IO are represented graphically by *stubs* (in G2, handles where a developer can click and drag to create a connection between two blocks). Stubs are color coded to prevent a developer from inadvertently connecting inputs and outputs of mixed type. For example, a developer cannot connect an analog output to a logical input.

Standard blocks are provided for filtering, signal processing, statistical analysis, limit checking, logical inference, evidence combination, and sequential control. The complete language contains over one hundred graphical blocks -- a list of major GDL blocks is provided in Appendix A. The first job for the developer is to create instances of the blocks he needs for diagnosis. This requires decomposing the deductive process into a series of tasks and creating a block for each task. Then, the developer creates an information flow diagram (IFD) by connecting the blocks. These connections specify how each block will get its input values and where it will send its output values. Lastly, the developer must specify configuration values for certain blocks. For example, if a first order filter block has been created, the time constant of the filter must be specified. When the IFD is complete, the developer can ask the Diagnostic Assistant to check the IFD for potential problems such as missing configuration values, cyclic paths, or missing connections.

Figure 1 illustrates an IFD containing a variety of GDL blocks. The standard GDL convention is for information to flow from left to right and from top to bottom. At the far left of the figure is an entry point block. Entry points are specialized blocks that collect and manage incoming data -- they denote the beginning of an IFD. When new data is received by the entry point, it flows through the IFD, being modified by the blocks it encounters. IFDs are fundamentally data-driven, forward chaining programs.

[1] G2 and the G2 Diagnostic Assistant are registered trademarks of Gensym Corporation, Cambridge, MA.

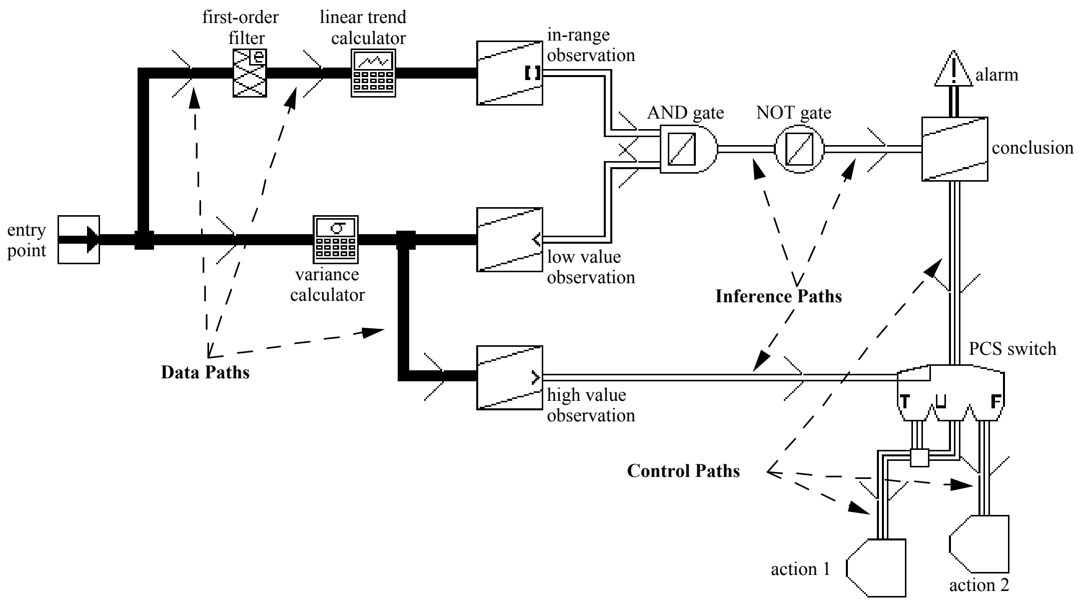


FIGURE 1: Sample IFD

Data Paths, Inference Paths and Control Paths

In figure 1, the entry point is connected to two other GDL blocks, a first-order filter and a variance calculator, via a type of connection called a *data path*. Data paths transfer analog values between blocks. Another data path transfers the output of the first-order filter to the linear trend calculator. All blocks that have analog inputs and outputs are subclasses of the class *data block*. Data blocks typically perform some numerical transform on the data. For example, the output of the linear trend calculator is the rate of change (slope) of a sample of its input values calculated by least-squares regression.

The next layer of blocks are *observations*. Observations accept analog inputs and produce logical outputs. The logical outputs can have values of TRUE, FALSE, or UNKNOWN and are determined by a test defined by the observation class. For example, the in-range observation tests if its input value falls within a specified range of values, and if so, produces a TRUE output. If not, a FALSE output is produced. The low value and high value observations test whether their respective inputs are below or above specified thresholds. An UNKNOWN output is produced if no input value is available, the input value is bad, or there is a high degree of uncertainty associated with the test result.

Logical values are passed between blocks by another type of connection called an *inference path*. Blocks that have logical inputs and outputs are called *inference blocks*. The AND gate and the NOT gate are examples of inference blocks. The AND gate produces a TRUE output only when all its inputs are TRUE. The output of the NOT gate is the inverse of its input (i.e. a TRUE input results in a FALSE output). At the output of the NOT gate is an inference block called a *conclusion*. The conclusion shown in this figure is a final conclusion since it is at the end of an inference path. GDL also supports intermediate conclusions.

Data and inference paths propagate both values and *program control signals* (PCS). A PCS tells a block to execute its evaluation procedure (method) and update its output. When a data or inference block posts a new output, both the new output value and a PCS are sent to every block in the IFD connected to the output of the block via a data or inference path. A third type of connection called a *control path* propagates PCS without associated values. In figure 1, a control path connects the conclusion block to two action blocks via a PCS switch. Whenever the conclusion posts a TRUE output, a PCS is sent to the PCS switch which routes the signal to either action 1 or action 2, depending on the output value of the high value observation. If the high value observation has a TRUE or UNKNOWN output, action 1 will receive the PCS; if the high value observation has a FALSE output, action 2 will receive the PCS. Actions 1 and 2 do not require input values -- they are side-effects. The switch acts as a conditional statement, determining which action will be performed.

Capabilities

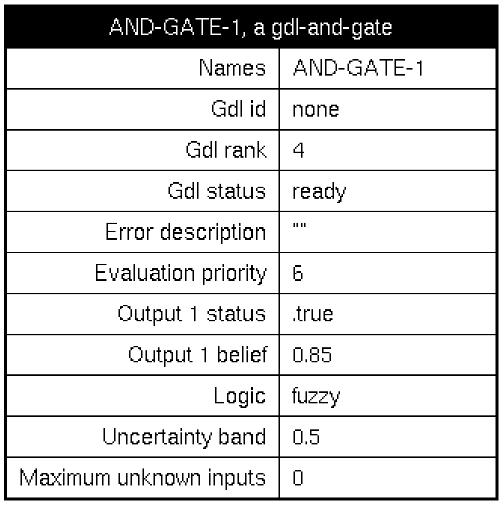
In designing a graphically oriented language, a balance must be developed between the number of blocks in the language and the average complexity of each block. If individual blocks are kept simple, the language as a whole will require more blocks to accomplish a given task set. In GDL, we have tried to keep the number of blocks manageable by giving most blocks at least one configurable optional behavior.

The drawback of optional behaviors is the loss of clarity that occurs when two blocks that look the same can behave in slightly different ways. To minimize this problem, major optional behaviors of GDL blocks are implemented as *capabilities*. Capabilities are separate graphical objects that can be attached to blocks to impart an optional behavior. For example, alarms are implemented in GDL as capabilities. Figure 1, shows an alarm capability object attached to the conclusion block. By configuring the alarm object, the developer can specify the type of alarm, the alarm severity, advice to the user that applies when the alarm is active, and so on.

The advantages of this approach are that the presence of an alarm is visible in the IFD rather than being hidden in the configuration attributes of the block and that the attributes necessary to configure the alarm are separate from the attributes of the conclusion, reducing the size and complexity of the conclusion block. This is beneficial since not every conclusion block will have an associated alarm.

Inference Path Output Filtering

When one or more inputs to an inference block receive new values but the output value of the block remains the same after evaluation, propagation along the inference path is terminated -- no PCS is sent to blocks further down the IFD. This feature is included to increase evaluation efficiency of the IFD. In essence, every inference block acts as a filter capable of stopping IFD propagation. For example, if both the in-range and low value observations in figure 1 have FALSE outputs, then the output of the AND gate will also be FALSE. If later, the output of the low value observation becomes TRUE, the output of the AND gate will remain FALSE because the in-range observation is FALSE. Since the output of the AND gate did not change, no PCS is sent to the NOT gate -- its output does not need to be recomputed. Evaluation of the IFD along the path stops at the AND gate. To implement this feature, each inference block must maintain locally a record of its output value. The most recent logical output, called the *output status*, is stored as an attribute of the block as shown in figure 2.



AND-GATE-1, a gdl-and-gate		
Names	AND-GATE-1	(optional)
Gdl id	none	(optional)
Gdl rank	4	(system)
Gdl status	ready	(system)
Error description	""	(system)
Evaluation priority	6	(user configurable)
Output 1 status	.true	(system)
Output 1 belief	0.85	(system)
Logic	fuzzy	(user configurable)
Uncertainty band	0.5	(user configurable)
Maximum unknown inputs	0	(user configurable)

FIGURE 2: AND Gate Attribute Table

Discrete Logic, Fuzzy Logic and Evidence Combination
A crucial aspect of real-time fault diagnosis is the ability to handle uncertainty. Fault diagnosis always involves uncertainty because faults typically cannot be observed directly and must be deduced from indirect measurements. Allowing UNKNOWN as a logical status is one way in which GDL expresses uncertainty. This is part of an overall scheme that allows GDL to manage uncertainty.

GDL defines a quantitative measure of uncertainty called *belief*. Every block that has an output status must also have an output belief. Depending on how a block is configured, its output status will be computed from its output belief or its output belief will be computed from its output status. Whenever a status value is transferred between blocks along an inference path, the associated belief value is also transferred -- status and belief are a data pair.

Belief can have real values in the interval [0,1]. Belief = 0 is equivalent to the status FALSE. Belief = 1 is equivalent to the status TRUE. Belief = 0.5 is equivalent to the status UNKNOWN. Belief values other than 1 and 0 denote levels of uncertainty. For example, Belief = 0.5 is exactly halfway between TRUE and FALSE and represents complete uncertainty in the output status. Belief = 0.85 indicates a relatively high degree of certainty that the output is TRUE; whereas, belief = 0.15 indicates a relatively high degree of certainty that the output is FALSE. In GDL, belief is used to implement a variety of techniques for managing uncertainty, including fuzzy logic [12,13], evidence combination [14], and probability [15].

An optional behavior of GDL logic gates (e.g. AND gates, OR gates, NOT gates, etc.) is the ability to operate in fuzzy logic mode. Logic gates have a configuration attribute named *logic*, shown in figure 2. When gate logic is set to discrete, the gate output status is computed using its input status values and an output belief value of 0, 0.5,or 1 is generated based on the output status. When gate logic is set to fuzzy, the gate output is computed using its input belief values and the *uncertainty band* of the gate is used to compute the output status. The size of the uncertainty band determines the range of beliefs that produce an UNKNOWN status. For example, in figure 3, the uncertainty band has a value of 0.5 (the default value). If belief is greater than 0.75, the output status is TRUE. If belief is less than 0.25, the output status is FALSE. Otherwise, the output status is UNKNOWN. The uncertainty band can have real values in the interval [0,1] and is an adjustable configuration parameter of gates that compute belief outputs.

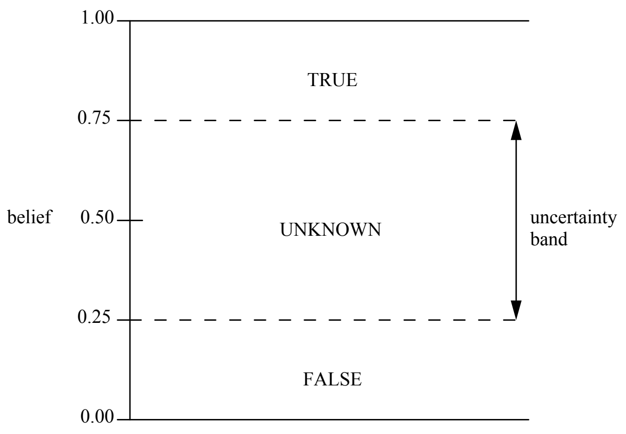


FIGURE 3: Uncertainty Band

In addition to discrete and fuzzy logic gates, GDL defines specialized inference blocks for evidence combination. In GDL, an evidence combination gate performs a non-logical computation on multiple input belief values. For example, the weighted evidence combination gate computes a weighted linear average of input belief values followed by an optional sigmoidal nonlinearity. The weighted combination of belief values provides a useful and flexible alternative to pure voting logic. With the added nonlinearity, it is possible for the output belief to be higher (or lower) than any individual input belief. This is useful when imperfect or redundant measurements are combined. Currently, no GDL blocks have been defined that manage uncertainty using probability theory, however, this is an active area of investigation.

Suppression of Transient Inference Disturbances

For many systems, the real-time data used for fault diagnosis will be imperfect. Even when sensors are functioning correctly, data can contain measurement errors. Measurement error typically contains a random component (noise) and a non-random component (bias). When a belief value is near a status transition threshold, measurement noise can produce rapid status changes. In figure 3, for example, if a series of noisy measurements produced a sequence of belief values (0.748, 0.752, 0.755, 0.747, 0.751) the resulting output statuses would be (UNKNOWN, TRUE, TRUE, UNKNOWN, TRUE). The consequence is that the output status exhibits undue sensitivity to small belief variations. If there are inference blocks configured for discrete logic connected at the output of a block exhibiting these rapid status changes, they too may exhibit rapid status changes. In rule-based expert systems, this phenomenon has been called *rule chattering* [16]. Rule chattering can result in an unstable diagnosis, characterized by rapid changes in conclusions and advice. An unstable diagnosis can quickly degrade user confidence in the diagnosis system [1].

To avoid chattering, GDL includes features designed to suppress status changes caused by small belief variations. The first technique is status *hysteresis*. When hysteresis is applied, the status will not change until the belief reaches the threshold for the inverse status. Figure 4 illustrates this feature. At t1, as belief crosses the uncertainty band threshold, the status changes from UNKNOWN to TRUE. At t2, the status would change back to UNKNOWN if hysteresis were not applied. However, with hysteresis, output status remains TRUE until t3. Hysteresis can be configured to apply unidirectionally to either the TRUE or FALSE status regions or bidirectionally.

The second technique is status *hold*. A status hold defines a minimum time period between changes in output status. This feature is shown in figure 5. At t1, data is received that results in a TRUE status. At t2, new data is received that would result in an UNKNOWN status (without hysteresis). However, if a status hold with hold period > $\Delta t1$ is placed on the TRUE region, then the status remains TRUE until the hold period expires. At the end of the hold period, the status is recomputed based on the last input value. If Δt1 < hold period < Δt2, then when the hold period expires the status will change to UNKNOWN. The start time of the hold period is updated whenever data is received confirming the current status. For example, if the status hold period has been set greater than Δt2, then the data points received at t2 and t3 are ignored and at t4 the status is TRUE. The new data received at t4 confirms the TRUE status and resets the hold period clock (i.e. a new hold period is started). This makes the status hold feature robust when confronted by oscillatory responses. In this example, if hold period > max (Δt2,Δt3),then the output status remains TRUE throughout the episode.

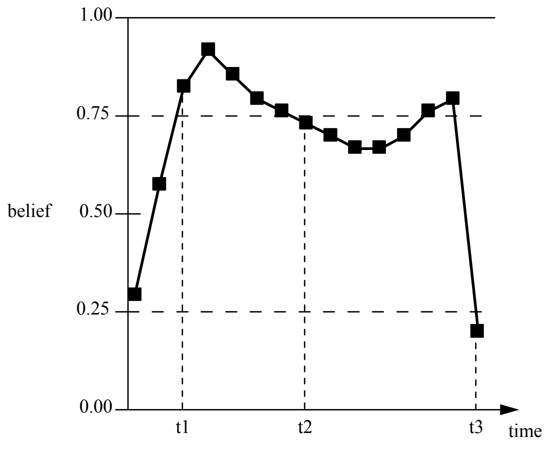


FIGURE 4: Status Hysteresis

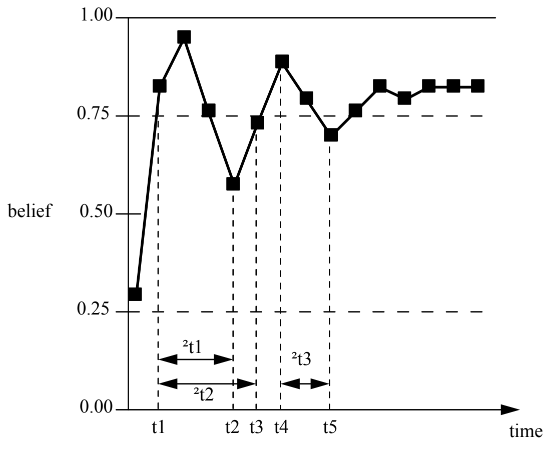


FIGURE 5: Status Hold

The response curves shown in figures 4 and 5 are typical of dynamic systems. The response curve shown in figure 4 could represent drift over a relatively long time period. The response curve shown in figure 5 typically represents short transients after a sudden change. Hysteresis can suppress either type of transient. Status hold is directed toward short duration transients. Specifying a long status hold period is not recommended since it might result in the diagnosis system ignoring significant status changes.

Temporal and Interval Logic

Temporal blocks are used to reason about the timing and sequence of status change events. A basic set of event analysis gates are supplied to compare the sequence of activation events (an input status changing to TRUE) and deactivation events (an input status changing to FALSE). Event analysis gates output TRUE if the temporal spacing of the input activation or deactivation patterns meet specified criteria. GDL provides event analysis blocks to compare:

1. the activation times of the two inputs,
2. the deactivation times of the two inputs, and
3. the deactivation time of one input versus the activation time of a second input.

The developer specifies a discrete or fuzzy time interval in each event analysis gate. The time interval is used to compare the time stamps of the most recent activation or deactivation events. If the difference between the time stamps falls within the specified time interval, then the output status of the block is TRUE. By specifying various time intervals in these blocks, the important temporal knowledge relations [17] can be constructed (e.g. A before B, A during B, etc.). Quantitative time constraints can also be configured (e.g. A *at least 5 minutes* before B). Event analysis is useful for diagnosing causal systems with time delays, because the order of occurrence of events can help identify the root cause. The ability to specify fuzzy time intervals and time constraints is necessary to handle the uncertainty often encountered with event order in dynamic systems [1].

In addition to event analysis gates, GDL provides other temporal blocks that retain maximum, minimum, and average belief values over specified time intervals. When these blocks provide input to a standard logic gate, the output of the logic gate becomes interval-based. For example, the logical output for a standard AND gate can become, semantically, 'Both A and B occurred *during the last 30 seconds*', so that the gate outputs TRUE even if A or B are currently FALSE. Unlike event analysis gates, there is no information about the temporal ordering of A and B in these inferences. This feature is termed *interval logic*.

IFD Encapsulation, Generic IFDs and IFD Compilation

Encapsulation is a technique that hides the details of a complex IFD and makes generic portions of an IFD reusable. IFDs are constructed on G2 *workspaces*. A special class of block called an *encapsulation block* can have an attached subworkspace that holds an encapsulated IFD. When an encapsulation block is evaluated, the encapsulated IFD is used to generate the output values. Figure 6 illustrates this concept. Multiple levels of encapsulation are supported. That is, an encapsulated IFD can itself contain encapsulation blocks. After an IFD has been encapsulated, its workspace can be hidden and the developer can move, connect, and clone (duplicate) the encapsulation block like a normal block.

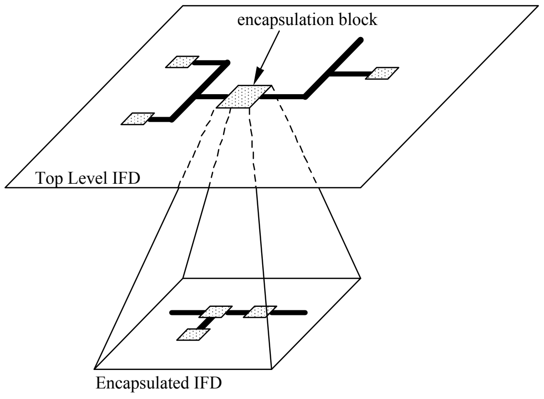


FIGURE 6: IFD Encapsulation

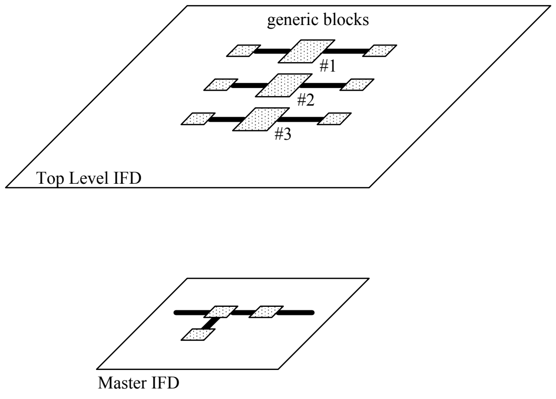


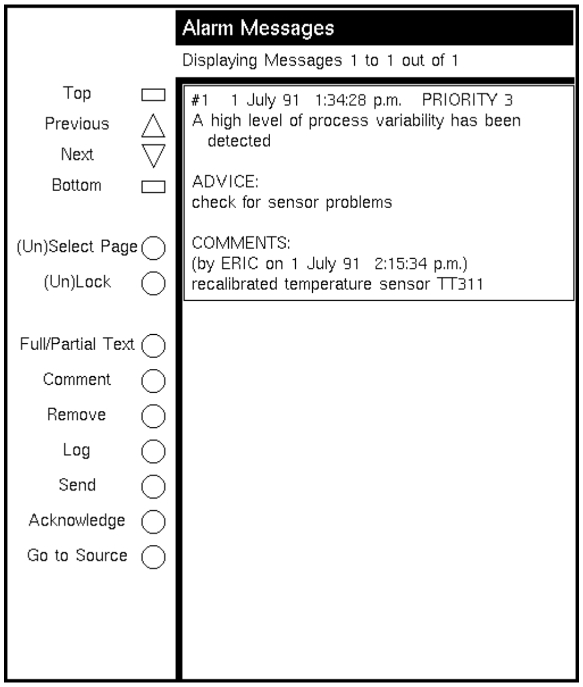
FIGURE 7: Generic IFD

Although encapsulation blocks are useful in creating hierarchical IFDs, they can be difficult to maintain in large applications, particularly if portions of the diagnostic logic have been duplicated many times. Suppose, for example, that an encapsulation block has been created and cloned many times so that there are numerous copies of the encapsulated IFD. If the developer decides to make a change in the encapsulated IFD, each copy of the encapsulated IFD must be modified individually. A better solution is to create a generic IFD as shown in figure 7. Here, *generic blocks* are created that use a separate master IFD for evaluation. Because the master IFD is not attached to any given block, it can be used by any number of generic blocks, and there is only a single copy of the master IFD to be modified.

IFD compilation involves eliminating the encapsulated or master IFD altogether in favor of a single compiled procedure and a single state vector to hold dynamic states such as statuses and beliefs. Compiled IFDs are inherently generic. The advantage of compilation is that it eliminates graphical interpretation of the IFD at runtime. The disadvantage of compilation is that the knowledge originally contained in the IFD is no longer in a form that can be easily browsed by the user.

End-User Interface
The Diagnostic Assistant provides a variety of displays to communicate with the end-user, and a menu system to allow the end-user to quickly move between displays. Major displays are summarized below:

Message Queues
Message queues are displays that manage text messages generated by the system. Queues support scrolling, message selection, comment entry, acknowledgement, and logging. Queue entries can be sorted chronologically or by priority. Several predefined message queues are provided for alarm messages, configuration error messages, explanations, and general messages. The developer can create additional queues if needed. Figure 8 shows a sample alarm message.



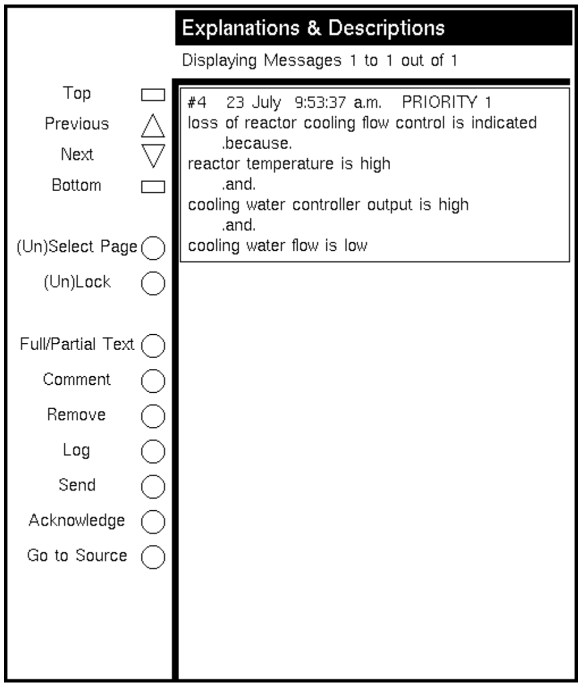
Displaying Messages 1 to 1 out of 1

#1 1 July 91 1:34:28 p.m. PRIORITY 3
A high level of process variability has been detected

ADVICE:
check for sensor problems

COMMENTS:
(by ERIC on 1 July 91 2:15:34 p.m.)
recalibrated temperature sensor TT311

FIGURE 8: Alarm Message Queue



Displaying Messages 1 to 1 out of 1

#4 23 July 9:53:37 a.m. PRIORITY 1
loss of reactor cooling flow control is indicated
.because.
reactor temperature is high
.and.
cooling water controller output is high
.and.
cooling water flow is low

FIGURE 9: Explanation Queue

Alarm Panels
Alarm panels allow the developer to simulate in software an alarm annunciator panel. Alarm panels use colors to indicate the severity of the underlying alarm. The colors used to represent severity levels can be configured by the developer. Alarm panels can be constructed hierarchically with higher level panels displaying a summary of lower level panels. Panels allow the user to override an alarm, acknowledge an alarm, ask for a text explanation of an alarm, or show the IFD from which the alarm originated. Text explanations are displayed on the explanation message queue. A sample alarm explanation is shown in figure 9. Alarm panels may be temporarily inhibited by the user.

Browsing and Highlighting
Facilities have been provided to allow users to browse IFDs. Using the IFD itself as a graphical explanation of the diagnostic reasoning serves as an alternative to the text explanation shown in figure 9. The highlight feature allows the user to highlight all IFDs that make use of an inference block's output. This feature facilitates developers organizing IFDs on multiple workspaces.

User Entry
User entry is accomplished through dialog boxes that are created on demand or as a consequence of IFD evaluation. Dialogs are provided to permit the user to override the output status of an observation or conclusion, and to enter status information in response to system queries. Some dialogs can be configured to timeout and perform a default action if the user has not responded in a specified time interval.

Applications

The tools contained in the Diagnostic Assistant support a variety of techniques for

1. sensor validation,
2. intelligent alarm filtering, and
3. active testing.

Before the result of a diagnostic analysis can be believed, the data on which the analysis rests must be believed. This is the role of sensor validation -- verifying that the fault lies in the system and not the sensor. Intelligent alarm filtering is necessary to focus the user's attention on the source of a malfunction and not its consequences. Active testing is required because, to truly identify the root cause of a malfunction, it is frequently necessary to test a fault hypothesis by introducing a controlled disturbance to the system and determining whether the system responds as predicted.

Real-Time Quality Management
In a broader sense, the Diagnostic Assistant supports *Real-Time Quality Management* (RTQM). RTQM is the integration of fault diagnosis techniques and statistical process control (SPC). Standard SPC tests [18] can be sensitive detectors of problems based on individual measurements, but they contain no knowledge of the interaction between measured signals and other system variables. Other than ranking plausible faults based on the statistical distribution of occurrence (Pareto Analysis), SPC offers the operator no guidance as to the root cause of problems or how to correct problems. This has long been a limitation of SPC. Traditional SPC techniques cannot capture process knowledge and reason with it. However, some new SPC packages utilize time series and correlation analysis to help a developer create expert system rules that can be used for root cause analysis once SPC violations are detected [19].

The Diagnostic Assistant integrates these various features and requirements in a single environment. The graphical language blocks of GDL include several which implement standard SPC tests such as X Bar and CUSUM, and others which can detect non-parametric patterns of failure, such as N out of M successive values above a threshold. The output of these blocks is analyzed in an IFD to provide intelligent alarm management and to trigger direct control actions to keep product quality on-spec. GDL contains a variety of pre-defined statistical and time-series blocks that can be used to extract diagnostic knowledge from real-time or historical data.

Developers can configure SPC tests to provide belief inputs to high level diagnostic reasoning based on fuzzy logic or evidence combination. The alarms are typically generated by the high-level conclusions rather than the output of the SPC tests. Often this means that the control limits for individual SPC violations can be quite strict (to increase sensitivity), but the occurrence of false or nuisance alarms is still held to a minimum. This contrasts with traditional SPC alarms which occur only after faults have propagated and caused a series of off-spec products. For example, a standard SPC test might require 6 successive values above the mean to produce an off-spec alarm; whereas in an IFD, a run of 3 successive values above the mean might combine with other evidence to produce a more specific or root-cause alarm. By building process knowledge into the system, a fault can be detected and corrected well before a traditional SPC alarm would be generated.

Distributed Applications

The performance (response time) of a diagnostic application is a complex function of the

1. number of sensors being concurrently monitored (-),
2. amount of signal processing performed on each sensor value (-),
3. complexity of the diagnostic logic (-),
4. degree of user interaction (-),
5. software efficiency (+), and
6. computer resources (+).

Those items marked with (-) are inversely related to performance. For example, the larger the number of monitored sensors, the worse the performance of the application, all other factors being equal. Conversely, those items marked with (+) have a positive effect on performance. Items 1 - 4 are largely determined by the system being diagnosed. Item 5 can be improved somewhat by IFD compilation. Only item 6 can be freely adjusted to achieve the TFD necessary for real-time performance.

For plant-wide applications, if a single computer is not sufficient for real-time performance, a flexible architecture consisting of two or more networked computer workstations is ideal. Each workstation is responsible for a portion of the overall application, and the exact number of workstations is determined by the performance requirements of the application. This type of distributed architecture is expandable, so that as the knowledge-base grows, more workstations can be added to maintain the same performance. A distributed architecture is also more robust. If one workstation suffers a power or hardware failure, other workstations continue to function.

For a diagnostic application to be distributed among several workstations, the knowledge-base must be distributed as well. GDL supports distributed applications by allowing IFDs to be divided between two or more workstations. Communication between distributed IFDs is handled transparently by built-in G2 network protocols.

Conclusions

By combining aspects of expert systems and object-oriented programming into a general graphical language for diagnosis, the Diagnostic Assistant satisfies the requirements of a real-time fault diagnosis environment. The Diagnostic Assistant will be marketed as an add-on product for the G2 Real-Time Expert System. Currently, the Diagnostic Assistant is being tested prior to release by users in the aerospace, chemical, food, manufacturing, and power industries.

References

1. Finch, F.E. (1989). Automated Fault Diagnosis of Chemical Process Plants using Model-Based Reasoning. Sc.D. Thesis, MIT.
2. Stanley, G.M. (1991). Experiences using Knowledge-Based Reasoning in Online Control Systems. *IFAC Symp. on Computer Aided Design in Control Systems*. Swansea, UK, 1991.
3. Rowan, D.A. (1988). AI Enhances On-Line Fault Diagnosis. *InTech*, 35 (5), 52.
4. Andow, P. (1991). A Real-Time System for Fault Diagnosis. *Expert Systems and Optimisation in Process Control*. Middlesex, UK, 1991.

5. Lalka. C.J. and R. Weber (1991). Real-Time versus Static Shells for Real-Time Expert Systems. AIChE Spring National Meeting, Houston, Texas, 1991.
6. Stefik, M. and D.G. Bobrow (1986). Object-Oriented Programming: Themes and Variations. *The AI Magazine*, 6 (4), 40.
7. Cox, B.J. (1986). Object Oriented Programming. Addison-Wesley, Reading, MA.
8. Buchanan, B.G. and E.H. Shortliffe (1984). Rule-Based Expert Systems. Addison-Wesley, Reading, MA.
9. Weber, R. (1991). Real-Time Diagnostic Toolkit. *Proc. Gensym Users Society Annual Meeting*. Houston, Texas, 1991.
10. Mertz, G.E. (1990). Application of a Real-Time Expert System to a Monsanto Process Unit. *Proc. Chemical Manufacturer's Association Process Control Conference*. Miami, Florida, 1990.
11. Filippini, F., S. Urbinati and M.Solimano (1990). An Expert System for the Prevention of Malfunctions and Technological Optimization in a Beet Sugar Plant. *Proc. Gensym European Users Society Meeting*. Munich, Germany, 1990.
12. Schwartz, T.J. (1991). Fuzzy Tools for Expert Systems. *AI Expert*. 6 (2), 34.
13. Maiers, J. and Y.S. Sherif (1985). Applications of Fuzzy Set Theory. *IEEE Trans. on Sys., Man, and Cyber.,* SMC-15 (1), 175.
14. Garvey, T.D., J.D. Lowrance and M.A. Fischler (1981). An Inference Technique for Integrating Knowledge from Disparate Sources. *Proc IJCAI-81*. Vancouver, B.C., 319.
15. Pearl, J. (1988). Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Mateo, CA.
16. Stanley, G.M., F.E. Finch and S.P. Fraleigh (1991). An Object-Oriented Graphical Language and Environment for Real-Time Diagnosis. *Proc. European Symposium on Computer Applications in Chemical Engineering (COPE-91)*. Barcelona, Spain, 1991.
17. Allen, J.F. (1983). Maintaining Knowledge about Temporal Intervals. *Comm. ACM*, 26 (1), 832-843.
18. MacGregor, J.F. (1990). Statistical Process Control and Interfaces with Process Control. *The Second Shell Process Control Workshop* (D.M. Prett, C.E. Garcia and B.L. Ramakar, eds). Butterworths, Boston, MA.
19. Frerichs, D.K. (1991). SPC + Expert Systems = Intelligent Alarms. *InTech*, 38 (6), 1991.

*Contact Greg Stanley at
http://gregstanleyandassociates.com/contactinfo/contactinfo.htm

Appendix A: Major GDL Block Classes

Entry Point

Filters
- Changeband
- Outlier
- Linear Exponential
- Non-linear Exponential

Calculation Blocks
- Sum
- Product
- Inverse
- Minimum
- Maximum
- Numerical Counter
- Fixed Bias
- Fixed Gain
- Shift Register
- Time Stamp
- Sample and Hold
- Linear Predictor

Time Series Blocks
- Integrator
- Linear Regression
- Quadratic Regression
- Cubic Regression
- Moving Average
- Moving Range

Statistical
- Median
- Average
- Variance
- Covariance
- Standard Moment
- Center of Gravity
- RMS Value
- Process Capability Index
- CUSUM Test
- EWMA Test
- Run Test
- Sign Test

Observations
- Equality
- Threshold Violation
- Deviation
- In-Range
- Out-of-Range
- Pattern
- Zero-Crossing

Conclusion

External Interfaces
- Assertion
- Network Condition
- Belief Transmitter

Logic Gates
- AND
- OR
- NOT
- Exclusive OR (EOR)
- Equivalence
- Voting Logic

Inference Gates
- Unknown
- Persistence
- Event Counter
- Event Timer
- Inhibit
- Logic Switch
- Sequence Gate

Evidence Combination Gates
- Weighted Evidence Combiner
- Belief Bias
- Belief Range
- Fuzzy Implication
- Fuzzy Set Combiner

Temporal Gates
- Average Belief
- Maximum Belief
- Minimum Belief
- Before
- After
- During

Actions
- Send Message
- Lock/Unlock
- Show Display
- Highlight
- Override
- Reset
- PCS Switch
- Activation Counter
- Activation Timer
- Inhibit
- User Query
- User Defined Action