Flexible Architecture for Factory Automation

INTRODUCTION

The cost of computer computation hardware, sensors and electronics has dropped – and the size of these pieces has shrunk dramatically. This enables a ubiquitous opportunity for a host of new applications. While the cost of hardware has dropped, the cost of a programmer has not. These new products can only be viable if the cost of software can also be reduced. Some software components, such as the operating system, can be universally used across a wide variety of applications and come at no cost, such as Linux. But when it comes to delivering a specialized solution, a ready COTS (commercial-off the-shelf) solution is not available. Instead of developing a solution from scratch, the cost and time to market can be greatly reduced by assembling a solution from a software application framework. The framework developed by SIGMADesign provides robust, configurable software for factory automation while reducing cost and time by enabling the reuse of existing modules for a variety of applications deployed on a PC platform. This paper discusses the framework’s architecture, the division of system parameters for repeatable process control and system resources provided by the framework. System resources include event notifications, logging, exception handling, reporting status/state changes and archive/restoration of parameters.

FRAMEWORK GOALS

SIGMADesign’s application framework must maintain flexibility while reusing existing components to configure a system. The components must have few dependencies but also self-contained to manage a subsystem (loosely coupled but highly coherent). These two design goals are often at odds with each other. A component that manages a subsystem is coupled to system resources for workcell supervisory control, logging events, and archive/restoration of recipe/configuration parameters. The design must provide a robust and simple solution to integrate subsystem components together without reworking the workcell supervisory controller or system resource infrastructure. The application framework is focused on machine control and factory workcell automation, but the concepts must apply to applications all the way down to embedded systems that require supervisory command and coordination of subsystems.

FRAMEWORK ARCHITECTURE

The architecture is a distributed system with centralized high level control. The software structure is centered on the Model/View/Controller (MVC) pattern. To illustrate the architecture of the application framework, we can use the structure of a business as an analogy. A business is sectioned into three main groups: 1) sales team 2) management and 3) departments. Each is given unique responsibilities; sales is the face of the company and interfaces with the customer, management translates sales orders into a sequence of work orders and departments execute the work orders.

1. Sales Team → View: The view provides the software user interface just as the sales team provides the customer interface for the company. They provide a single line of communication and conceal the details of the organization from the user. The sales team can market and take orders without having to know the details of how those products are made. The sales team keeps the customer updated on the status of orders similar to the view providing feedback of parameters, state, and status of system. The view receives asynchronous updates from the model/departments on the status of orders that are in process for the user to view. The view is loosely coupled to the system through a symbolic interface that is adaptable to application reconfiguration changes. Loose coupling also supports multiple views such as a factory interface for remote workcell monitor/control and the local user interface for operator use at the tool.
2. **Management → Controller**: The controller contains the application business logic which a company’s management team is responsible for. The view sends the controller high level instructions which are then broken down by the controller into low level commands and sent to the model. Similarly, the sales team forwards customer purchase orders to the management team to be broken down into a schedule of work that is sequenced down to the departments for processing. The controller isolates the view from the model just as the sales team sends purchase orders to the management, not directly to departments. By centralizing the business logic in the controller, the model can be application independent and reused by other similar applications. Because of the concurrency of many simultaneous tasks in the various layers, the framework is designed around a multithreading architecture that maintains integrity, minimizes latency, and prevents deadlocks.

Complex workcells that coordinate concurrent processing of multiple lots (possibly executing different recipes) require a scheduler who knows the work order prioritization, requirements, and business logic for coordinating departments into a plan. The executive controller then executes the scheduled plan. The schedule consists of a chronological sequence of subsystem commands that transport and process lots through the system. The scheduler tries to maximize throughput balancing the limited resources available. An important feature of the system is its ability to recover from processing errors and failures. The system can request the scheduler to do different types of rescheduling to recover from process errors and failures.

3. **Department → Model**: The model provides a simplified abstraction of the physical devices it controls just as a business’s various departments are self-contained functional groups that can independently perform tasks and report back results. The various departments/models in a company provide a cohesive standalone grouping of common functionality without the management/controller knowing intimate details of how the department runs. The goal of each department is to achieve a self-sufficient existence; a department should be aware of its capabilities and limitations, aware of the environment it runs in, self-diagnostic, self-recovering.

Modularity and reuse are improved by further subdividing the model into supervisor, driver, and devices. The analogy in business would be to break the department down into supervisor, courier, and worker.

3.1. **Department Supervisor → Supervisor**: The supervisor provides the interface and control of the department. The supervisor’s reporting structure to management is the same across all departments, which simplifies management’s high level coordination task. There are common methods for configuration, initialization, self-test, and shutdown. The tight coupling between models is minimized by not allowing peer to peer communication. Communication between model/departments is only allowed if routed through the controller. This enables centralized control and reusable modularity. Departments/models are typically event driven and normally quiescent while waiting for events to occur but depending on their responsibilities they may require polling of devices.

3.2. **Department Courier → Driver**: The model driver is analogous to a courier that transports information between the supervisor and the worker. Similarly, the software driver translates high level device commands/responses to and from the device. Implementing the driver as a standalone module promotes reuse for various applications. A device that communicates over a serial link, such as RS-232 or Ethernet, requires a driver to establish a link over the network, to serialize messages, perform error checking and de-serialize responses. The driver is responsible for managing the serial bus protocol. Serial buses use various protocols to packetize the data with a destination address, function code, data and checksum/CRC. The driver can retry a failed transmission and keeps statistics of corrupted or failed transactions. Drivers can be specific to a unique device that uses a custom protocol or generic enough to be used by several different devices that communicate with a protocol standard, such as CAN, MOD, or EthernetIP.

Just as a courier can deliver information to multiple supervisors, the driver can also act as a server to allow shared access to a device by one or more supervisors. The driver must provide multi-thread protection since it is concurrently supporting multiple requests. The server driver should also perform its task quickly enough to not cause any application latency when simultaneously servicing multiple requests. An example of a device used by
multiple supervisors would be a bank of relays. The relays connect to actuators distributed over several devices but the relay hardware requires a single driver/server to coordinate concurrent access.

**3.3. Department Worker → Device:** Devices are the analogous factory workers that provides services to actuate or sense. From an external view of the system the user only sees the external devices but behind each device there is a significant amount of software that is required to integrate it into the system. The software framework developed by SIGMADESIGN provides a configurable structure that enables the simplified integration of various devices. There is a wide spectrum of devices that are integrated into automation applications. They range from complex 5 axis robots to simple digital I/O for sensing a door closed. A data repository would also be considered a device that provides access to a remote database for information used by various supervisors. Other examples of automation devices would be output drivers, relays, motors, lights, and sensors such as analog, digital, and temperature sensors.

The five software layers to control a Fused Deposition Modeling (FDM) workcell 3D printer are shown in Figure 1. When a local user (or the remote factory interface) directs the system to build a 3D part, the scheduler generates the sequence of motion profiles. The motion sequence is sequenced by the controller to the various models. Before processing, the controller must verify safety checks are satisfied and nozzle temperatures have reached the recipe set point. The digital I/O driver is shared by several of the supervisors for actuation and sensor feedback. The temperature sensor is used by both the nozzle heater supervisor for closed loop temperature control and the safety supervisor which monitors for excessive temperature. There are no links between supervisors since peer to peer communication is only allowed by routing through the controller.

**REPEATABLE PROCESSES CONTROL**

Factory automation must guarantee that a product is processed with the same conditions even if they are processed on different workcells, countries or even in different years. The application framework supports the following abstractions to ensure a consistent processes is executed by the tool. The various processes parameter files and the scope of how they are shared between systems and between factories is shown in Figure 2.

- **Lot** - Manufacturing cells process products in units called *lots*. The term lot is an abstraction which applies to a group or a single item.

- **Work Order** - The work order lists the sequence of workcells to process a lot and links the process recipe associated with each workcell. The recipe along with configuration, calibration and compensation parameters is sufficient to process the lot.
- **Factory Controller** - The factory controller manages a lot’s process recipe and work order. It also tracks a lot’s progress and archives process results reported by the workcell into a database as a virtual work order traveler.

- **Recipe** - A process recipe contains the complete instructions and process parameters to process a lot. A workcell’s calibration and compensation parameters allow a process recipe to be system independent. The recipe can be used across similar workcells on the factory floor without the need for customization on a workcell by workcell basis. When a lot is run on a workcell it shall be processed with the most recent recipe. The recipes should be stored on a central networked server so changes made by a process engineer on one system can be checked in and then automatically distributed to all systems. An advanced recipe server provides revision control to log what was changed and who approved the change. Ideally this revision tracking is done down at the recipe parameter level.

- **Calibration** - Parameters which tune the systems performance to adjust for environmental and system hardware variances are stored in calibration files. Calibration ensures all systems have the same behavior and performance so recipes are system independent and do not required tuning to run on a specific system.

- **Compensation** - Parameters that change often and are tuned at system initialization or during run time are stored in compensation files. While calibration can be time consuming and require special fixturing that provide traceability to standards, compensation can often be done automatically with little or no fixturing.

- **Factory Configuration** – Parameters that customize the application for a specific factory and are applied globally across the factory. Examples would be user interface settings such as language, permission levels and what features to enable/disable. Factory setting would also include settings to connect to databases, recipe server, and factory controller.

- **System Configuration** - Parameters which are independent of the recipe and software version are stored in system configuration. Hardware configurations are typically stored in the system configuration and includes what features are installed on the system, how IO is mapped into the system, and factory specific settings such as user accounts/language settings. A smart motor connected to a comm port would be a system configuration. The comm port is independent of the recipe and stays the same no matter what version of application software is installed. Through the life cycle of the tool, new hardware upgrades may be added to the tool which require updated configuration settings to define and support the upgrade.

- **Application Configuration** – These settings are specific to the application the workcell is performing and are universal across all tools performing the same process. The application configuration is tightly coupled to the current revision of software. The application framework flexibility comes from its’ configurable architecture. Application configuration includes items such as scripts used by a subsystem controllers (i.e. motor controller or PLC), exception response handling, and internationalization language resource files. A new software revision will also include updates to the application configuration.

---

### Scope of shared vs independent parameters

<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Created by</th>
<th>Factory A</th>
<th>Factory B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calibration/Comp</td>
<td>Maintenance</td>
<td>System 1</td>
<td>System 1</td>
</tr>
<tr>
<td>System Cfg</td>
<td>System integrator</td>
<td>System 2</td>
<td>System 2</td>
</tr>
<tr>
<td>Recipe(s)</td>
<td>Process engineer</td>
<td>System...</td>
<td>System...</td>
</tr>
<tr>
<td>Factory Cfg</td>
<td>Process manager</td>
<td>System 1</td>
<td>System 1</td>
</tr>
<tr>
<td>Application Cfg</td>
<td>System Developer</td>
<td>System...</td>
<td>System...</td>
</tr>
</tbody>
</table>

**Figure 2. Scope of shared processes parameters**

At the heart of a flexible framework is the ability to configure it for a variety of applications. The settings and configuration are stored in human readable xml/json file format. These parameter files can be changed at runtime to enable different behavior. A new installation of the application automatically creates valid default settings for configuration, calibration and compensation files. As the tool evolves through its lifecycle the framework supports
upgrades to the recipe/configuration/calibration files. Software upgrades add new parameters initialized to default values.

Software upgrades should never remove obsolete parameters. A shared parameter file that has an obsolete value on a system running a new version of software may be needed by another system which is running an older version. Even if the parameter file is not shared between systems, obsolete parameters would be needed if the software was rolled back to a previous version. When a parameter file is loaded, it is validated. A default value is used for missing, invalid or out of bounds parameters and the system supports the ability to reset parameters to a valid default value.

SYSTEM RESOURCES

A challenge for making reusable software components is their loose coupling to system resources. System resources include event notifications, logging, exception handling, reporting status/state changes and archive/restoration of parameters. The components must be integrated into the system resources but also self-contained to manage a subsystem (loosely coupled but highly coherent). These two design goals are often at odds with each other.

SIGMA DESIGN has solved these challenges in a variety of ways. The following sections describe in detail the framework system resource for events, logging, and façade. The façade aggregates all the individual components together so the user interface has a unified view of the system.

EVENT NOTIFICATION

Events are the basis for framework notifications and are used by logging, exception messages, user interface view, and trace utility. Events are used to report errors, save a historical record of systems behavior and display status updates to the user. Event consumers need detailed information captured about: 1) type of event, 2) context of where the event happened, 3) state of the system/values triggering event and 4) a grammatical description that clearly states the issue.

An event notification publish-subscribe design pattern is used to decouple event generation from the consumer of events. The event can be published without the tight coupling of needing to know who is listing. The event contains many attributes and these values are packaged into a JSON string for publishing as a notification or throwing as an exception. The various consumers of events can selectively subscribe for specific events or all events. An example of loose coupling would be a standalone diagnostic trace utility that can be dropped into the project. The trace utility would subscribe to debug event messages, keep them in a circular buffer and triggered to dump events to a trace file when an exception occurs.

It is the responsibility of each task to report significant events which include diagnostics, traces, warnings, errors and critical asserts. Event creation and publishing is distributed throughout the software code base but the developer will only liberally add the necessary event notifications if the event reporting mechanism is easy to use. An event system is worthless if it is difficult to add new message types into the repertoire or cumbersome to generate the event. Defining events globally or by local definitions are two approaches to implement events but both approaches have shortcomings:

- **Global Event Definition:** A common way to define events is a central global event table. This makes it easy to avoid duplication, make updates and define event responses but is a poor architecture because it tightly couples subsystem components to the central global event table. Reusing a component from a previous application requires plucking the events the component references from the old event table and inserting them into the new application’s global event table.

- **Local Event Definition:** Alternatively, event descriptions can be hardcoded at the point of event creation. The downside of having all events local is the lack of a complete event list, inability to localize strings, and events do not have a unique identifier.

SIGMA DESIGN’s framework provides a hybrid solution that defines events locally and compiles a global list of all events at system initialization. Each subsystem module defines a table of events as a class. During event class initialization the base class compiles a report into a central repository that lists all the system messages that a component can generate. The consolidated list facilitates the review of the event message presented to the user for clarity and usefulness. The event list can also be used to create a resource file of localization event strings index by the event unique ID identifier.
Since all the event consumers have the potential to display a message to the user, the event description must be in terms the user can understand, grammatical, and include relevant data about the event. A well composed event message allows the user to resolve or work around the issue without having to call customer service. The event must also include several attributes that capture the full context of the event. The context captured at the time of the event must provide as much information as possible for a developer to fix the root cause of programmatic errors. The framework event class also supports serialization of the event to/from a JSON string. The JSON string is a convenient format to pass the event across network boundaries or archiving the event. The following event fields/attributes are captured when a framework event is created:

- **Event Category**: Grouping events into categories such as error, warning, informational or debug provides a high level of understanding if corrective action is required or while filtering through a log of events to understand the system behavior.
- **Formatted Messages**: Event messages contain formatting that allows insertion of arguments. The manipulation of a format string with arguments can be error prone and must gracefully handle exceptions while recovering as much information as possible to report.
- **Unique ID**: A unique name or event number facilitates communication or provide a supervisory system with a key to lookup a corrective action.
- **Context**: The context of the event is important for a developer to understand the root cause. Languages such as C# provide a convenient compiler services retrieval of the event file, method and line number which generated the event. If an event is thrown as part of an exception the event has a field for capturing the call stack.
- **Event Data**: Event analysis is facilitated by storing the formatted arguments as individual fields in a log record. System performance analysis often requires mining the log files for logged parameters and analyzing trends in the data. It can be laborious to parse the log messages to extract the event parameters. By storing the formatted arguments as individual fields in a log record, it is easier to extract all temperature recording events from the log so a plot of temperature vs time can be generated.

**EVENT LOGGING**

The Logger is responsible for maintaining a historical database of significant system events. The logger is a critical resource and is much more than just translating a log ID into a string that is pushed into a file. The framework logger is an event driven, standalone module that captures events. The logger has a configurable filter to select which events are logged. An example would be enabling debug/trace messages during development but disabling for deployment. The logger can supports various options of archiving events into archives such as CSV file or SQLite database. The logger becomes a mission critical tool to maintain and diagnose the system after it has been deployed and it is important to give this level of importance during development.

**EXCEPTIONS & ASSERTIONS**

SIGMADESIGN’s framework uses a policy to report all errors as exceptions instead of return values because it is a common mistake for the application to ignore the errors returned by a function. It is far better to throw an exception and handle the exception than to allow the application to continue with undetermined results. How an exception is handled depends on the context of the application. A decision of the exception error handler could be to shut down the application. A graceful shutdown is far superior to a software crash caused by ignoring an error.

When an exception must be thrown the framework provides an exception class that is derived from the language’s exception base class. The framework’s exception class takes a JSON event string as its construction argument since a framework event captures the full context of an error. Ideally the exception is caught as the derived framework exception and has access to the full context and information about the event. Error handling can decide if the system can recover by itself or if user intervention is required. If the exception is caught as a standard exception the ‘message’ property contains the serialized JSON event string.

Exceptions can be divided into two broad categories, software defect or an application error. An event that is thrown because of a software defect is intended to gracefully handle the error and help the developer resolve the issue in a future release. These defects should be found in unit/integration testing and resolved before the release ships.
Assertions are used to check for anomalous conditions that should never happen in an execution. The framework provides a wrapper to test assertions. If the assertion fails the assert wrapper creates an assertion event, logs the event and throws an exception. Explicit error handling can then attempt to recover the system. The assertions should not be ignored, even in the final product because they also provide an important role to catch misbehaving code at the point of failure.

**FAÇADE DATABASE**

An important feature of the framework is the ability to combine various models/components together but these must be presented to the view in a consistent unified interface. The software façade pattern is used to combine the parameters and status of the aggregated models into a database that is symbolically accessed. The façade provides the ability for the view to retrieve the list of model names and their parameter/status names. The view then iterates through the list to retrieve the values and attributes. Attributes include items such as name, description, range limits, and type. The ability to access status and parameters via their name is a powerful tool. Not only can you iterate through the list of parameters to save and restore their value from a file, but it is also possible to automatically create a diagnostic interface to view all system statuses and set any parameter with bounds checking.

The façade loosely couples the view to the model. The MVC pattern uses events to notify the view of parameter/status changes. A publisher/subscriber pattern is used to enable the view to receive change notifications. A notification is fired if there is a change in a parameter or status value. The flexibility of this architecture allows a standalone transactional logger to register notifications and log them into a circular queue. The transactional log can also be dumped to file when triggered by a significant event such as an exception.

When a manager initializes, it registers its parameters and statuses into the database. The registration associates the parameter/status name with a delegate that can set or get the value. The registration also include the following attributes for each parameter and status: name, type (integer/boolean/float/etc), default value, description, format to convert to a string, mutual exclusion lock, and min/max range of a parameter. When the system is first initialized all parameters are set to default values.

The implementation of the façade uses a wrapper around the delegate for retrieving or setting a parameter or status. The delegate does the actual work calling into the driver to retrieve or set the value. The wrapper encapsulates the delegate with multithread protection, bounds checking and change notification. A locking mechanism provides mutual exclusion when threads are concurrently accessing parameters or status of a model. For example, when communicating over a serial link to retrieve a status or set a parameter, the request/response transaction must be completed before the next request/response by another thread can execute. The wrapper uses the following procedure for setting or acquisition:

- **Parameter Setting**: When a parameter is set directly by a model’s class property (or set indirectly via the façade) the façade parameter wrapper performs bounds checking, validates that the model is in a valid state to make the change, sets a mutual exclusion lock and then it calls the configured delegate to make the parameter update. If the parameter has changed from its previous value then a notification is broadcast to all subscribers.

- **Status Acquisition**: When a status is retrieved directly by a model’s class property (or retrieved indirectly via the façade) the façade status wrapper sets a mutual exclusion lock and then it calls the configured delegate to acquire a status value update. If the status value has changed then a notification is broadcast to all subscribers.

Parameters are flagged as either configuration or recipe parameters so it is possible to stream out just the configuration parameters and save them as a *json* file to create a default configuration. The same operation can be performed for recipe parameters to create a default recipe configuration. The recipe/configuration file includes parameter description, name, and range attributes to facilitate offline editing of the settings. When preparing the system to run the code to load or save a recipe/configuration is a standalone utility that does not need to know any intimate details about the system since the façade allows access by just the name of the parameter. Loading of a recipe/configuration looks up the parameter by name in the database to verify it is a valid parameter, the correct type and value is within range limits. The user is notified of any exception and also if any values are missing values from the recipe/config file.
As the software evolves through its life cycle, parameters will be added to the recipe and configuration. A software upgrade will automatically update the recipe/config files with missing parameters. When the system is initialized the database contains default values. Any parameter the config/recipe files is missing will remain set to its default value and then streaming the recipe or config back out to file will update any missing values with their default values. The policy of keeping obsolete recipe parameters persistent in the file supports backward compatibility and ability to share configuration/recipe files with systems running different version of software.

**SUMMARY**

There are many frameworks available depending on your application and the target hardware. The workcell framework developed by SIGMADESIGN provides the infrastructure, configurability and reduces time and cost by enabling the reuse of existing modules for a variety of applications deployed on a PC platform. The framework components can be reconfigured and interchanged because the architecture loosely couples them to the system resources, such as logging, reporting status/state changes and archive/restoration of parameters, while providing a robust and specialized solution to factory automation.