Presented at: Proceedings of the International Federation of Automatic Control (IFAC) Symposium on On-line Fault Detection and Supervision in the Chemical Process Industries, Newark, DE, April 22-24, 1992.

INTEGRATING DATAFLOW AND SEQUENTIAL CONTROL IN A GRAPHICAL DIAGNOSTIC LANGUAGE

Steven P. Fraleigh, F. Eric Finch, and Gregory M. Stanley*

Gensym Corporation, 125 CambridgePark Dr., Cambridge, MA, 02140 (USA)

Abstract. An object-oriented Graphical Diagnostic Language (GDL) has been designed and implemented. GDL merges dataflow and sequential control specification into a single environment for graphical development and run-time deployment of expert diagnostic applications. The language components of GDL are graphics objects (blocks) used to construct fault diagrams and action sequences that are directly executed in real time. GDL components include blocks to detect faults, classify root causes, initiate corrective actions, recognize recurring problems, execute operations tests and procedures, and manage alarm displays and messages. A Gensym product, the G2 Diagnostic Assistant™, is built upon GDL.

This paper describes the object-oriented framework underlying our implementation of GDL in Gensym's G2 real-time development and deployment system. The object-oriented model unifies dataflow and control flow, so that both aspects are represented in a single graph showing blocks and their connections. As a direct result of this integration, GDL supports simple configuration and deployment of generic diagnostic solutions, in which a process expert configures reusable diagnostic modules and libraries that can be distributed to non-specialists for on-line implementation. Advanced features, such as automatic on-line explanation of diagnostic diagrams, are also made possible by this approach.

INTRODUCTION

Motivation for on-line, knowledge-based diagnosis is growing because of concerns about product and environmental quality, but experience has shown that special features are needed to address the requirements of real-time diagnostic systems (Rowan, 1988). Although basic diagnostic procedures perform well, real-world issues such as sensor drift can lead to problems with nuisance alarms and loss of operator confidence in the system, unless features such as delayed alarming are installed. On-line use has also revealed the need for lag and dead-time considerations, along with simple high/low limit and rate detection.

Implementing these real-time features can be difficult in traditional knowledge systems, particularly when their need only arises after the application has been installed and is operating on-line. We have developed a graphical environment that addresses these issues by providing rapid, interactive configuration of diagnostic applications. This product, the G2 Diagnostic Assistant™, has built-in facilities such as delay objects, hysteresis, confidence integrators, temporal logic, and other tools to aid in the design of real-time diagnostic systems (Finch, Stanley, and Fraleigh, 1991). These features are configured using a tool called Graphical Diagnostic Language (GDL) that allows a domain expert to encode diagnostic knowledge in a series of connected block diagrams.

Graphical Languages

Previous work. Existing graphical languages for engineering applications generally emphasize either sequential program control or data flow. The graphical objects in a sequential control language represent procedures. Connections between the objects represent sequential or concurrent ordering of procedure execution. Data flow, and the existence of variables, are not shown as part of the graphical language. An advantage of this representation is that the control of program execution steps is clear, and can be easily

indicated at run-time. This approach works well for supervisory-level sequence control, discrete event simulation, and representation of sequential reasoning procedures (Arzen, 1991).

Traditional data flow systems use objects to represent transformations of input data to output data, with the connections representing information flow. The step-by-step execution of program steps is not shown as part of the language. The advantage of this representation is that the flow of data and their transformed values are clearly visible, independent of the underlying, hidden program control mechanism. This approach works well for continuous system simulators and signal processing applications (Santori, 1990).

Graphical Languages in G2. In addition to standard components such as objects and rules, G2 supports a rich graphics environment which enables reasoning about connectivity between objects. Common forms of graphical knowledge representation in G2 include maps, networks, process schematics, program flowcharts, organizational charts, fault trees, decision trees, and project management schedules. Consequently, many developers use G2's standard features to build application-specific graphical languages for real-time knowledge processing (Stanley, Finch, and Fraleigh, 1991; Arzen, 1991; Nillson, 1991; Sarraut, 1991, Weber and Lalka, 1991).

GDL Strategy. Unlike hierarchical diagnosis schemes, the fundamental strategy in GDL is to forward chain from process data (propagating only on significant changes), derive conclusions from the process, and use these conclusions to trigger action sequences as appropriate. Integrating these facilities in a single environment requires careful integration of dataflow and program control mechanisms. Functions for filtering, detection of symptoms and trends, and combination of evidence or pattern recognition are naturally accomplished in a graphical language emphasizing data flow. However, sequential actions are needed to correct problems and to sequence through manual or automatic operating procedures. Also, sequential actions as part of "active testing" may be required. For instance, to discriminate between several faults, it may be necessary to perform a test on the process, wait, determine the results, and plan further tests. When control actions must be represented, a sequential control-oriented language is best. GDL integrates dataflow and sequential control during real-time execution.

OVERVIEW OF GDL

Blocks

GDL provides blocks that are connected graphically to create information flow diagrams (IFD's). Real-time process data enters a diagram and initiates forward chaining of information from block to block along the graphical connections. The blocks have evaluation methods that execute in sequence during forward chaining; arguments for the evaluation methods are passed from block to block. Important real-time techniques are utilized during method dispatch, including task prioritization, asynchronous concurrent operations, and real-time task scheduling. These techniques are automatically provided by the underlying G2 development system.

To configure a diagnostic application in GDL, a developer selects blocks from system palettes and places them on G2 workspaces. Depending on how its class is defined, a block can have zero, one, or multiple inputs and outputs.

GDL provides data blocks, inference blocks, and action blocks. Data blocks provide filtering, conditioning, signal processing and statistical analysis. Inference blocks form the 'expert analysis' portion of an application and provide limit checking, event filtering, fuzzy and discrete logic, evidence combination, and temporal sequence recognition. Action blocks provide sequential functions, e.g. in response to a detected condition. Finch, Stanley and Fraleigh (1991) describe functional details of GDL blocks in further detail.

Connections

A typical IFD is partitioned for data analysis, logical analysis, and action sequencing. The partitions are implicitly specified by connection types - data, inference, and action blocks use different connections. The

connections define which data types are passed and which methods are invoked during forward chaining. To aid in the overall visual understanding of an application, connection ports on the objects are arranged to encourage users to configure data flow from left to right, and sequential function steps to proceed from top to bottom. A fourth connection type used by certain action blocks can specify the recipient of an auxiliary signal, such as a status lock or reset. The final connection type is used to link blocks with capabilities. These are separate graphical objects that impart optional behavior to a block, such as the ability to set an alarm.

An Example IFD

Data blocks. Figure 1 illustrates a simple IFD. At the far left of the figure is an entry point block that collects and manages incoming data. In Fig. 1, the entry point is connected to two other GDL blocks - a first-order filter and a variance calculator - via a *data path* connection. Data paths transfer analog values between blocks.

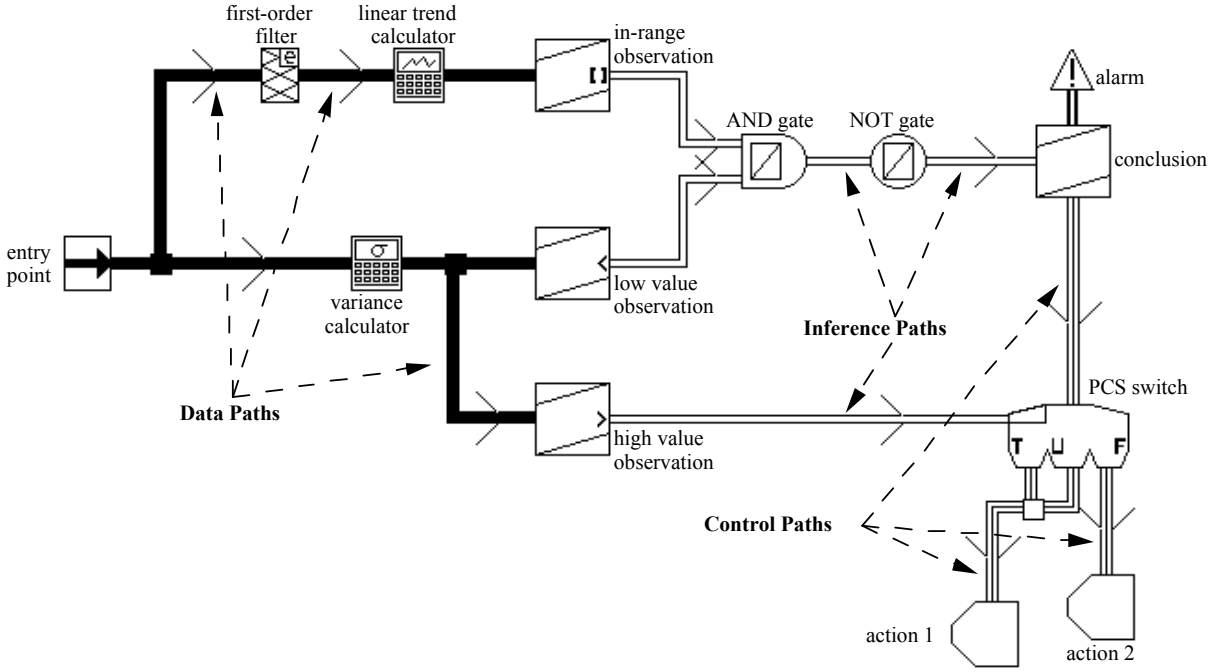

Figure 1. Sample IFD.

Inference blocks. The next layer of blocks are *observations*. Observations are inference blocks that accept analog inputs and produce logical outputs. The logical outputs can have status of TRUE, FALSE, or UNKNOWN and are determined by a test defined in observation's evaluation method. 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.

Logical values are passed between observations and other inference blocks along *inference paths*. Logical inferences are configured with *gates*. In Fig. 1 the AND gate and the NOT gate are also inference blocks. The AND gate produces a TRUE output only when all its inputs are TRUE. The output of the NOT gate is the logical inverse of its input. GDL defines gate classes for each of the standard logic functions. For combining information from multiple sources, there are also gates for voting logic and other forms of evidence combination. Thus the primary methodology is a logic network approach, but with important extensions for uncertainty management and temporal reasoning.

At the output of the NOT gate in Fig. 1 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, but conclusions can be placed at any point the path. Other inference blocks such as counters and timers are provided for analysis of systems with time lags and delays.

Action blocks. In Fig. 1 a *control path* connects the conclusion block to two action blocks via an action control switch. Whenever the conclusion posts a TRUE output, a signal is sent to the switch which then reroutes 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 control signal; if the high value observation has a FALSE output, action 2 will receive the signal. For Actions 1 and 2 the evaluation methods do not receive data derived from the process -- they perform side-effects. The switch acts as a conditional statement, determining which action will be performed.

DESIGN ASPECTS OF GDL

The switch in Fig. 1 provides an important example of data flow and dynamic control integration in GDL applications - in this case a logical status derived from the real-time process explicitly regulates the branching of a sequential procedure that is carried out on the process.

Seamless integration at the application level stems from the implementation-level integration of dataflow and control flow in GDL. For data and inference blocks in particular, the connections specify both dataflow and control flow. Because of the dual role performed by data and inference connections, all of the information needed to execute an algorithm is provided in an IFD - there is no separate implementation step to combine the underlying object, dataflow, and dynamic models, and an IFD can be directly executed in a manner similar to discrete event simulation.

In the following sections, aspects of GDL structure relating to these capabilities are presented from an object-oriented modeling viewpoint.

The Object Model

In standard object languages, an object represents an active data structure. Operations (methods) that manipulate the data structure are associated with a block's class. One of the key benefits of object-oriented systems is that data and behavior are unified via the class hierarchy.

GDL block classes. Following the object-oriented approach, graphical objects in GDL are instances of classes defined in G2. The principal objects are the blocks. Block classes are organized in a hierarchy derived from the parent class, GDL-Block (Fig. 2).

The graphical objects in GDL become active as soon as they are connected and initialized. The application diagrams provide interactive system specification and runtime interface, with status indication by color and animation.

Attributes. Each block instance contains a set of attributes, as shown in Fig. 3 for an AND gate. The attributes for a block can be inspected by selecting the block with the mouse. Some attributes define configuration information, such as parameter values used to fine-tune a block's operation. For example, if a first order filter block has been created, the time constant of the filter can be specified. Other block attributes hold dynamic state information, such as the current output value(s), data histories, and the block status. The block status contains a symbolic value such as *reset*, *ready*, *running*, *locked*, or *error*. Block methods reference the block status during execution. For example, if the block-status is *locked*, a block's evaluation method will not compute or propagate a new output. Finally, all blocks have an attribute called GDL-id that can hold an optional symbolic tag name; this is often used to relate blocks to objects such as sensors or user interface components that are not graphically connected to the blocks.

Observations and conclusions have attributes that can hold text-based descriptions to associate with the logic states TRUE, FALSE, and UNKNOWN. For example, the description-when-true attribute of a

conclusion might read "Reactor shutdown is advised". This description is used by the dynamic explanation facility, described below.

Occasionally it is convenient to dynamically modify block attributes at runtime. A good example is the attribute that specifies a threshold limit value in an observation block. There is a data block in GDL that can store its output value into a named attribute of a target block connected to it.

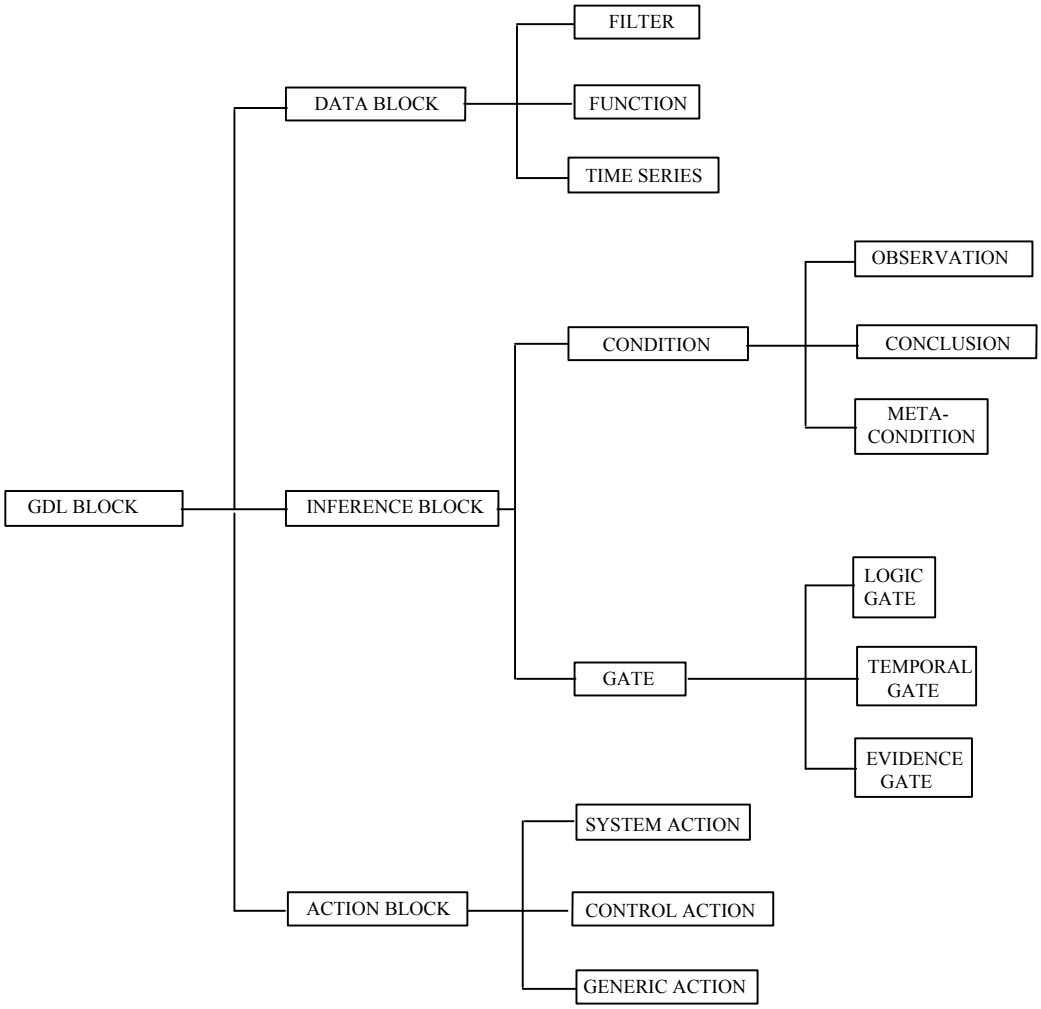

Figure 2. Partial GDL class hierarchy.

Associations. A key element in an object model is the definition of associations (relations) between objects (Rumbaugh, 1987). At the user level of GDL, the important associations for runtime execution are inherent in the connections, both feedforward and feedback, between the blocks.

However, some associations must be explicitly configured between objects that are not graphically connected. For example, the application developer must specify the data source (typically a G2 sensor object that receives data from an external control system or database) for each entry-point block. This specification is provided by matching symbolic tags entered into certain attributes in the sensor object and in the entry point block. The same mechanism is used to link user-interface facilities with components in an IFD.

Information hiding and reuse. In G2, any object can have a subworkspace upon which other objects can reside. This feature provides a generic mechanism for modularity and information hiding; in GDL it provides the foundation for IFD *encapsulation*. Encapsulation hides the details of a complex IFD by

packaging it on the subworkspace of single, higher-level object called an encapsulation block (Fig. 4). When an encapsulation block is evaluated, the encapsulated IFD is used to generate the output value(s) of the encapsulation block. Multiple levels of encapsulation are supported. That is, an encapsulated IFD can be composed of other encapsulation blocks. After an IFD has been encapsulated, its workspace can be hidden and the developer can move, connect, and even clone (duplicate) the encapsulation block like a normal block. Based on the cloning feature, the encapsulation block provides the template for a new, abstract block class whose evaluation method is defined graphically by the user. New class instances are generated via the G2 cloning operation.

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 3. AND gate attribute table.

One of the principal benefits of object-oriented systems relates to software reuse (Johnson & Foote, 1988). In GDL, users create reusable diagnostic software modules as encapsulation blocks using point-and-click mouse operations, i.e. without doing object-oriented programming in the traditional sense. This allows diagnostic experts such as plant engineers to build reusable modules and libraries, and to hide the implementation details as appropriate. These modules might represent advanced SPC strategies, sensor validation utilities, alarm filtering schemes, and so on.

Object Model Extensions. Typical users do not directly extend the GDL object model; however, advanced users who are familiar with G2 can define block subclasses and build their own methods for these classes using G2 procedures. This provides an alternative to building reusable IFD modules based on encapsulation blocks.

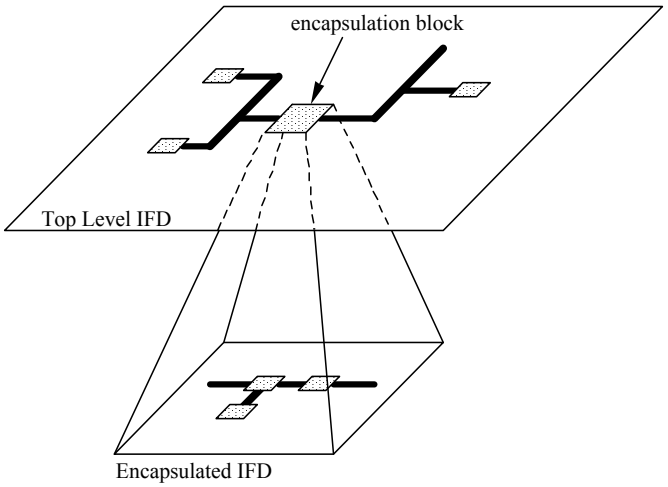

Figure 4. IFD Encapsulation.

The Dataflow Model

A traditional dataflow diagram shows the flow of data values from objects through processes that transform them to their destinations in other objects (Rumbaugh and co-workers, 1991). Connections in the dataflow diagram show all possible computation paths. In many respects, building a GDL application means building a dataflow diagram, but in GDL the concept of a dataflow diagram is slightly modified. Within an IFD, the objects (blocks) and processes (methods) are unified - an object of a particular filter class represents both the data structure for the filter (filter constant, current output value, etc.) as well as the particular filtering algorithm to be applied.

Methods. At the GDL user level, a block executes a fixed set of predefined methods. As previously discussed, evaluation methods determine the run-time behavior of a block. While most blocks have pre-defined evaluation methods, certain action blocks can be configured to execute user-specified methods defined in G2's text-based procedural language (this is one of several mechanisms in GDL to interface to the underlying G2 system and to the external world). Special auxiliary methods for functions such as initialization and explanation are also defined for certain block classes.

Capabilities. Major optional behavior of GDL blocks is configured using *capabilities*. Capabilities are separate graphical objects that can be connected to blocks to impart optional behavior, such as the ability to set an alarm, during execution of evaluation methods. From an object-oriented programming viewpoint, capabilities provide some of the features of *before* and *after* methods in an object language such as CLOS (Keene, 1989), with the advantage that capabilities can be specified for individual block instances. The main advantage of this approach is that the presence of an optional function is visible in the IFD rather than being hidden in the configuration attributes of the block.

The Dynamic Model

The focus of GDL's dynamic model is the branching and sequencing of evaluation methods during forward chaining. IFD's are fundamentally data driven programs executed by the G2 real-time scheduler. For blocks that are connected, forward chaining involves data *and* control flow (data and inference blocks), or just control flow (action blocks).

Data and inference paths propagate both output values and *program control signals* (PCS). During forward chaining, a PCS signals a block to execute its evaluation method. 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.

Logical control of PCS. GDL provides data and inference *inhibit* blocks that operate only on the input PCS. These blocks use inference output values to determine how the PCS is regulated. They receive an auxiliary logic input, and either pass or inhibit the propagation of a PCS on the path, depending on the logic status value. This is useful for activating or deactivating entire branches of an IFD that might be associated with particular process phases such as startup or shutdown. It can also be used for functions such as alarm filtering: when other conditions are detected, an inference inhibit can block an inference path so that a particular conclusion block cannot trigger an alarm.

In addition to inhibit blocks, there are also blocks that perform switching operations on data and inference paths. These blocks route PCS signals *and* data or inference values between sets of input and output connections. Like inhibit blocks, the routing is determined by the status of an auxiliary logic input. The highest level of dataflow and control integration is represented by these blocks, since they dynamically alter the direction of dataflow in an IFD according to feedforward or feedback signals derived directly from the process.

Inference output filtering. When the new output value computed by the evaluation method for an inference block is same as the existing value, no PCS is sent to downstream blocks. This feature can dramatically

increase the evaluation efficiency of a large-scale diagnostic application. Since it acts to terminate a PCS, logic filtering clearly plays an important role in the GDL dynamic model.

Inference blocks can also be locked, so that the output status value remains fixed for a period of time. The lock feature can be invoked by an action block, or automatically via a built-in facility that reduces transient disturbances (logic 'chattering') by applying hysteresis or hold periods to the logic outputs of specified blocks (Finch, Stanley, and Fraleigh, 1991). An output override capability is also provided as an extension of the lock facility.

Application-level sequential control. GDL manages low-level program control transparently from block to block. At the application level, higher-level sequential control in response to detected conditions is configured explicitly using action blocks. An action sequence can be triggered by an observation or condition: when the output value changes to TRUE, a PCS signal (but no data) is also sent to any connected action blocks. The PCS can then propagate to a series of connected actions along action paths to carry out a sequential procedure or set of procedures.

Control actions. Control action blocks are used to graphically configure standard programming idioms such as branching and iteration. A unique aspect of GDL is that predicates for these operations are provided in the inferencing portion of an IFD. For example, Fig. 5 shows an action sequence to manage the startup phase of a batch reactor. The key events in the action sequence are triggered by changes in the reactor level during loading as analyzed by inference blocks. The feedback control paths in the action sequence specify 'wait until TRUE' steps - while the logic inputs to the action switches are FALSE the PCS is recycled; when the inputs are TRUE the PCS branches to the subsequent block in the action procedure.

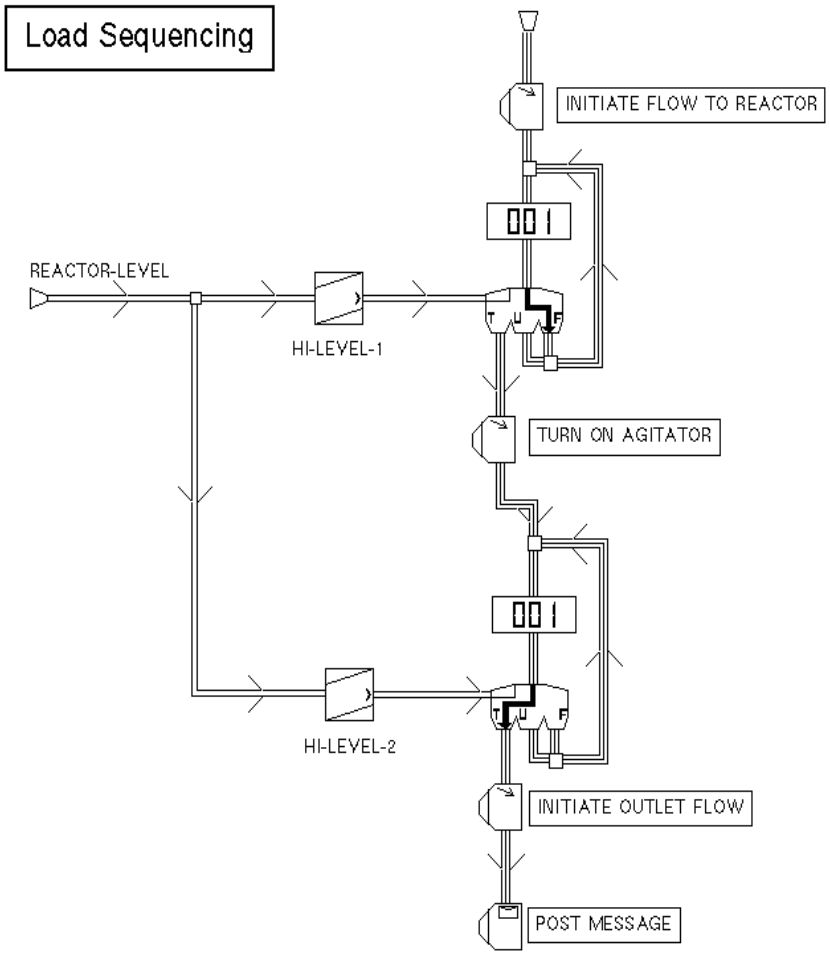

Figure 5. Information flow diagram for reactor load sequencing.

An important control action block (not shown here) is the checkpoint. A control path can fork into parallel branches so that separate action sequences can be performed concurrently and in parallel. If both branches terminate at a single checkpoint block, this block schedules a wait state until both sequences have completed, before passing a PCS to subsequent blocks. Other control action blocks are provided for accumulating the count of PCS signals propagating on a particular control path and for propagating a prespecified logic value on an inference path 'on-demand'. The counter can output a logic value on an inference path when the count reaches a specified level. These blocks function as gateways from sequential control procedures back into the inference portion of an IFD, and consequently provide another unique aspect of GDL.

Action links. Some action blocks can be connected to a target block via action links. These blocks modify the state of their target block by automatically triggering an operation normally performed manually by the user. Example operations are block reset, block lock/unlock, modifying a configuration attribute value, and displaying the subworkspace of a block. This latter facility can be used to drive operator dialogs and other user interface components in response to real-time events, e.g. for emergency, maintenance or operations procedures.

Explanation facility. A key element of a diagnostic language is the ability to generate explanations from alarms and other conditions. In GDL, the explanation methods generate text-based explanations in response to user commands. The flow of control during explanation generation is reverse to that of normal control flow for inference paths. The explanation method for the appropriate conclusion is invoked and this recursively calls the explanation methods of the logic gates and other inference blocks in the inference paths leading to the conclusion. For logic gates, each explanation method applies a boolean algebra operation specific to the type of gate.

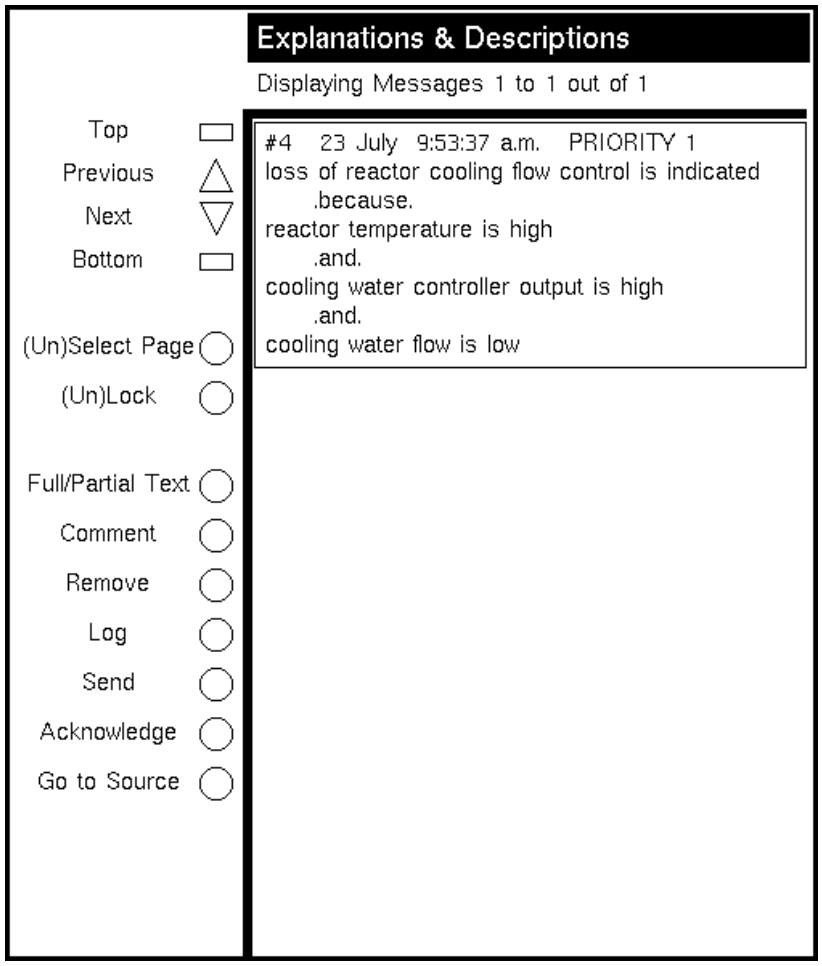

Figure 6. Sample explanation message.

The recursion stops when all observation blocks responsible for the state of the chosen conclusion are located; the text-based explanation is comprised of the logical states of these observations, or optionally, the text-based descriptions that are associated with the different logic states and stored as attributes within the observation blocks. Fig. 6 shows a sample output message from the explanation facility.

FUTURE EXTENSIONS

Future emphasis will be placed on graphical blocks for automated model building and on-line prediction. These tools have many applications such as multivariable statistical quality control and model-based fault classification. The basis for these enhancements will be the introduction of a new connection type for passing vectors between data blocks. Object-oriented treatment of data vectors provides opportunities to integrate multivariate statistical analysis and even neural networks directly into IFD's.

In addition, adaptive techniques are being evaluated for on-line learning of GDL configuration attributes such as limit thresholds and fuzzy membership functions.

CONCLUSION

Because of its unique integration of dataflow and sequential control, GDL supports a variety of techniques, such as fault trees, filters, flowcharts, decision trees, and GRAFCET-style procedural control. Common

applications of the system will be alarm filtering, fault detection, and recovery from extreme conditions. The economic incentives for GDL applications are product quality, equipment protection, environmental protection, and assuring a good set of measurements for use in control and optimization schemes.

REFERENCES

Arzen, K.E. (1991) Sequential function charts for knowledge-based, real-time application. *Proc. 3rd Intl. Workshop on Artificial Intelligence in Real Time Control*, Sonoma, CA.

Finch, F.E., G. M. Stanley, and S.P. Fraleigh (1991). Using the G2 diagnostic assistant for real-time fault diagnosis. To appear in *Proc. European Conference on Industrial Applications of Knowledge-Based Systems*, Segrate (Milan), Italy, Oct. 17-18 1991.

Johnson, R.E. and B. Foote (1988). Designing reusable classes. *J. Obj. Oriented Prog*. 1(2), 22-35.

Keene, S.E. (1989). *Object-Oriented Programming in Common Lisp: a Programmers Guide to CLOS.* Reading, MA: Addison-Wesley.

Nilsson, A. (1991). *Qualitative model-based diagnosis - MIDAS in G2*. MS Thesis, Dept. of Automatic Control, Lund Institute of Technology.

Rowan, D.A. (1988). AI enhances on-line fault diagnosis. *InTech*, 35 (5), 52.

Rumbaugh, J. (1987). Relations as semantic constructs in an object-oriented language. *ACM SIGPLAN* 22(12), 466-481.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson (1991). *Object-oriented modeling and design*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Santori, M (1990) An instrument that isn't really, *IEEE Spectrum*, Aug., p. 36-39.

Sarraut, P. (1991). *Implementation of a toolbox for colored Petri nets in G2*. M.S. Thesis, Dept. of Automatic Control, Lund Institute of Technology.

Stanley, G.M., F.E. Finch and S.P. Fraleigh (1991). An object-oriented graphical language and environment for real-time diagnosis. *Computer-oriented process engineering - proceedings of COPE-91*, Barcelona, Spain, Oct 14-16, 1991. (L. Puigjaner and A. Epuna, eds). Amsterdam: Elsevier. pp 265-270.

Weber, R. and Lalka, C. (1991). Real-time diagnostic toolkit. *Proc. World Congress on Expert Systems*, Orlando, FL., p. 1903-1910.

Contact Greg Stanley at

http://www.gregstanleyandassociates.com/contactinfo/contactinfo.htm