INTRODUCTION

Designing, developing and implementing laboratory automation systems is a complex task. It takes careful planning and a disciplined approach to implement the functionality required at the beginning of the development cycle, while still allowing for additions/modifications to the system’s functionality without major reconfiguration. In today’s world of increased pressure for shortened development timelines, it is important to build systems that are flexible and can easily be reconfigured to extend their functionality. Traditionally, automated systems are developed for target applications. The hardware and the software of a system is purchased and configured with a particular application in mind. However, as time passes and the system’s utility is evaluated, it is very soon realized that additional functionality can be extracted from an existing system. The ease with which the existing system’s functionality is extended depends largely upon the two following factors:

- Coupling between module-level software and the top-level application software
- Ability of the application level software to adapt new hardware additions or enhanced functionality

Coupling between the module-level and the application level software is an important aspect when considering extending an existing system’s functionality. There are many systems that are currently functional whose application software is tightly coupled with the module-level software; i.e., the application level software is aware of the commands a particular module understands and accepts. In order to extend the functionality of the system, via the addition of new modules, it becomes necessary to redesign/re-engineer the application level software. Depending upon the system’s regulatory environment, this may or may not be an issue. Application level software can be continually refined in systems that are not required to be regulatory compliant, however, any such changes in systems that require compliance warrant change control procedures, and perhaps extensive re-validation. Loose coupling between the application software and the module level software is highly desirable. In other words, it is important to have the ability to make changes to module-level software to add additional functionality without any major changes to the application-level software.

Another major aspect of extending the functionality of an existing system is for the application software to be flexible to accommodate changes in the system configuration. Many systems are built to perform a specific task. Changes to the originally conceived application can require extensive reconfiguration if newly added hardware/modules do not make themselves “known” to the application software. This is the case unless the application software is specifically designed to seek out these additions. More advanced systems store the system configuration in a repository. This system configuration is read from the repository each time the application’s software is invoked.

In addition to the two factors identified above, there are several other architectural requirements that make an automation system more flexible and adaptable. Coupling and adaptability affect the fundamental manner in which automation systems are designed, developed and implemented and impact all the issues identified below.

Many existing automation systems revolve around the robot, i.e., the mechanical manipulator. While it is important to recognize the contributions of the manipulator, the system needs to be more application oriented. The mechanical manipulator should be treated as any other module by the architecture. Further, modules should be treated as objects providing services to the application. In many of the existing systems, although hardware provides certain services, close coupling between the application-level software and the module-level software makes this distinction diffi-
cult. Systems developed according to the traditional methodology have monolithic architecture and are very tightly coupled. This makes it difficult to view modules as objects providing specific services. Objects should be able to provide services independent of the underlying hardware and this forms the basis for a services based architecture for the design and development of any automation system.

In order to facilitate communication between independent objects, a standard mechanism for communication needs to be established. The communication mechanism in many traditional systems is unidirectional. The application software initiates a service on the device object and queries the device object periodically to establish the status. A more robust acknowledgement based messaging architecture is required where communication is bi-directional and each object acknowledges the receipt of the intended message with appropriate feedback response and control.

Automation systems built according to the traditional methodology depend heavily on vendor-specific architecture. It is important to note that not all hardware on a particular system comes from a single hardware vendor. Most hardware communication has some level of standardization at the micro level in the instrumentation industry, e.g., RS-232, RS-488 etc. The architecture for building flexible and adaptable systems must transcend individual vendor specific architectures and provide a macro level of standardization between hardware manufactured by different vendors.

In systems built according to the traditional methodology, a particular piece of malfunctioning hardware brings the entire system to a halt. Consequently, the system has to be shut down and restarted without the malfunctioning hardware. Fault-tolerance mechanisms allow for a malfunctioning piece of hardware to be taken off-line in a “live” system and substituted with another piece of hardware with equivalent functionality. Fault-tolerant systems achieve this via run-time configuration. The application software requests services of a particular device object only when it needs it. Fault-tolerance mechanisms allow the application software to terminate connection with any module-level object and re-establish the connection when needed. This loose coupling between the application-level and the module-level software provides for a higher degree of assurance that the automated system will not completely fail, if a particular piece of hardware malfunctions.

In order to support fault tolerance, the objects providing services need to support a mechanism where the state (an object's working data) of a particular object is set by the client of that object. In many systems that are built according to the traditional methodology, each object is expected to maintain its state and provide services when requested by the client. If the underlying hardware malfunctions, the device is shut-down, thereby loosing its state. Fault tolerance is achieved by designing stateless objects, i.e., objects whose state is controlled by the client requesting its services. In this case, reinitializing the object does not effect its state (since the client provides the working data) and it continues to provide its services independently.

Decision making from the data generated by a laboratory automation system is usually not an integrated part of the system and is often addressed off-line. Facilitating technologies like groupware are treated separate from the core automation system. It is important to recognize the ability of the decision making body, either scientific staff or a rule-based decision making object, to have integrated access to the data generated by the automation system in a timely manner. Integration of these technologies is also an area that needs attention as a part of the development of automated systems for the laboratory.

Access to automation systems demands close attention in a regulatory environment. Different levels of users require different levels of security and access. A central mechanism to configure and control access to the system is an important consideration in design and development of automation systems.

Architectural and design issues such as those identified above impact the ability of a system to extend itself and adapt to new chemical/biochemical techniques. In order to develop systems that are flexible and adaptable, a component-based framework is being developed that is based on existing strategies for building flexible, scalable and adaptable systems in related areas such as network engineering, manufacturing technology, database design, etc. [1]. This approach, along with the benefits of using a component-based framework for the design and development of laboratory automation systems is discussed in this paper.

**OVERVIEW OF THE COMPONENT BASED FRAMEWORK**

In order to address the architectural requirements identified above, a robust and reusable framework is being modeled upon a services based architecture that defines specific protocols for building components of a laboratory automation system. The fundamental requirement for building a flexible and adaptable system is to make the framework support scalability of the constituent components [1]. Scalability is achieved in this framework by the following two fundamental mechanisms:

- Centralized facilities that offer the common services for all components to use, e.g., Event and Communication Management, Alert/Monitor Management, Scheduling, Log Management, etc. [1]

- Well-defined and documented guidelines for the development of the constituent components that provide the functionality of the individual hardware of an automated system [1]

Components developed according to the established guidelines reuse the services provided by the central facilities. These centralized facilities are designed to support standard communication mechanisms, fault tolerance and run time configuration and stateless components thereby making the resulting automation system flexible and adaptable.

This framework is based on three fundamental activities: behavior, identity and realization [2]. Behavior refers to the ability of an object to support a well-defined behavior and to provide an unambiguous interface to it. Identity refers to the ability of the object to maintain the characteristics of a particular entity and to provide a convenient reference for that entity to any user. Realization refers to the ability of the object to realize its behavior and maintain its identity in real world implementation. Each component developed based on this framework supports all three activities. Together, these activities allow for the integration of these components into an application that is built upon well-defined interfaces.

Components are stand-alone objects that can be plug and play across applications, networks, language tools and operating systems. A component is a static abstraction with plugs [3]. Static refers to the component as a long-lived entity that can be stored in
A software base, independently of the applications in which it has been used. Abstraction refers to the encapsulation of the business logic component offers. Plugs refer to well-defined interfaces through which a component offers its services.

A basic feature of a component is a clear separation between its interface and its implementation. Also, components provide services that guarantee the interoperability with other components and facilitate the assembly of components in a heterogeneous hardware/software environment. In addition, a component’s implementation is mature with respect to quality, robustness, performance and documentation. Finally, a component offers a defined set of coherent functions that provide an effective solution to a well-defined problem [2,3].

The framework defines two distinct classes of components: System Components are those that provide central facilities that are reused by the Peripheral Components, which support the functionality of the individual hardware of an automated system. Both classes of components implement complementary interfaces that allow them to communicate and use the services published by each other. The communication mechanism supported by the framework is based on the functionality of individual components, their ability to have their published services scheduled by the system components and addresses the ability of the peripheral components to raise errors and provide audit facilities through defined interfaces.

This framework also addresses the assembly of the peripheral components into an automated system. Peripheral components designed and developed according to established guidelines are added to the framework and seamlessly integrate with the system components. The process of this assembly and the mechanism by which the system components establish and use the services supported by the peripheral components is documented as a part of the framework.

The fundamental feature of this component-based framework is its dependence on a services-based architecture. Services based architecture is a services-based view of the system. Services, which encapsulate the functionality of components, are accessed through a consistent interface that encapsulates the component’s implementation. Each service has a published specification of the face and its behavior. Constituent components (both system and peripheral) provide interfaces. Services provide for a macro level of reuse based upon software structures that transcend vendor specific development methodologies. This architecture provides a design philosophy for the reuse of components at the high end of the process, i.e., at the application level. It defines the constraints that allow for reuse of existing components that adhere to established development guidelines. It also helps capitalize existing services into new offerings or products.

The design of this framework is based upon the principles of Object-Oriented Design. A laboratory automation system is composed of various objects that provide specific services to produce the system’s functionality. Each object encapsulates a specific piece of hardware/software exporting a specific functionality through a defined interface. However, an object can implement many interfaces. All objects are separate and operate independent of each other. Each object in the system implements a set of common interfaces defined by the framework. The design of each component is services based and independent of the hardware it encapsulates. The component exports it functionality through an interface that is polymorphic, i.e., there is one generic interface defined for each class of objects that provide similar functionality.

The system objects provide services such as scheduling, logging, event and data management, monitoring and security. Consider the following objects: the system coordinator object, the system repository object, the system scheduler object and the system logger object. Each of these objects has specific responsibilities. The system coordinator object works with the scheduler to coordinate the execution of the tasks on the system. The system repository object is a store for the system configuration. It knows the current configuration and provides services to change the configuration. The system logger object provides services to all other objects to log module specific data through a defined interface.

Communication between various objects is based on a standard messaging model, which was modeled on the Microsoft Component Object Model [4]. Messages are specific and intended for each object. The framework implements an acknowledgement-based architecture to manage communication between various system objects.

Each component is designed to be stateless, i.e., it does not store the run-plan for a specific run. The only persistent data maintained by each component are its communication protocol and configuration (which can be overridden at run time). All the services of the component are configurable at run time. The component provides for an audit trail of all events requested by a client via the interface. The component also provides for structured exception handling mechanisms.

Each peripheral component supports dual activation, stand-alone activation where the services are provided via a GUI and activation via automation. In addition, each peripheral component supports multiple interfaces dedicated to various groups of services. A component also supports a local database to persist configuration, setup and audit information.

The framework provides services to build the top-level application through a defined assembly process. Since the top-level application is built on the services provided by the system objects, it is very loosely coupled with the peripheral components that encapsulate the individual device objects and can be easily reconfigured. This provides an effective strategy for fault-tolerance.

The framework also addresses the integration of enabling technologies such as groupware and workflow management into an automated system. It provides guidelines to integrate on-line analytical processing (OLAP) into the system architecture. In addition, it provides services to publish raw and processed data to the world-wide-web.

The component-based framework supports role-based security. A role is defined as a set of actions and responsibilities associated with a specific activity. In the context of laboratory automation, a role can be thought of as a set of privileges that a user or a set of users can exercise within the context of a particular system. It is a way of abstracting classes of action that can be assigned to a set of users who are allowed to access specific services from a particular component. Access to services exported by a specific component is based upon permissions and privileges assigned to the role of the client (user) of that particular service. Upon authentication, a client (a user or another component) is assigned a specific role and the privileges associated with that particular role are carried around with that user in the context of the automation system. Each resource (service exported by a specific component) has
access configuration that is specific to the role that seeks access to it. When a user tries to access a resource, the system consults the access configuration and obtains permissions associated with that resource. If these permissions match those of the role associated with the user, access to the resource (service) exported by the component is granted. Roles and their associated privileges are persisted across component boundaries in the context of the automated system. The placement of a role as an intermediate to enable a user to exercise a privilege provides much greater control over access configuration than does directly relating users to privileges.

**Figure 1** shows a schematic representation of a microplate system built on this framework. It depicts the four system components and their interactions with the various peripheral components. Consider the Microplate Reader Component. It exports its services through two defined interfaces, IReader and ICommunication. The IReader interface provides access to the services supported by this peripheral component, for example, to read the optical densities of the reagent in a microplate. The ICommunication interface works with the ICoordinate interface of the system scheduler component and provides the services of this peripheral component in automation mode. The IDispatch and the IUnknown interfaces are standard COM interfaces. In addition, this peripheral component also supports a local component database that is used to persist optical density data. All messages between various components adhere to the COM standard. This figure also shows the integration of components supporting workflow, Laboratory Information Management System, the world-wide-web and custom reporting. Four system components are depicted in this diagram, however, framework provides many other system components that offer specific services for audit, log, monitor and alert management [1]. This figure also shows a system repository, which is supported by the framework to persist system configuration. Another important aspect of architecture based upon this framework is the fact that the robot is treated as any other peripheral component.

**DISCUSSION**

With the component based framework, the focus shifts from building applications to building constituent components that can be assembled into applications. This development strategy has a big impact on the way laboratory automation systems are built. Instead of directing the development toward a specific task, component are built to support a generic set of functions. These functions are then used by the application services (services that are...
specific to the task being automated) to provide the functionality required of the automation system. This essentially forms the basis of a multi-tier systems development strategy, the first tier containing the peripheral components, the second tier that contains reusable system services (system components), the third tier that encapsulates the business rules specific to a particular task and the fourth tier that forms the human communication interface (e.g., the Graphical User Interface).

The services based architecture serves to de-couple the peripheral components from the system component and their user services (application services and the human communication interface). The effect of de-coupling components from the top-level application is beneficial in the sense that enhancements, modifications to an existing application do not affect the existing hierarchy. New peripheral components can be added to an existing system without extensive reengineering. Existing functionality of a peripheral component can be enhanced by following the guidelines for developing components that work within the constraints of the framework.

Since the framework supports stateless components, run-time configuration becomes the standard way of configuring the peripheral components. All communications for services and their parameters are requested at run-time and hence, a component can be uninitialised and reinitialised while the system is running. Run-time configuration has an additional benefit of making automation systems fault-tolerant. Since all transactions between the system coordinator and the individual components are atomic (i.e., instantaneous and with only two possible outcomes), it is possible to provide an effective audit of the communication and thereby provide a more robust recovery and rollback mechanisms for the execution of all the Laboratory Unit Operations in the system.

Communication between the constituent components is bi-directional and acknowledgement based. The framework follows the communication mechanism implemented by the Microsoft Component Object Model. COM is a wire-level in-memory binary standard that offers many benefits. It is automatically remappable (provides three types of implementation, Inproc, Local and Remote), provides automatic activation, versioning and dynamic feature negotiation among many other benefits. Details of the COM specification are available elsewhere [4]. Each component implements a mechanism that allows its client to request services synchronously or asynchronously. In both modes of operation, a component involved in either providing a service or receiving a service is responsible for acknowledging the receipt of a message or a service.

COM also offers another benefit that has a direct impact on the scalability of the applications built with this framework. Microsoft Transaction Server technologies provide a powerful platform for managing object lifetimes and thereby enhancing the scalability of the applications built with this framework. MTS specification is available elsewhere [5]. MTS supports intelligent resource management. This aspect of the framework is used to design the scheduling facility to recognize hardware redundancy and provide fault-tolerance.

This framework also supports interface polymorphism. Interfaces are designed based on the nature of the services they provide and are independent of the underlying hardware. Consequently, a single interface is used to support multi-vendor hardware that offers similar functionality. Another implication of this feature is that it becomes easy to substitute hardware from a different vendor to perform the same function thereby making the automated system more versatile. The impact of this feature is that a particular piece of malfunctioning hardware can be replaced with another piece of hardware that provides equivalent functionality without shutting the system down.

Components developed based upon this framework provide for a higher level of reuse compared to objects since, typically, each component encapsulates several objects to provide the functionality required by the interface. Developing components for laboratory automation systems according to the component-based approach, over time, results in a repository of components supporting a multitude of hardware that can be used to build plug-and-play systems. This approach to assembling systems from components instead of building them from low-level modules provides a more application-centric view of the system rather than a robot-centric view. An additional benefit of this approach is that extending existing applications becomes a norm rather than an exception.

Although the business value of the components may not seem immediate, it becomes apparent once it is shown that component reuse offsets the development efforts for new automated systems in a major way. Adapting this framework for the development of new components requires investment in terms of time and resources to build components that support fairly generic functionality. This has an impact on the development timeline and project completion. In addition, it is important to realize that component development is not time-boxed, i.e., the management of component development cannot be held to the same tight timelines as a project since the requirements for these components are fairly generic and they are targeted at a much wider user community. Testing of the function-

<table>
<thead>
<tr>
<th>TRADITIONAL APPROACH</th>
<th>COMPONENT BASED APPROACH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Much of the functionality can be exposed</td>
<td>Often, functionality is exposed via a published</td>
</tr>
<tr>
<td>via a user interface</td>
<td>service and less often, via a user interface</td>
</tr>
<tr>
<td>Development scope is limited to current</td>
<td>Scope has potentially unlimited boundaries</td>
</tr>
<tr>
<td>functionality</td>
<td></td>
</tr>
<tr>
<td>Business requirements are specific</td>
<td>Business requirements are generic</td>
</tr>
<tr>
<td>There is a narrow target user community</td>
<td>There is a wide target user community</td>
</tr>
<tr>
<td>There are few interfaces to other systems</td>
<td>There are potentially many interfaces to</td>
</tr>
<tr>
<td>with significant changes required to add</td>
<td>other systems with little reconfiguration</td>
</tr>
<tr>
<td>new functionality</td>
<td>required to adapt to other systems</td>
</tr>
<tr>
<td>Immediate business value</td>
<td>Long term business value</td>
</tr>
<tr>
<td>Quality Criteria is defined since the system</td>
<td>Stringency of quality criteria is very broad</td>
</tr>
<tr>
<td>is built to specifications</td>
<td>since all possible uses cannot be anticipated.</td>
</tr>
</tbody>
</table>

Table 1. Traditional development approach vs. Component based approach.
ality supported by a component is required to be more rigorous than that is required for an object built to satisfy the documented needs for a specific system. However, the benefits of this approach are for the longer term. This development strategy also increases quality and reliability of the systems since components are tested more rigorously and hence are more reliable. The following table summarizes the unique features of the traditional automation systems development approach and contrasts it with the component based approach.

Figure 2 shows a schematic overview of a component developed following the component-based approach. A unique feature of the components built with this framework is their support for dual instantiation. While the focus on the development of these components is on their services, the component framework requires that each component support stand-alone instantiation. With the dual instantiation feature (stand-alone instantiation and automation instantiation), a component’s services can be effectively used both in the stand-alone mode and also as a part of an integrated system.

Use of this framework in the design and development of automated systems is expected to increase productivity and reduce costs associated with the development process. Since the emphasis is on the development of rugged and tested components, building applications is the process of assembling tested components rather than building systems from scratch. Systems developed based upon this framework result in decreased complexity of the application since the framework supports encapsulated complexity inside each component. As a result, automated systems become more adaptive. Hardware purchased for one specific purpose is adapted to many applications thereby extending its use.

CONCLUSION

This paper discussed the need to revise the existing strategy for developing and implementing automated systems. It outlined a number of features that contribute to the development of flexible and adaptable systems. It also provided a high level overview of a component-based framework that is being developed to build systems whose functionality can be extended. Salient features of the framework were also discussed.

The biggest challenge facing laboratory automation today is lack of standardization. An inordinate amount of time is spent in re-inventing the wheel when a solution to a similar problem is available elsewhere. This has a direct impact on the development life cycle and the time to production. Standardization of communication between peripherals of an integrated system is paramount to addressing this issue. The initial attempt at de-coupling the system components from the peripheral components resulted in the design and development of instrument specific interfaces. However, standardization requires that a set of generic interfaces be modeled and developed both for the system and peripheral components. It is anticipated that refinement of this framework with continual prototyping will provide the next generation of interfaces that will not be instrument specific. Another important issue is scalability of automated systems. This framework addresses this issue by providing centralized facilities that offer services that make an automation system scalable.

This framework is a first step toward reducing systems development timelines by reusing existing functionality supported by well-defined components. We intend to continue refining our approach and extending this framework to allow the inclusion of various platforms and technologies in the hope that one day, all peripheral hardware will be able to communicate to an external client immaterial of the make, model and version of the device driver.

REFERENCES


Copyright© 1998, Bristol-Myers Squibb Company