ABSTRACT

An architecture for building laboratory automation systems was developed using the component-based framework [1]. The requirements to build a flexible and adaptable automation system were exemplified through the architectural design process. Existing tools and technologies facilitated the development of this scalable architecture. Several design patterns were used to model this architecture. The influence of these patterns on systems design and the impact of Unified Modeling Language on component and interface design was explored. Integration of the system components with the peripheral components to build an automated system is discussed with emphasis on run-time configuration and component reuse.

INTRODUCTION

The technology and development process for laboratory automation systems has matured in the past five years and the focus has shifted to deploying both stand-alone workstations and large scale integrated systems that allow for flexible expansion and adaptation. An automation system that is designed to be reliable and robust not only contributes to an immediate need, but also adaptation. An automation system that is designed to be reliable and robust not only contributes to an immediate need, but also adaptation. An automation system that is designed to be reliable and robust not only contributes to an immediate need, but also adaptation. An automation system that is designed to be reliable and robust not only contributes to an immediate need, but also adaptation.

The traditional configuration mechanism for each individual component requires the user to specify parameters that describe how it behaves. Flexibility refers to the ability of the component to accommodate changes in configurable parameters either prior to, or during a run. Complexity refers to the number of parameters that a particular component needs, to accurately describe its behavior.

Configuration management of an integrated system deals with managing configurable data from both the system level and the component level. System level configurable data usually affects the behavior of the entire system, while configuration data from each component affects their respective run time behavior and performance. Configurable data from both sources has static and dynamic characteristics. Static configuration data describes the organization of the system and its individual components at startup. Changes made to the system or any component during run time create dynamic configuration data, which influences the tasks that a component performs until reconfiguration takes effect.

Configuration management provides an environment that allows a Central Configuration Manager to discover each integrated component’s configurable options. These options can be expressed as attributes such as configuration name, data types, acceptable ranges, etc. The Central Configuration Manager should have the ability to configure these options, which can be activated by the system user or component logic during the execution of respective laboratory unit operations (LUOs).

The traditional configuration mechanism for each individual component is addressed through a Graphical User Interface (GUI) provided by the component and supported by its corresponding initialization file. Configuration management in an integrated system is dynamic and requires each component to open its configuration channels, both GUI and programmatic, in accordance with systems. This paper outlines specific technical implications of this strategy to address the above requirement and a possible solution built using a component based-framework [1].

ARCHITECTURAL REQUIREMENTS

This section discusses several important requirements and the vision for developing large-scale automation systems. In order for automation systems to be flexible and adaptable, it is important that common facilities such as scheduling, event management and operations management are isolated from individual components and managed centrally. One of the fundamental features of the architectural requirements is the need for centralized facilities offering reusable services. All the requirements identified below need support for this feature to be scalable.

1. Configuration Management

An automation system contains components that control instruments, robots and other hardware. Some systems are small and contain a few components, while, others are large and contain a number of components from many vendors. Usually, the larger the number of components, the more complex a system is. Each component requires the user to specify parameters that describe how it behaves. Flexibility refers to the ability of the component to accommodate changes in configurable parameters either prior to, or during a run. Complexity refers to the number of parameters that a particular component needs, to accurately describe its behavior.

Configuration management of an integrated system deals with managing configurable data from both the system level and the component level. System level configurable data usually affects the behavior of the entire system, while configuration data from each component affects their respective run time behavior and performance. Configurable data from both sources has static and dynamic characteristics. Static configuration data describes the organization of the system and its individual components at startup. Changes made to the system or any component during run time create dynamic configuration data, which influences the tasks that a component performs until reconfiguration takes effect.

Configuration management provides an environment that allows a Central Configuration Manager to discover each integrated component’s configurable options. These options can be expressed as attributes such as configuration name, data types, acceptable ranges, etc. The Central Configuration Manager should have the ability to configure these options, which can be activated by the system user or component logic during the execution of respective laboratory unit operations (LUOs).

The traditional configuration mechanism for each individual component is addressed through a Graphical User Interface (GUI) provided by the component and supported by its corresponding initialization file. Configuration management in an integrated system is dynamic and requires each component to open its configuration channels, both GUI and programmatic, in accordance with systems.

This paper formed part of the Proceedings of the International Symposium on Laboratory Automation and Robotics (ISLAR), 1998. Reproduced with the permission of ISLAR (for further information, see: www.islar.com).
certain predefined protocols. It is feasible to allow each component to support a way in which it simply presents its configuration GUI when it receives some predefined calls. The approach frees the client from discovering the configuration options. This mechanism however, requires the user to interact with GUI; therefore it can not accommodate programmatic run time reconfiguration, and it is not feasible in a distributed environment. It can be used as optional configuration channel along with programmatic reconfiguration.

In addition to providing facilities to allow reconfiguration to occur, correctness and consistency issues need to be considered. When a component undergoes reconfiguration, there are changes in its state. Such changes in the state should not lead to changes in the semantics associated with the component before the reconfiguration point (RP), nor should it lead to an unexpected behavior after reconfiguration. The former requirement leads to correctness issues and the latter to consistency issues [2]. Architectural guidelines should require an individual component to enforce the correctness and consistency rules. If the above rules cannot be enforced during reconfiguration, the protocol should allow configurable options marked as “not configurable at run time.” This implies those new values for the configuration data will only become effective when the component starts the next time.

Configuration management also impacts system structure, which provides a mechanism to reassemble or construct a system from its components. The first goal of this architecture is to allow the system developer to “plug” in new components, or to discover and reorganize the components that can be “played.” Second, it should be possible to define relationships properly by dragging and dropping individual components into the system context. The configuration management facility can then persist the components and their respective system structure information in the system repository. Finally, the architecture should allow other facilities the ability to “play” this new component.

2. Event and Communication Management

All components in an automated system generate events. Events are occurrences such as the change of status in a component’s internal state, progress of an operation, detailed status of each operation, and status related information from the underlying hardware.

An event notification mechanism ensures cooperation between different components so that a system level coordination and management can be accomplished. Issues related to job synchronization, message acknowledgement and task notification fall under the purview of event and communication management. Event and communication management provides a mechanism by which a particular component recognizes that other components in the system are interested in internal events. This facility also provides a mechanism that establishes how many components are interested in a particular event and what mechanism should be used to notify them when that event occurs. The notification and response of timely events is important in ensuring accurate performance in real time system.

Additionally, synchronization patterns of concurrent processes during message passing is an issue of concern that must be addressed in the design of an automation system. When passing messages, good event and communication management should provide an effective model for different concurrent behavior, such as asynchronous and synchronous communication. The real-time characteristics of automation systems make it more critical that the underlying architecture supports effective synchronization mechanisms. The key characteristic for this type of event sharing and management mechanism is loose coupling between components by announcing and responding to events through well-defined and effective communication protocols. These protocols govern the rules, formats, and procedures agreed upon by all components wanting to communicate when certain events occur.

Another important aspect of event and communication management is a facility that utilizes the established protocols. This facility can be built either into each component or in a centralized location. Each component can carry a lightweight facility to support this type of messaging, which allows it to inform interested components when specific events occur. However, this type of arrangement does not scale well in an enterprise system. In addition, lightweight event and communication facility supported by each component leads to repeated implementation in each component. Large-scale systems require a central facility that manages event and communication from all other components.

Event notification mechanism provides a backbone for communication and event information sharing infrastructure that other management services component can use to function effectively. These components include scheduling, control management, monitor management, alert management, log management, etc. Relationship with other facilities, especially with the monitor management facility, requires the event management facility at the system level to support collecting, filtering, grouping, and routing events to a centralized resource like a hub. Policies can be defined and configured in other facilities like alert management, so that pre-configured significant events will trigger certain specific actions, such as alerting or automatic aborting. It is important to recognize that event management facility is not the place for implementing and carrying out alerting, controlling, logging, etc., instead it exists to manage and deliver pertinent and well-organized events that different services can utilize in the most appropriate way.

Therefore, at the component level, the architecture must ensure effective communication so that event information will be shared as needed. Event and communication management at this level allows all components in the system to have an effective mechanism to use for notification about the events they are interested in. At the system level, the architecture should ensure that it is easy to manage events in a way that other services can utilize them more effectively.

3. Scheduling

Scheduling is the process of selecting among alternatives and assigning resources and times for each task, so that the assignment obeys the temporal restrictions of activities and the capacity limitation of a set of shared resources. Scheduling is an optimization process where a limited number of resources such as a robot, and other peripheral instruments are allocated for both parallel and sequential LUOs.

In an automation system, the scheduler is responsible for executing the LUOs according to the run time specification. Designing and developing a good scheduling component requires familiarity with complex theory, the application of static, dynamic,
off-line and on-line scheduling and a set of rules: Jackson’s rule, Smith’s rule, McNaughton’s theorem, Liu and Layland’s rate-monotonic rule, Mok’s theorem and Richard’s anomalies [3]. This challenge is further escalated by the fact that many real-time systems that involve resource sharing are by nature NP-complete or NP-hard (NP is the class of all decision problems that can be solved in polynomial time by a nondeterministic machine).

Static scheduling makes its decisions before the system is run; it usually generates a dispatch table type information and optionally saves it into a persistent storage facility. At run time, it relies on the pre-constructed task table to run the system. Since it needs to generate the most optimized feasible schedule of LUOs, it needs detailed information about these LUOs in advance, for example execution time, resource constraints, deadlines, etc. A static scheduler uses this information to construct a feasible schedule by searching a list of LUOs with the support of appropriate heuristics strategies. Dynamic scheduling, on the other hand, delays its decisions until run time, when new tasks arrive at random. It relies on the parameters and constrains associated with the new arriving task to continually make scheduling and synchronization decisions.

Both static scheduling and dynamic scheduling algorithms have their advantages and disadvantages and therefore their applicability also differs. Static scheduling problems have been extensively researched and is supported with classical scheduling theory. Good predictability of the system behavior also makes static scheduling a practical choice for many automation systems. Dynamic scheduling is a better alternative when real-time automation is implemented in a dynamic environment, for example when a new higher priority task needs to be accommodated. However, dynamic scheduling requires greater computing capacity to address real time iterative scheduling. Tradeoffs have to be considered when making a choice between static and dynamic scheduling, such as predictability of the task being scheduled, complexity of the tasks being scheduled, efficiency of the process (scheduling algorithm), throughput of the process, and adherence to the constraints specified. It is not often true that a single scheduling algorithm applies to all the different applications in laboratory automation domain. The architecture should allow the user to select from a list of available scheduling algorithms, otherwise the process being scheduled needs to be adapted to the algorithm, provided the modifications satisfy the scientific and business constraints. In addition, the architecture should accommodate the ability to evaluate different scheduling algorithms off-line so that a user may make the best use of the scheduler. The scheduling component and its executive work with other components in the system to recognize resources and operations supported by peripheral components and other centralized management services. They rely on facilities, such as control, event, and communication management, to execute the schedule. Though strong collaboration must exist between the scheduling component and other central management facilities, it is important to ensure loose coupling between the scheduler and other components in order to facilitate independent growth/evolution of respective technologies.

4. Control and Operations Management

Control and Operations Management deals with the control of all the components that are a part of an integrated system. For components that control hardware, this mechanism is ultimately the channel that drives the functionality of instruments and motion of the robots.

Control requests may come from various components, such as scheduling components, monitoring components, or simply as a result of the user’s interaction through GUI. Control management requires that each component publish its LUOs and its associated parameters. Mechanisms should also be set up so that once an operation is started, it can be paused, re-activated, or stopped if necessary in a manner consistent throughout the system.

Even though each component supports different kinds of operations, they possess many common properties, which describe an operation, e.g., name, parameters, input/output ports, resources involved, expected execution time, and the final destination of resultant data as applicable. This commonality helps recognize the key abstractions and mechanisms involved. Once a common way of modeling and controlling operations independent of the underlying functionality is established, separating component functionality from the interaction mechanism between the component and its client can be easily addressed.

5. Monitor Management

Monitoring is the process of getting the status of systems and instruments through software sensors. Users and administrators are interested in the overall system status and progress, and in the status of specific instruments. Central system facilities are also needed to monitor each component’s status in order to make appropriate run time decisions.

Monitoring at the system level requires that users have the ability to determine the exact state of the system. Monitoring mechanisms should support services that allow users to monitor the system either locally or remotely. Further, this mechanism should allow a user to monitor several systems at once in real-time. In an integrated automation system, each component contains internal software sensors to help monitor the underlying functionality. In order to provide efficient system level monitoring, it is important that individual components adhere to established guidelines for monitor management, e.g., publish monitor information and its associated parameters according to the set standard. In addition, the architecture should support a system level facility to define monitor management policies that govern the behavior of the system at run time.

Different roles among the users and administrator of a system usually result in different needs for monitoring and controlling the system. When remote management issues are introduced into the system, security and safety issues need to be addressed. Remote management involves remote monitoring and remote control. Remote monitoring involves no safety and minimal security issues compared to remote control and hence should be addressed first.

6. Alert Management

Alert management provides the functionality to notify users and administrators of the system about significant events based on predefined event policies, using different communication mechanisms. Alert management differs from event management in the sense that alerts deal only with significant events in which the user needs to be involved, e.g., a system error. An alert is accompanied by information regarding the event it is associated with.
The alert management facility is responsible for collecting user's input regarding the configuration data for alert management and then persisting it in the system repository. Alert management must present a set of available events to the user. Alertable events are a subset of all events generated by an automation system. They are categorized with several criteria, e.g., the severity level of the alertable event. Alert management facility should be aware of the attributes of different users of a system. It should extract information about users and groups from the system user repository, so that the mechanism of an alert can be configured for individual recipients. Also, alert management should support different kinds of communication, e.g., GUI based notification, electronic mail, pager, etc. The facility responsible for alert management should work with other facilities, such as event and communication management, to collect and filter significant events that users want to be alerted with. In addition, working with the logging facility, alert management should also be able to also send a copy of alert information to the logger for audit, and other facilities to trigger some predefined actions, e.g., decision support systems.

7. Log Management

Logging is an intricate part of any laboratory operation used for many different applications. A log has many uses, including trouble shooting and optimization. Information in the log is responsible for generating a categorized audit trail. The log manager should support the ability to extract types of logging services from a peripheral component through a standard interface and configure them based upon the nature of the run and the options chosen by the user. Two aspects of log management demand attention, the ability to log events and the location of the log. The location of log can involve local component specific repositories, a centralized system repository, or even a Universal Resource Locator (URL).

8. Information Management

This architectural requirement addresses two major needs, data management and report management. A laboratory automation system contains many components that generate data. A common mechanism to extract this data from individual components in realtime and channel it to the appropriate decision making process needs to be addressed. Report management addresses the presentation of the data according to specific requirements set by the user. Reports can be broadly classified into the following categories, component specific reports and system wide reports. Further, reports can also be classified by the format they are generated in, i.e., reports with predefined format and reports with user configurable formats. Component level reports support both standard formats (i.e., error, diagnostics, etc.) or specific formats. Configurable reports require that the report manager must have the ability to extract the subject and content of a “plugged in” component in order to report the information to a perspective user. It is important to integrate all physical systems with the back-end support of multiple databases such as those supported by Laboratory Information Management Systems, GroupWare Server, and the world-wide-web.

9. Security Management

Complex automation systems always involve many users with different roles, such as different level of end-users, methods developers, and system administrators. Complexity and responsibility of security management increases with the functionality and complexity of the automated system.

Security management from external software packages such as an operating system or a network system require users to remember different names and passwords. Flexible and adaptable automation systems need the security management facility to be merged with the user database from either the local operating system or the network system. In addition, they require the access management to systems resources in a centralized manner.

10. Intranet Enabled

Centralized system services such as monitoring and administration lend themselves well to Intranet technologies. In addition, Intranet technology can be used effectively for information management. Through careful configuration of Monitor Manager and Alert Manager, users should have the ability to view the progress and the status of an automated system via this technology. The ability of system architecture to provide centralized services such as monitoring system status, alerting users to specific events and controlling the operation of automation systems via the world-wide-web is an important architectural requirement in order to build flexible, adaptable and scalable systems.

11. Human Communication Interface

Human communication interface is an important aspect of the development of an automation system. It directly addresses the usability goal of an automation system. For most users, human communication interface is the environment that allows for interaction with the system. Although the user interface part of an automation system has evolved from the MS-DOS style to GUI, better user interfaces are continually needed. Good user interface design guidelines such as directness, consistency, feedback, aesthetics and simplicity should be considered when developing human communication interfaces. In addition, emerging user interface technology such as three dimensional interfaces and web browser based interfaces should be considered to provide a more realistic view of an automation system.

12. Availability

The goal of an availability strategy during the design of a system is to ensure minimum downtime of the system once in production. The reasons for failure could originate either in the software or in the hardware. Usually, the number of failure points is a function of complexity of the system. The implication of this requirement is to minimize the number of failure points in the system. It also requires that hardware and software components be fault tolerant. One way to increase the availability of a system is to accommodate several run modes based upon use-case modeling of the process being automated. This implies that the system configuration utility should detect offline components and reconfigure the system without them. It also infers that the system configuration facility should accept a request to take a malfunctioning component offline. Tools like resource matrix vs. functionality assessment provide practical implications availability strategy. One of the consequences of a good availability strategy is to support the operation of an integrated system in both workstation and integrated mode.
TECHNOLOGY AND SOLUTION

In the last section, some important design goals for a component-based framework were discussed. This section discusses the state-of-the-art in software engineering and the tools that are currently available that aid in the development of the architecture for the requirements identified above.

1. Object Technology

Object technology provides powerful problem domain abstraction compared to traditional structured methods. In the development of laboratory automation, software developers have used object-oriented approach to model and develop laboratory systems [4,5]. Although object technology provides tangible benefits in systems development, it is necessary to move beyond simple object-oriented programming concepts, which focus primarily on individual objects and classes; and therefore are restricted to the scope of the software at a component level. In other words, the complexity of laboratory automation systems can not merely be addressed by a few well-engineered object-oriented classes whose scope is limited to controlling one or two instruments. In order to address the development of complex systems at the architectural level, it is important to move beyond object technology to something that provides more abstract modeling capabilities (e.g., component based framework strategies [1]).

2. Component

Software component technology promotes higher level of reuse than simple objects. A component is a static abstraction with plugs [1]. ‘Plugs’ refer to the in and the outgoing interfaces by which components can communicate with each other regardless of their internal implementation. Binary interfaces allow a component to make its services available to outside clients across regardless of the language the client is written in and the platform the client is running on. The independent deployment and composition property of component technology provides the ability for components to work alone and, at the same time, participate in an environment that requires them to work with other components to provide complete functionality. A component-based approach is the fundamental strategy for building a framework for laboratory automation systems.

3. Framework

Component Frameworks have received much attention recently in contemporary software development. A framework is a collection of classes or components that provides a set of services for a particular domain; a framework thus exports a number of individual classes and mechanisms, which a client can use or adapt.

A complete version of a framework for a complex domain such as laboratory automation is difficult to design because of the range of applications the framework is expected to support (in that domain). The goal of an ‘application-independent and domain specific framework’ provides a unique opportunity and challenge for systems developers in the area of laboratory automation. The Hollywood principle, “Don’t call us, we’ll call you,” characterizes one of the biggest differences of the framework vs. class library approach. The implication is that a framework controls the component flow by providing a collection of important services that will automatically support and manage various components that will be “plugged in” into the framework in the future. It is important to recognize that the framework should be well defined to support new components and the components should honor the specifications and constraints set by the framework. As a result, a framework needs to be extensible and flexible, otherwise certain applications might find it difficult to utilize the framework due to its constraints. The evolution strategy of both the frameworks and its components need to be carefully considered and designed.

4. Component and Object Cohesion

Two main camps in software industry have provided wiring standards and common infrastructure to enable the interoperability between objects and components regardless of where they are located in a distributed environment. The first organization, Object Management Group (OMG) supports Common Object Request Broker Architecture (CORBA) as the basic architecture level to support interoperability, and Object Management Architecture (OMA) for higher level application, communication and other services. The second set of standards is COM/DCOM (Component Object Model/Distributed Component Object Model) and their related services supported by Microsoft Corporation. (More recently, a new set of standards for components developed with Java, “JavaBeans” and their distribution model have emerged with support from SUN Microsystems.) Issues for interoperability between the components built according to these standards are also being addressed by the organizations involved and new products for bridging components based on these two standards are starting to appear in the market.

The two component based approaches, i.e., COM/DCOM and CORBA/OMA, deliver similar solutions with common features such as binary interoperability, interface definition language support, multiple interface support, and common services facilities etc. Emergence of these two standards and their related services provide an opportunity for systems developers to build better systems with object and component technology. Any component that is developed to be a part of an integrated solution has to consider support for at least one of these two standards in order to work seamlessly with other components both in small and in large scale environments. Both COM/DCOM and CORBA/OMA already support a wide variety of communication protocols for distributed inter-process communication. This, coupled with the recent stabilization of these two technologies, suggests that common communication protocols (interfaces) between components for any domain can easily be standardized on one of these two standards. The object and component bus provided by COM/DCOM and CORBA/OMA should be used to implement the backbone for ‘plugging in’ components into an environment like a framework.

The component-based framework and systems discussed in this paper is based on COM/DCOM technologies. The use of COM as opposed to the other distributed object model is mainly based on internal environment (predominantly Microsoft Technologies) for both laboratory automation and information management applications. This approach is further justified by the availability of industry tested development tools, and, the fact that most
major laboratory automation vendors provide better support for the Windows® or Windows NT® environment.

5. Unified Modeling Language:

Unified Modeling Language (UML) is an object modeling technique developed by the three object-oriented gurus, Grady Booch, Jim Rumbaugh, and Ivar Jacobson as a consequence of consolidation of their individual object-oriented methodologies. UML represents the best concepts from each of their original methods. As a result, UML provides effective notation and a meta-model for designing and developing complex enterprise scale systems. UML is not only a good tool to assist developers to analyze the problem domain and design architecture for a system; it also provides a common ground for developers to communicate their ideas, and to document the results of their analysis and design. It is anticipated that UML will eventually become an industry standard for systems modeling. Several applications of UML in the development of real-time systems are available in the literature [6]. UML provides notations and diagrams such as use cases, classes and their relationship diagram, sequence diagrams, collaboration diagrams etc. UML also provides very useful notation for message synchronization and state diagram representation for real-time systems. Additionally, it supports component diagram and deployment diagram used in modeling and documenting systems architecture and physical deployment. It is important to recognize that UML focuses on notation at this time, the developers of UML are in the process of finalizing the application of UML to systems development life cycle.

6. Software Design Patterns:

Design patterns by nature promote reuse, not only at the individual class level, but also at higher levels, since they capture the core solution - good design or architecture, for a problem that might re-occur in a certain environment. They focus on more than just one or two objects or classes; rather their responsibilities, their structure relationships and their collaboration model. Good design patterns are documented with a description of the problem they address, the successful solution, and the consequence which reveals results and trade off when applying the pattern in different environments. Leveraging good design patterns allows system developers to focus more on the goal of their work, i.e., to deliver solutions, by avoiding unnecessary re-invent-the wheel type of work when solving common software problems. Use of design patterns is an important technique that helps overcome the complexity of a system. In the development of the component-based framework, several proven design patterns were used. The development process used existing design patterns to address the architectural vision outlined earlier, while, at the same time, tried to uncover new patterns for the laboratory automation domain. The discussion below exemplifies existing patterns that were used in the development of the component-based framework.

Observer pattern [7], which is also called the Publish-Subscribe pattern, defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

This pattern tries to address following situations:

1. When a change to one object requires changing others, or other objects are interested in a change that occurs in that object and it is not clear how many objects are interested in the change.

2. When an object should notify other objects without making assumptions about what the other objects are. In other words, all objects are not tightly coupled.

The following diagram illustrates a use case for the observer pattern. In this diagram there is one subject and two observers.

![Figure 1.0 Use case diagram for observer pattern](image)

The solution for the use case depicted in figure 1.0 and its structure can be expressed in the following diagram:

The core model shows the observer registers with the subject with regard to its interest in the change of the subject; for example a significant event of an instrument as it relates to the laboratory automation domain. Registration is achieved through the interfaces provided by the subject. The subject maintains a list of its observers, such as storing their references and their interest. When the subject changes, it notifies each observer who has registered its interest in that change. Notification is done through callback to the observers’ notification function.

This pattern was modified and used in several areas of the development process. For example, this pattern was used in the design of many components to de-couple the internal objects, (such as those that control the instruments, which assume a subject role with respect to this pattern), from their views e.g. user interfaces (observer). This allows freeing the internal objects from worrying about user presentation while enabling the presenting
mechanism with different views. This way of using the observer pattern is similar to the famous MVC (Model/View/Controller) model from the Smalltalk [7] community. The observer pattern was also used to model lightweight event notification mechanisms that each component implements in order to allow their observers to receive information about significant events that occur in that component. Observers are central environment facilities, other centralized services components, and peer components. The component that plays the subject role does not care who the observer is, since the pattern successfully decouples them.

In addition to the observer pattern, the following patterns were also used in the development of the component-based framework. The Façade pattern [7] was used to model the connector for accessing Laboratory Information Management System (LIMS). This pattern helped define a unified interface to a set of interfaces in the sub-components that work together to link an application to the LIMS. Strategy pattern [7] is applicable in modeling scenarios where a family of algorithms (such as scheduling and diluting) needs to be evaluated for different scenarios (prior to selecting an optimal solution). Another interesting pattern is applicable to the object persistence mechanism. There exists a mismatch between object technology and object persistence mechanism when the persistence format is a relational database. A modified lightweight version of the CORBA object persistence pattern was used to model a COM component.

7. Framework vs. Pattern

Several major differences exist between design pattern and framework concepts [7,8]. Design patterns are small architectural elements compared to frameworks; design patterns are more abstract than frameworks; and design patterns are less specialized than frameworks, which are usually associated with a particular domain. Across a particular domain, products are architecturally very different, however, within each vertical application domain, there are clear patterns of idioms and mechanisms that serve to simply model a problem at hand. One of the important strategic objectives a framework designer needs to address is the collaboration of objects and components. As a result, a number of important design patterns of collaborations usually exist within a framework. Overall, design patterns support the process of creating a framework, and the process of creation provides opportunities to uncover new patterns for later use.

MODELING THE ARCHITECTURE:

An effort to model the architecture for automation systems based upon the component-based framework was initiated. Several common service facilities have been identified, modeled and are under development. These include the configuration manager, system repository, scheduler, logger, alerting engine and manager and a central event management facility. Peripheral Components have also been modeled in order to ensure 'plug and play' across the system. The following discussion includes specific components, their interfaces, and their relationships and interaction mechanisms between them. Further, the assembly of these components into an automated system is discussed.

Configuration manager and system repository are the components that provide services for configuration management. These components operate on the assumption that peripheral components plugged into the systems will support a set of standard interfaces. These interfaces make the component configurable and capable of assembly (system structure configuration). The central configuration facilities include an outlet that is built for other components to be "plugged in" later. Since the framework type architecture enforces each component as an element to support a set of required standard interfaces, which ultimately ensure the
plugability of components, several possible ways to plug new components into the system can be envisioned.

The configuration manager can be used to register or "plug" new components into system; the configuration manager works with component and the system repository service to accomplish the "plugging" task. At run time, other central service components, such as the scheduler, will be able to "play" that component through the interface implemented by the component. Alternately, an automatic discovery process can be used. Supporting a set of common interfaces in each component makes it possible for the central configuration manager to automatically recognize new components when they are installed into the same computer. This can happen by asking the configuration manager to discover components that support a specific set of interfaces, such as checking the operating system registry information under Windows NT™. The configuration manager collects information on these components and persists it into the system repository. The discovery process can be manually activated after the system developer installs a new component or runs periodically to update/maintain the accuracy of component library in the system repository. In reality, one only needs to activate this process when reconfiguring the system, unless the system is used to perform centralized monitoring for all systems. With the second approach, the scope of the system needs to be addressed to avoid unnecessary overhead during reconfiguration. Automatic discovery not only leads to populating the system repository, but also results in building a graphical view of the system and its components. Consequently, click management mechanism to be activated as a part of the user services for an automated system.

One of the fundamental requirement of this architecture is that the system components like the scheduler control different peripheral components in a consistent manner. The scheduler, for example, only views these components as resources that support a set of LUOs that can be activated. For other central management facilities, these components are manageable and controllable identities. Interfaces that make the assumption of the functionality can still be engineered with good object-oriented design and interface development concepts. Instrument specific interfaces provide direct access with lower overhead. This concept is useful in many occasions where a client is built and modeled after a component's specific interface. However, it usually requires special outlets on the client side in order to "plug in." It is possible to catalogue all instruments supporting similar functionality into a single outlet, however, this might be an impractical approach for many vendors and to accommodate new functionality into an existing component. As a result, in an integrated environment supporting a framework (where many components need to be plugged into predefined outlets) these non-standard interfaces should only be used sparingly to quickly prototype functionality. Once a standard interface is modeled and designed, it serves as a complement for the outlets implemented in the central facilities. Peripheral components, including completely new components with some highly advanced operations, can then easily be plugged into the architecture. Central components can then "play" their functionality through the standard interfaces regardless of how different the components supporting these standard interface are. The standard interfaces were modeled to accommodate all aspects of interface responsibility with respect to the interaction mechanism that resides in the peripheral component side. The interface design was consistent with the Microsoft Component Object Model [9].

For control and operations, the standard interface lets individual components publish their respective LUOs and their associated parameters. It allows the operations (LUOs) to be activated along with their respective parameters. The standard interface also provides a consistent way to pause, stop, and reactivate these operations. In addition, the standard interface allows each component to publish its logging services. A client can activate some or all of these services with an option to route the logging data to a different destination, if desired, including a centralized logger. For configuration management, the standard interface allows the component to publish a set of configuration options and their associated properties. A client can use this interface to set/change these values. For events and communications, the standard interface allows a set of clients to register their interest with a component. The component persists each client's reference for later callback with information when an interested event occurs. For callback, the component design makes an assumption of a predefined callback interface on the client. If the callback fails, the component broadcasts a message throughout the system. Although the design of standard interface allows a component to inform a number of clients simultaneously, in enterprise scale automation systems centralized management facilities allow for better scalability and design through lightweight facilities on the individual components.

Event and communication management for the peripheral components is addressed by interfaces that were designed to share information about events that occur inside the component. A centralized event and communication facility along with peer components can use this interface to register and receive notification of the events they interested in.

Interface design assumes that once a client, such as a centralized manager, (e.g., the scheduler), completes negotiation with a component and both components agree to carry out an operation, (e.g., logging a request, configuration request, etc), it is the component's responsibility to honor these requests. These commitments include performing the operations according to specifications, logging events to the correct destination and persisting new configuration data and changing its state if requested at run-time. Any errors encountered after committing to the request can be relayed to the event notification facility or an alternate error recovery mechanism.

Each component is also expected to inform the client of its ability to support simulation. Off-line simulation is useful when integrating an automated systems since it provides system developers with the ability to perform testing of run-time interactions between components without the underlying hardware.

Centralized event and communication management facilities were modeled with the specific purpose of meeting the system level event and communication management. A generic framework for event-based integration (EBI) from Daniel J. Barret at the University of Massachusetts [10] was used to help model this interaction. This framework is neither an implementation guide, nor a deliverable integration framework. Rather EBI is a concept, a high-level, general and flexible reference model that helps developers identify common components at the center of event-based software integration and serves as a basis for comparison of different event-based integration mechanisms. Although event and communication management is not unique, and similar situations exist in related areas such as telecommunication, networking and real-time systems, there are semantic differences in the way these inter-
actions are described in the respective areas. Hence, a formal model, like the EBI, helps system developers analyze and design central management facilities by abstracting common issues.

The EBI model is depicted in **figure 1.2**. It describes the relationships among the informer, listener, register, router, message transforming functions (MTF), and delivery constrains (DCs) [10]. Participant interaction is supported for all of the framework components: Registers, routers, message transforming functions and delivery constraints. Before a participant can transmit or receive any messages, it must register its intent to do so via a registrar. Using information from the registrar, a router delivers messages from informers to their respective internal listeners. EBI appears similar to the observer pattern described earlier; however, it addresses the interaction mechanism a much higher level. Since EBI it is not an implementation guide, it was used to model event management in the development of the component-based framework. Other centralized service components such as log management, monitor management, scheduling and security facilities are also being modeled and are under development.

**DISCUSSION AND CONCLUSION**

This paper introduces some important issues that impact the development of enterprise scale laboratory automation systems. Issues like scalability and the factors that contribute to the design of scalable systems were addressed in detail. The process of modeling the architecture was described. Systems developers in the laboratory automation community have modeled this domain in a generic manner for quite some time. For example, Standard Laboratory Module [11] was one of the earliest concepts that recognized this need. Modeling modular architecture in this domain was also addressed [12]. In addition, many automation vendors have also started to develop better solutions to support complex laboratory systems. It is important to continue making advances in modeling and developing better frameworks in this domain, so that we may address the need to build flexible and adaptable systems that contribute to reducing the drug development timeline.

More recently, advances in domain modeling, software engineering technology and other techniques, such as design patterns discussed earlier, coupled with the functionality of better underlying “plumbing” standards and services such as COM/DCOM and CORBA, have provided opportunities and challenges to systems developers to design and build better automation systems for the enterprise. In addition to providing plumbing standards for object and components to interoperate regardless of their location and environment, both COM/DCOM and CORBA [13,14] support horizontal common facilities. Simultaneously, the organizations promoting these technologies have created vertical domain facilities and standards to provide common infrastructure for the vertical industry. There exist two areas that are related to the laboratory automation domain; first is OPC (OLE for Process Control Specification), which uses the COM/DCOM technology for process control applications; the second uses CORBA technology with real-time extensions that continue to make progress.
However, our research hasn’t found direct CORBA extensions for laboratory automaton or process control.

The process of modeling an architecture based upon the component-based framework produced a prototype automation system that is continually being refined. In order to promote interoperability among various vendor specific technologies, the need for an industry standard communication mechanism becomes imperative. There are some initiatives underway to address this issue. The architectural details of this prototype will be discussed in a future publication.

ACKNOWLEDGMENTS

The authors would like to acknowledge the support of Dr. Edward McNiff, Patricia Ziemba, Dr. Marshall Palmoski, Cheol-Min Hwang, Benjamin Chen, Lakshmi Anumolu, Vidyut Tarkas and Padmanabhan Raman for their involvement in this effort.

Windows NT is a registered trademark of Microsoft Corporation.

REFERENCES


ADDITIONAL REFERENCES


Copyright© 1998, Bristol-Myers Squibb Company Page