

Presented at: European Symposium on Computer Applications in Chemical Engineering, COPE-91, Barcelona, Spain, Oct. 14-16, 1991

AN OBJECT-ORIENTED GRAPHICAL LANGUAGE AND ENVIRONMENT FOR REAL-TIME FAULT DIAGNOSIS

G.M. Stanley*, F.E. Finch and S.P. Fraleigh

Gensym Corporation, 125 CambridgePark Dr., Cambridge, MASS. (USA)

ABSTRACT

A new, extensible graphical language GDL (Graphical Diagnostic Language) addresses fault diagnosis in static or dynamic systems. GDL is used to detect faults, classify the root causes of the faults, initiate corrective actions, recognize recurring problems, plan and execute tests, and manage alarm displays and messages. GDL is an environment for specification, development, run-time use, and maintenance.

GDL is comprised of blocks defined in an object-oriented environment. Each block can transform, combine, or manipulate incoming data via a predefined algorithm. Blocks are connected graphically to form information flow diagrams (IFDs). IFDs provide both system specification and run-time interface, complete with status indication by color and animation.

Techniques necessary in real-time systems are supported, including task prioritization, asynchronous concurrent operations, and real-time task scheduling. Signal processing and statistical process control blocks generate events from historical data. Other blocks provide event detection, event filtering, fuzzy and discrete logic, and event sequence recognition. Action blocks are used for control.

The system supports a variety of techniques, such as fault trees, filters, flowcharts, and decision trees. A common application of the system will be alarm filtering. 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.

INTRODUCTION

Graphical Languages

The advantages of the graphical language approach are simplicity and declarativeness. Graphical languages allow much of the complexity of the underlying functions to remain hidden from the user. The user deals with

graphic tools and icons, each encapsulating one or more underlying programs.

Graphical languages are often interfaces to standard text-based languages, rather than independent languages. Because the graphical language operates on a higher level, the user need not be an expert in the underlying text-based language. It is possible to implement a common graphical language using several different text-based languages, making the graphical language portable from machine to machine. Furthermore, graphical languages bypass linguistic barriers when developing applications for use in multiple countries.

Often a graphical language is a less flexible, more constrained environment. However, in many applications, the additional constraints of a graphical approach are a benefit. By providing the user with fewer choices, a graphical language may be easier to use and understand. Enforcement of constraints can also minimize programming errors.

The most powerful argument for graphical languages is the ability to program in a style that closely mimics the way people model problems. Examples of mental models that are naturally represented graphically include fault trees, decision trees, ladder logic, organization charts or other hierarchical decompositions, program flowcharts, project management schedules, and system schematics. Graphical languages fit users' existing perception of the problem, making the application easier to build, debug, document, and maintain. An added benefit is the use of color and animation to provide real-time feedback as the system operates.

Purpose of GDL

The Graphical Diagnostic Language (GDL) addresses the following requirements associated with diagnosis:

- Filtering/signal processing/statistical analysis
- Detecting fault symptoms
- Identifying root causes

- Generating and managing alarms
- Planning and executing further active tests
- Giving advice and taking corrective actions
- Identifying and correcting recurring problems (meta-alarms)
- Predictive maintenance based on cumulative operating time

These abstract problems translate into application areas such as safety, quality control, equipment protection, yield/production maximization, loss prevention, and environmental protection (refs. 1-3). The diagnosis of faulty sensors or other equipment is also a prerequisite for success of online optimization.

To address these problems, GDL supports the implementation and combination of a variety of existing diagnostic and analysis techniques, such as:

- Comparing the results of sensors, and checking limits, rates of change, and variances
- Logic networks (graphical representation of rules, calculations and procedures)
- Alarm filtering to eliminate redundant alarms and chattering
- "Pattern of failure" recognition
- Fault trees
- Voting logic, fuzzy logic, neural networks, and other evidence combination techniques
- Decision trees and troubleshooting diagrams
- Model residual analysis

Domains for which this language is well suited include plants of all types, such as discrete manufacturing, batch and continuous chemical processes, and electrical power generation. Other domains include environmental monitoring, telecommunications, electrical power distribution, aerospace remote telemetry monitoring or on-board fault diagnosis, medical monitoring, and financial trading analysis.

OVERVIEW OF GDL

Overall structure

GDL is a real-time object-oriented system. The language provides graphical objects (blocks) with attributes specifying their behavior. The user creates an IFD by connecting objects graphically. The objects can be connected and configured interactively while the system is running.

Examples of the objects include filters, variance calculators, "AND" and "OR" gates, sequence recognizers, and objects representing procedures. Examples of attributes include time delay parameters, and logic type (fuzzy or discrete).

Directed connections between the objects represent information flow, or program control. Each block has a

unique iconic representation that identifies its function to the user. Similarly, different types of connections are color-coded.

The overall run-time strategy for interpreting the GDL diagrams is forward chaining. Forward chaining, as found in expert systems, passes new information from the output of one block to the inputs of the next blocks. Data enters the system, typically is filtered, and then events are generated from an analysis of the filtered data. The results are passed through a series of logic gates and action objects that take appropriate action when a particular failure is recognized by the logic.

GDL filters out small changes to minimize "alarm chattering". Alarm chattering (i.e. rapid changes in alarm status) is caused by small variations in the input signal when values are near an alarm limit.

Figure 1 illustrates a simple IFD created from GDL blocks. Numerical sensor data enters the diagram at the entry point block (far left) and flows from left to right. Just to the right of the entry point the data path branches (GDL supports unlimited branch points). The trend calculator on the upper branch performs a linear regression and outputs a rate of change to Event Detector #1. Event Detector #1 outputs TRUE if its input exceeds a predefined reference value and FALSE otherwise. The changeband filter on the lower branch stops small changes in input value from propagating to Event Detector #2. The outputs of the event detectors are combined by the AND gate. The AND gate will produce a TRUE output when both event detectors output TRUE (i.e. the data value is high and increasing). Next, an inferred event stores the result of the AND gate (here the inferred event is shown latched as indicated by the lock symbol). The message action sends a message to the end-user.

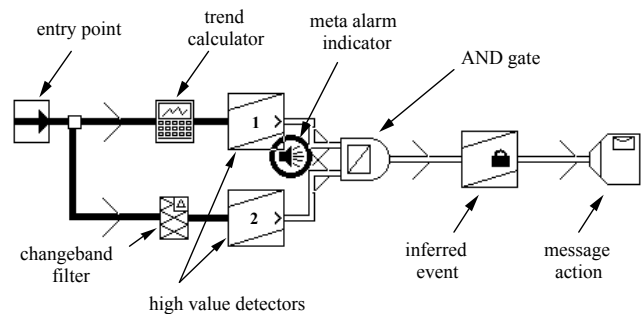


Figure 1

At runtime, the system animates the diagram with colors and symbols to indicate the status of each block. For example, blocks with TRUE output are shown in red while those with FALSE output are shown in green (the actual colors are user configurable). The meta-alarm

indicator attached to Event Detector #1 indicates a recurring TRUE condition. The lock symbol on the inferred event shows that the system has latched the current output value of the block (latching is an optional behavior of inferred events).

An IFD can be decomposed into individual modules that can be developed and viewed separately. Modules can interact within an application. In addition, an icon can represent an entire IFD, so that hierarchies of diagrams can be created. Information can be passed up and down the hierarchy. These hierarchical diagrams can be thought of as graphical macros. Hierarchical graphical languages are discussed in ref. 4.

GDL is highly declarative. For example, interactions between blocks are explicitly drawn in the IFD. The diagram can be analyzed using the underlying expert system. For instance, rules and procedures analyze the diagram for inconsistencies. An explanation facility analyzes the graphical structure so that the reasoning behind a given conclusion can be presented to the user in text form.

In addition to the language blocks, a development, debugging, and run-time support environment is provided. The overall system is called the Diagnostic Assistant™.

The system is built on top of an underlying real-time, object-oriented tool, G2, in which the objects are created (refs. 5-7). This underlying expert system provides facilities for asynchronous data acquisition and other interfacing, task scheduling, object definition, user interface, historical data collection, trending, rules, dynamic simulation for model-based reasoning, and a procedural language, which can all be used to support GDL. For instance, there is a block in GDL providing access to a procedure implemented using a text-based procedural language.

Real-time Considerations

To fully represent a dynamic system, it is necessary to maintain the values of a set of state variables. The objects in the graphical language provide storage locations for these variables. The end-user interface also accesses these objects. For instance, the graphical objects provide convenient places for an end-user to override data and conclusions made by the diagnostic system.

The language blocks are used for specification, run-time debugging and end-user interface. The graphical diagram is made "live" at run time through animation and color changes, so that the current status and its explanation are obvious. This significantly reduces the

time to debug an application, and explain it the results to end-users.

The language supports the needs of real time systems, including:

- Asynchronous data acquisition blocks, which can either poll or respond to external events
- Real time scheduling of all activities, with priorities and asynchronous processing
- Signal processing, such as filters
- Historical data collection and time series analysis, such as rate of change, variance, and statistical process control (SPC)
- Time-based elements, including filters, latches, counters, and analog and logic delays
- Recognition of concurrent or sequential events, with fuzzy comparison
- Validity intervals - a variable without a recent value fails to an "unknown" current value
- Alarm and message management, with provisions for acknowledgement and logging.
- User overrides for any data or conclusions, with a graphical "lock" symbol so that the overridden status is obvious.
- Provisions for taking sensors and alarms out of service

GDL COMPONENTS

Language Blocks and Connections

The major types of blocks in the language cover:

- Calculation and signal processing
- Simple event recognition (high/low signals, high/low rate of change, etc.)
- SPC calculations and event detection
- Time series analysis (e.g. regression)
- Logic (AND gates, OR gates, neural network evidence combination nodes, etc.)
- Event sequence recognizers
- Program control (e.g. branching logic based on an input signal or user input)
- Actions (e.g. sequential execution of procedures, user interactions, message generation)
- User interface objects (e.g. alarms, readouts, message management)

The graphical connections types include:

- Data paths, passing numerical, logical, or symbolic value from the outside world and through filters to calculation blocks and event recognizers
- Inference paths, passing fuzzy logic values and symbolic status information between various blocks
- Control paths, controlling the sequence of calculations for switches and other selectors
- Action paths, specifying the "target" of an action, such as a reset

Inhibition of excessive computation caused by noise is a major issue in real-time forward-chaining systems. This

problem is addressed by filtering of the analog signals, by deadbands, SPC, fuzzy logic in the conversion of analog signals to events, and by "logical event filtering". Logical event filtering includes latching for a specified time period or delaying decisions until the input has been continuously true for a specified time period.

Data conditioning, calculation and signal processing

The signal processing blocks include filters, estimation blocks, and time series analysis blocks, including SPC. Filters include the change band filter, outlier removal, an exponential and nonlinear exponential filter, and moving averages. Also included are linear, quadratic, and cubic data fits for filtering and rate of change estimation. Blocks are provided for SPC functions such as CUSUM, X Bar, and range tests.

The change band filter does not propagate a change until the input changes beyond a certain deadband centered around the last input value. This is a form of data compression, sometimes used in historical data collection schemes to reduce the amount of memory required to store historical data. This filter is well-suited to eliminating propagation in a forward chaining system. The SPC blocks can be thought of as sensitive filters. They convert historical data trends into discrete quantities that can be tested for high or low-value events.

Calculation blocks for standard mathematical functions such as addition and multiplication are provided.

Event detection blocks

Event detection blocks convert numerical signals into logical conditions. The developer can specify deadbands and fuzzy logic in an event block such as a high-value detector. A deadband adds hysteresis, so that small, high-frequency noise will not cause propagation of logical output changes. This particular approach is commonly used in computerized alarm systems, to avoid alarm chattering. Fuzzy logic provides a continuous, normalized transition from "no alarm" to "alarm".

Logic blocks

Once events have been detected, a diagnostic system needs to analyze logical combinations of events. The logic gates in GDL include types such as "AND", "OR", "NOT", equivalence, and others. The gates can be configured for either fuzzy or discrete logic.

The logic blocks use event filtering to inhibit excessive forward chaining. Events can be latched for a specified minimum period of time, to ignore the effects of short-term noise. Similarly, decision-making to change an output can be delayed until the input condition has been continuously true for a specified period of time. This prevents premature conclusions during a transient condition.

In addition to discrete and fuzzy logic, other forms of evidence combination are supported. One form of evidence combination node provides a weighted linear combination of input fuzzy truth values, followed by an optional sigmoidal nonlinearity. With this nonlinearity, it is possible for the belief in a conclusion to be higher than the belief of the individual inputs. This is desirable when imperfect, redundant measurements are combined. In certain configurations, evidence combination blocks behave similar to neural network nodes. This provides a link between graphical rules and a run-time neural network, although learning is currently not provided as part of the system. Evidence combiners also provide characteristics similar to other evidence combination schemes used in expert systems.

With fuzzy values and evidence combination nodes, it is possible to avoid the brittleness of traditional expert systems based on discrete logic. For example, traditional fault trees and decision trees can be sensitive to sensor failures.

Voting logic systems can be implemented with GDL's evidence combination features. For example, in a compressor shutdown system, the developer can combine evidence from a variety of bearing temperature sensors and vibration sensors at various frequencies, orientations, and locations. With voting logic, no single sensor failure can cause an unnecessary shutdown. As more sensors indicate a problem, the belief that a problem is present increases.

Sequence recognition gates are similar to "AND" gates, except that they require the inputs to occur in a fixed time order. A fuzzy truth value is generated, so that the developer can account for uncertainty in the timing of the actual events vs. the detection of the events. That uncertainty can arise from process variability, heavy filtering, use of SPC techniques, or other process or computational delays. Similar gates recognize concurrency, such as one state occurring "during" another one. These gates are useful for diagnosing causal systems with time delays, because the order of occurrence of the events can help identify the root cause. When one event will cause another, following some delay, a sequence gate recognizes that the events occurred in the expected order for a given root cause.

Control of data flow is accomplished using blocks which act as switches. The logic switch routes the data to one of several destinations, based on a control input signal with possible values of true, false, or unknown. Similar blocks are used for end-user input, where the end-user can choose among up to four alternatives. These blocks can be used to develop traditional manual input decision trees, for example. (Although traditional fault tree and

decision tree analysis is susceptible to instrument failure or similar errors in many sensor-driven real-time systems).

Action blocks

The action blocks can link to end-user interfaces, specify sequences of corrective actions, or control active testing. The action blocks for end-user interfacing carry out tasks such as sending messages to a text message manager, querying the end-user for manual input, showing a display, and calling procedures which play back sound files. Active testing means planning a test, manipulating some variable in the system, waiting for the results, and making a decision on what to do next: perform a new test or complete the diagnosis.

Facilities are provided in the language to recognize and act on recurring alarms. This is important for recognizing problems such as:

- Controller tuning leading to excessive oscillations
- Overly-conservative alarm limits
- Noisy sensors
- Recurring or intermittent failures
- Poor operations practice

Other features

A "meta-alarm" (an "alarm about alarms") condition is optionally generated for an event block when its alarm frequency exceeds a certain rate per hour, as set by the developer. When the condition occurs, a meta-alarm symbol appears next to the event block, and is also propagated automatically to any alarm windows associated with the event block. A meta-alarm action object can be attached to the event block as well, if actions are to be taken. For instance, actions might include changing of controller tuning or setpoints, resetting of alarms limits, or suppressing the rate of occurrence of the underlying alarm.

A framework is provided for model-based analysis. A comparator block is used to calculate a model residual by computing the difference between a model output and measured data. The residual is propagated to an event detection block, or an SPC test. For instance, a simple model using pressure and temperature may be delayed and compared against an online analyzer which only reads a measurement at 10 minute intervals. The online measurement is in turn compared to the results of lab samples. The resulting pattern of residuals can point to failure in the sensors, analyzer, or lab. Simple models could be used, as could full differential equation models.

Comparison of GDL and other graphical languages

GDL shares some characteristics of GRAFCET (ref. 8), a graphical language designed for sequential control applications. GDL executes sequences of actions by following directed connections between objects which

represent procedures. GDL and GRAFCET both highlight the currently active object at run-time.

However, GDL differs from GRAFCET particularly in that it can represent information flow as well as program control structures. GRAFCET does not represent information flow or variables. By comparison, GDL objects can pass numerical or fuzzy logic values through inference connections. The sequencing of GDL actions is implicit in the forward chaining through the inference connections, and also explicit when passed through control connections.

This data-flow style of graphical representation is appropriate in real-time systems which often have large quantities of data passing through. This style is more comfortable for many existing users of real-time systems than a production-rule system or GRAFCET.

GDL can also represent the "target object" of an action, while GRAFCET just focuses on program control aspects of sequential and concurrent actions. For instance, blocks can force resets or evaluations of other blocks by following an action-path connection.

GRAFCET focuses on the high-level program steps, and leaves the implementation to another, lower-level language. In contrast, GDL handles both levels. It provides objects which directly execute their own standard programs, such as filters. GDL is also extensible, so that other code outside GDL can be executed as well.

GDL shares characteristics of other graphical object-oriented languages. Typically, the icon representing an object can be inspected by selecting it with a mouse to show a table of attributes. Details of the object's behavior are stored in the attributes. Thus, details can be hidden from a high-level view, but the details can be made immediately apparent. LabVIEW (ref. 9), developed for laboratory applications, is an example of such a language.

GDL differs in philosophy from some CASE tools which also have graphical representations. With CASE tools, a graphical language is often used for specification, but this is translated into another target language for execution, and the interactive debugging can be lost. In GDL, the graphical language itself is executed, so that the end-users and developers can interact directly with the high-level specification.

REFERENCES

1. D. A. Rowan, On-Line Expert Systems in the Process Industries, AI Expert, August, 1989, pp. 30-38.

2. G. E. Mertz, Application of a Real-Time Expert System to a Monsanto Process Unit, Proc. Chemical Manufacturer's Association Process Control Conference, Miami, Florida, March, 1990.
3. J.F. Muratore et al., Real-Time Data Acquisition at Mission Control, Comm. ACM, Dec., 1990, pp. 18-31.
4. H. Elmqvist, S.E. Mattsson, IEEE Control Systems Society Third Symposium on Computer-Aided Control System Design (CACSD), Arlington, VA, Sept. 24-26, 1986.
5. R. Moore et al., The G2 Real-Time Expert System, Proc. ISA, Oct. 16-21, 1988, pp. 1625-1633.
6. R. Moore, H. Rosenof, and G. M. Stanley, Process Control Using a Real-Time Expert System, Proc. 1990 IFAC, Estonia, USSR, 1990, pp. 234-239.
7. A.G. Hoffman, G.M. Stanley, and L.B. Hawkinson, Object-Oriented Models and Their Application in Real-Time Expert Systems, Proc. Society for Computer Simulation International Conference, San Diego, 1989.
8. A.D. Baker et al., GRAFCET and SFC as Factory Automation Standards: Advantages and Limitations, Proc. American Control Conference, Minneapolis, Minn., June 10-12, 1987, pp. 1725-1730.
9. M Santori, An instrument that isn't really, IEEE Spectrum, Aug. 1990, pp. 36-39.

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