Case Study: Componentization of an Industrial Control System

Frank Lüders Ivica Crnkovic Andreas Sjögren

ABB Automation Technology Products AB
Control & Force Measurement
SE-777 77 Västerås, Sweden

Mälardalen University
Department of Computer Engineering
PO Box 883, SE-72 123 Västerås, Sweden

{frank.luders, ivica.crnkovic, andreas.sjogren}@mdh.se

Abstract

When different business units of an international company are responsible for the development of different parts of a large system, a component-based software architecture may be a good alternative to more traditional, monolithic architectures. The new common control system, developed by ABB to replace several existing control systems, must incorporate support for a large number of I/O systems, communication interfaces, and communication protocols. An activity has therefore been started to redesign the system's architecture, so that I/O and communication components can be implemented by different development centers around the world. This paper reports on experiences from this effort, describing the system, its current software architecture, the new component-based architecture, and the lessons learned so far.

1. Introduction

Increased globalization and the more competitive climate make it necessary for international companies to work in new ways that maximize the synergies between different business units around the world. Interestingly, this may also require the software architecture of the developed systems to be rethought. In a case where different development centers are responsible for different parts of the functionality of a large system, a component-based architecture may be a good alternative to the more traditional, monolithic architectures, usually comprising a large set of modules with many visible and invisible interdependencies. Additional, expected benefits of a component-based architecture are increased flexibility and ease of maintenance [1][2].

This paper reports on experiences from an ongoing project at ABB to redesign the software architecture of a control system to make it possible for different development centers to incorporate support for different I/O and communication systems. While it is obvious that the component-based approach in the long run brings advantages in terms of time-to-market and less costs for system adaptability and improvements, it is also clear that the redesign itself and the additional costs for designing components to be reusable require more costs in the beginning of the process [3]. Minimizing the additional costs of the project in its starting phase was one of the main challenges. The second challenge of the project was to achieve a good design of the architecture where the interfaces between reusable parts are clear and sufficiently general. The third challenge was to keep the performance of the existing system, since the separation of system parts and introduction of generic interfaces between the parts may cause overhead in the code execution.

The remainder of the paper is organized as follows. In section two, the ABB control system is described with particular focus on I/O and communication. The software architecture and its transformation are described in more detail in section three. A brief analysis of the effects on different quality attributes is also presented. In section four, we analyze the experiences from the project and try to extract some lessons of general value. Section five reviews some related work in this area, and section six present our conclusions and outlines future work.

2. The ABB control system

Following a series of mergers and acquisitions, ABB now has several independently developed control systems for the process, manufacturing, substation automation and related industries. To leverage its worldwide development resources, the company has decided to continue development of only a single, common control system for these industries. One of the existing control systems was selected to be the starting point of the common system. This system is based on the IEC 61131-3 industry standard for programmable controllers [4]. The software has two main parts, the ABB Control Builder, which is a Windows application running on a standard PC, and the system software of the ABB Controller family, running on top of a real-time operating system (RTOS) on special-purpose hardware. The latter is also available as a Windows application, and is then called the ABB Soft Controller.

The ABB Control Builder is used to specify the hardware configuration of a control system, comprising one or more ABB Controllers, and to write the programs that will execute on the controllers. The configuration and the control programs together constitute a control project. When the control project is downloaded to the control system via the control network, the system software of the controllers is responsible for interpreting the configuration information and for scheduling and executing the control programs. Only periodic execution is supported. Figure 1 shows the Control Builder with a control project opened. It consists of three structures, showing the libraries used by the control programs, the control programs themselves, and the hardware configuration, respectively. The latter structure is expanded to show a configuration of a single AC800M controller, equipped with an AI810 analogue input module, a DO810 digital output module, and a CI851 PROFIBUS communication interface.

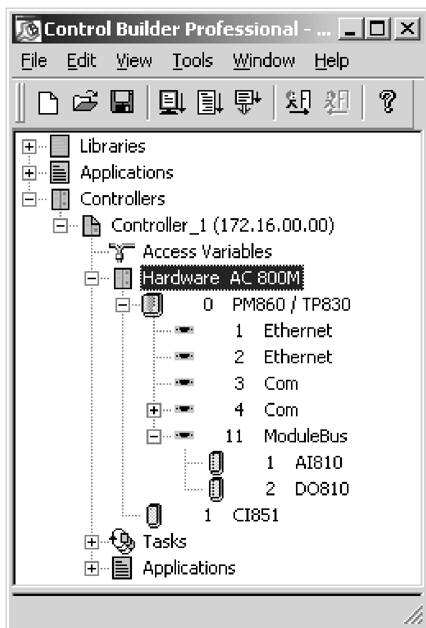

Figure 1. The ABB Control Builder.

To be attractive in all parts of the world and a wide range of industry sectors, the common control system must incorporate support for a large number of I/O systems, communication interfaces, and communication protocols. In the current system, there are two principal ways for a controller to communicate with its environment, I/O and variable communication. When using I/O, variables of the control programs are connected to channels of input and output modules using the Control Builder. For instance, a Boolean variable may be connected to a channel on a digital output module. When the program executes, the value of the variable is transferred to the output channel at the end of every execution cycle. Variables connected to input channels are set at the beginning of every execution cycle. Real-valued variables may be attached to analogue I/O modules.

To configure the I/O modules of a controller, variables declared in the programs running on that controller is associated with I/O channels using the program editor of the Control Builder. Figure 2 shows the program editor with a small program, declaring one input variable and one output variable. Notice that the I/O addresses specified for the two variables correspond to the position of the two I/O modules in Figure 1.

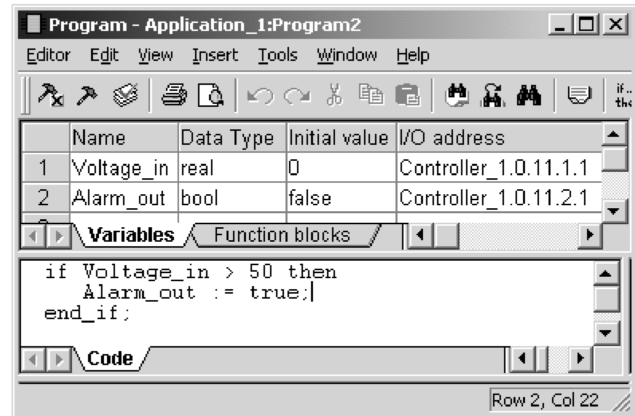

Figure 2. The program editor of the Control Builder.

Variable communication is a form of client/server communication and is not synchronized with the cyclic program execution. A server supports one of several possible protocols and has a set of named variables that may be read or written by clients that implement the same protocol. An ABB Controller can be made a server by connecting program variables to so-called access variables in a special section of the Control Builder. Servers may also be other devices, such as field-bus devices. Any controller, equipped with a suitable communication interface, can act as a client by using special routines for connecting to a server and reading and writing variables via the connection. Such routines for a collection of protocols are available in the Communication Library, which is delivered with the Control Builder.

3. Componentization

3.1. Current software architecture

The software of the ABB Control System consists of a large number of source code modules, each of which are used to build the Control Builder or the controller system software or both. Figure 3 depicts this architecture, with emphasis on I/O and communication. Many modules are also used as part of other products, which are not discussed

further here. This architecture is thus a product line architecture [5], although the company has not yet adopted a systematic product line approach. The boxes in the figure represent logical components of related functionality. Each logical component is implemented by a number of modules, and is not readily visible in the source code.

To see the reason for the overlap in the source code of the Control Builder and that of the controller system software, we look at the handling of hardware configurations. The configuration is specified using the control builder. For each controller in the system, it is specified what additional hardware, such as I/O modules and communication interfaces, it is equipped with. Further configuration information can be supplied for each piece of hardware, leading to a hierarchic organization of information, called the hardware configuration tree. The code that builds this tree in the Control Builder is also used in the controller system software to build the same tree there when the project is downloaded. If the configuration is modified in the Control Builder and downloaded again, only a description of what has changed in the tree is sent to the controller.

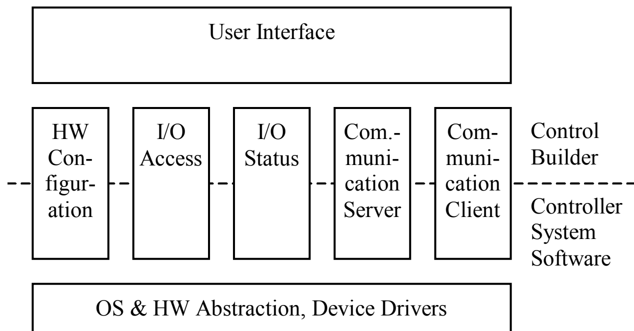

Figure 3. The current software architecture.

The main problem with the current software architecture is related to the work required to add support for new I/O modules, communication interfaces, and protocols. For instance, adding support for a new I/O system may require source code updates in all the components except the User Interface and the Communication Server, while a new communication interface and protocol may require all components except I/O Access to be updated.

As an example of what type of modifications may be needed to the software, we consider the incorporation of a new type of I/O module. To be able to include a device, such as an I/O module, in a configuration, a hardware definition file for that type of device must be present on the computer running the Control Builder. For an I/O module, this file defines the number and types of input and output channels. The Control Builder uses this information to allow the module and its channels to be configured using a generic configuration editor. This explains why the user interface does not need to be updated to support a new I/O module. The hardware definition file also defines the memory layout of the module, so that the transmission of data between program variables and I/O channels can be implemented in a generic way.

For most I/O modules, however, the system is required to perform certain tasks, for instance when the configuration is compiled in the Control Builder or during start-up and shutdown in the controller. In today's system, routines to handle such tasks must be hard-coded for every type of I/O module supported. This requires software developers with a thorough knowledge of the source code. The situation is similar when adding support for communication interfaces and protocols. The limited number of such developers therefore constitutes a bottleneck in the effort to keep the system open to the many I/O and communication systems found in industry.

3.2. Component-based software architecture

To make it much easier to add support for new types of I/O and communication, it was decided to split the components mentioned above into their generic and non-generic parts. The generic parts, commonly called the generic I/O and communication framework, contains code that is shared by all hardware and protocols implementing certain functionality. Routines that are special to a particular hardware or protocol are implemented in separate components, called protocol handlers, installed on the PC running the Control Builder or on the controllers. This component-based architecture is illustrated in Figure 4.

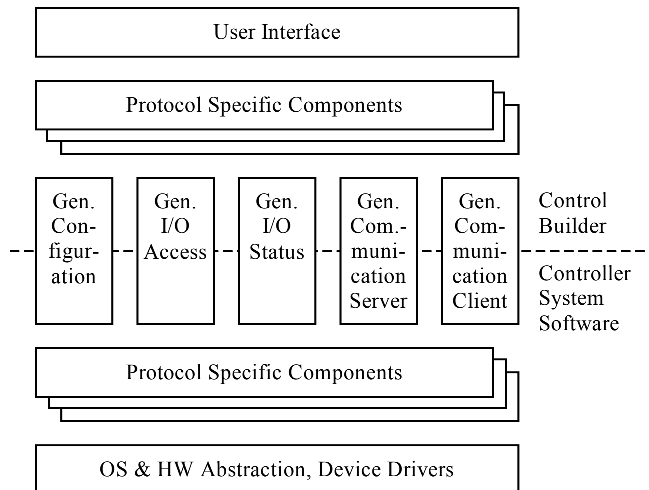

Figure 4. Component-based software architecture.

To add support for a new I/O module, communication interface, or protocol to this system, it is only necessary to add protocol handlers for the PC and the controller along

with a hardware definition file. The format of hardware definition files is extended to include the identities of the protocol handlers.

Essential to the success of the approach, is that the dependencies between the framework and the protocol handlers are fairly limited and, even more importantly, well specified. One common way of dealing with such dependencies is to specify the interfaces provided and required by each component. ABB's component-based control system uses Microsoft's Component Object Model (COM) [6] to specify these interfaces, since COM provides suitable formats both for writing interface specification, using the COM Interface Description Language (IDL), and for run-time interoperability between components. For each of the generic components, two interfaces are specified: one that is provided by the framework and one that may be provided by protocol handlers. Interfaces are also defined for interaction between protocol handlers and device drivers. The identities of protocol handlers are provided in the hardware definition files as the Globally Unique Identifiers of the COM classes that implement them.

The use of COM implies that all invocations of an interface's operations are sent to a particular object. This turns out to work very well for this system, as it allows several instances of the same protocol handler to be created. This is useful, for instance, when a controller is connected to two separate networks of the same type. Also, it is useful to create one instance of the class implementing an interface provided by the framework for each protocol handler that requires the interface. An additional reason that COM is the technology of choice is that it is expected to be available on all operating systems that the software will be released on in the future. The Control Builder is only released on Windows, and an effort has been started to port the controller system software from pSOS to VxWorks. In the first release of the system, which will be on pSOS, the protocol handlers will be implemented as C++ classes, which will be linked statically with the framework. This works well because of the close correspondence between COM and C++, where every COM interface has an equivalent abstract C++ class.

When a control system is configured to use a particular device or protocol, the Control Builder uses the information in the hardware definition file to load the protocol handler on the PC and execute the protocol specific routines it implements. During download, the identity of the protocol handler on the controller is sent along with the other configuration information. The controller system software then tries to load this protocol handler. If this fails, the download is aborted and an error message displayed by the Control Builder. This is very similar to what happens if one tries to download a configuration, which includes a device that is not physically present. If the protocol handler is available, an object is created and the required interface pointers obtained. Objects are then created in the framework and interface pointers to these passed to the protocol handler. After the connections between the framework and the protocol handler has been set up through the exchange of interface pointers, a method will usually be called on the protocol handler object that causes it to continue executing in a thread of its own. Since the interface pointers held by the protocol handler references objects in the framework, which are not used by anyone else, all synchronization between concurrently active protocol handlers can be done inside the framework.

To make this more concrete, we now consider the interface pair IGenServer, which is provided by the framework, and IPhServer, which is provided by protocol handlers implementing the server side of a communication protocol on the controllers. Figure 5 is a UML structure diagram showing the relationships between interfaces and classes involved in the interaction between the framework and such a protocol handler. The class CMyProtocol represents the protocol handler. The interface IGenDriver gives the protocol handler access to the device driver for a communication interface.

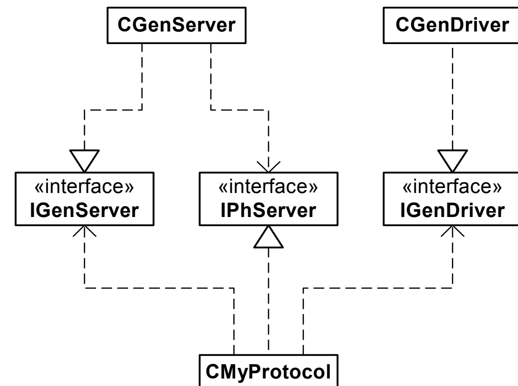

Figure 5. Interfaces for communication servers.

The definition of IPhServer is shown below. Three operations are defined by this interface. The first two are used to pass interface pointers to objects implemented by the framework to the protocol handler. The other two operations are used to start and stop the execution of the protocol handler in a separate thread.

```
Interface IPhServer : IUnknown
{
    HRESULT SetServerCallback(
        [in] IGenServer *pGenServer);
    HRESULT SetServerDriver (
        [in] IGenDriver *pGenDriver);
    HRESULT ExecuteServer();
    HRESULT StopServer();
};
```

The UML sequence diagram in Figure 6 shows an example of what might happen when a configuration is

downloaded to a controller, specifying that the controller should provide server-side functionality. The system software first invokes the COM operation CoCreateInstance to create a protocol handler object and obtain an IPhServer interface pointer. Next, an instance of CGenServer is created and a pointer to it passed to the protocol handler using SetServerCallback. Similarly, a pointer to a CGenDriver object is passed using SetDriverCallback. Finally, ExecuteServer is invoked, causing the protocol handler to start running in a new thread.

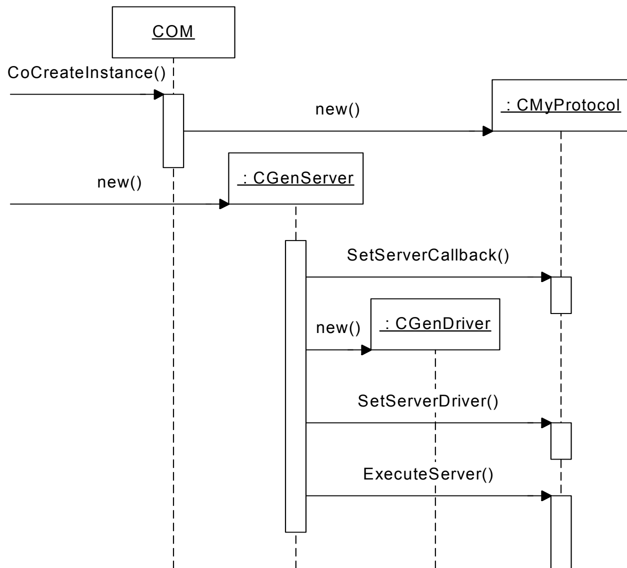

Figure 6. Call sequence to set up connections.

To see how the execution of the protocol handler proceeds, we first look at the definition of IGenServer. This interface defines four operations. The two first are used to inform the framework about incoming requests from clients to establish a connection and to take down an existing connection. The two last operations are used to handle requests to read and write named variables, respectively. The index parameter is used with variables that hold structured data, such as records or arrays. All the methods have an out parameter that is used to return a status word.

```
Interface IGenServer : IUnknown
{
    HRESULT Connect([out] short *status);
    HRESULT Disconnect([out] short *status);
    HRESULT ReadVariable(
        [in] BSTR *name, [in] short index,
        [out] tVal *pVal, [out] short *status);
    HRESULT WriteVariable(
        [in] BSTR *name, [in] short index,
        [in] tVal *pVal, [out] short *status);
};
```

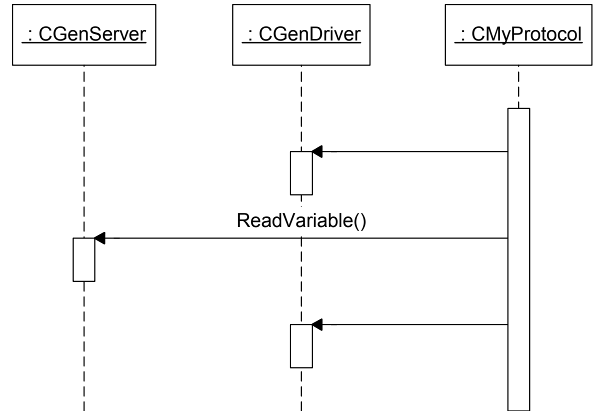

Figure 7. Call sequence to handle variable read.

Running in a thread of its own, the protocol handler uses the IGenDriver interface pointer to poll the driver for incoming requests from clients. When a request is encountered the appropriate operation is invoked via the IGenServer interface pointer, and the result of the operation, specified by the status parameter, reported back to the driver and ultimately to the communication client via the network. As an example, Figure 7 shows how a read request is handled by calling ReadVariable. The definition of the IGenDriver interface is not included in this discussion for simplicity, so the names of the methods invoked on this interface are left unspecified in the diagram. Write and connection oriented requests are handled in a very similar manner to read requests.

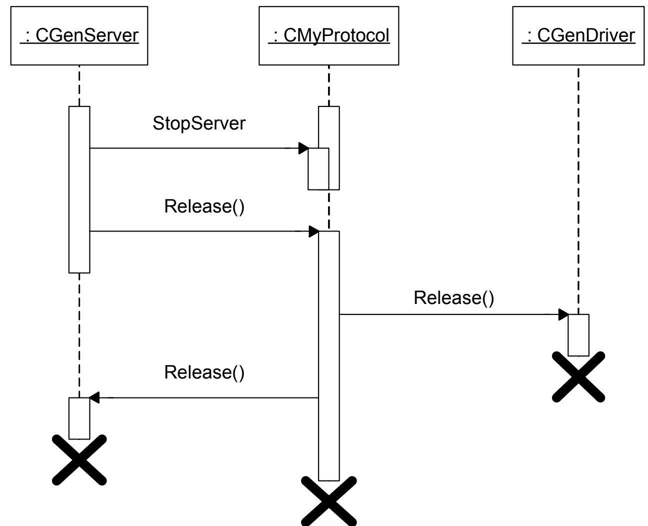

Figure 8. Call sequence to take down connections.

The last scenario to be considered here, is the one where configuration information is downloaded, specifying that a protocol handler that was used in the previous configuration should no longer be used. In this case, the

connections between the objects in framework and the protocol handler must be taken down and the resources allocated to them released. Figure 8 shows how this is accomplished by the framework first invoking StopServer and then Release on the IPhServer interface pointer. This causes the protocol handler to decrement its reference count, and to invoke Release on the interface pointers that have previously been passed to it. This in turn, causes the objects behind these interface pointers in the framework to release themselves, since their reference count reaches zero. Assuming that its reference count is also zero, the protocol handler object also releases itself. If the same communication interface, and thus the protocol handler object, had also been used for different purposes, the reference count would have remained greater than zero and the object not released.

3.3. Quality attribute analysis

The software architecture of a system is considered a primary means of achieving the correct quality attributes for the system [7]. In this section, the possible effects of componentization on the quality attributes of the ABB control system is analyzed. This analysis is based on preliminary experiences with the system as well as our reflections on the general effects of adopting a component-based architecture. The quality attributes discussed include attributes observable at run time, such as performance and reliability, and attributes such as maintainability and scalability, which are only observable during development.

Performance. As for all embedded, real-time systems, performance in terms of both time and memory usage is a primary concern for the controller. It is not expected that the componentization will affect the system's ability to meet its real-time deadlines, since code related to I/O and communication in the framework as well as the protocol handlers will execute in threads of lower priority than the time-critical control tasks. A component technology such as COM is expected to introduce some memory overhead. By taking care only to use expensive features when absolutely necessary, however, general experience with COM indicates that this overhead will be acceptable.

Reliability. The integration of independently developed components into an industrial system raises the question of reliability. Special functions for supervision of components and possibly automatic reset of components exhibiting faulty behavior might be necessary to detect and contain the effects of faulty components. Although no such functions have been implemented, it is expected that supervision of software components can be added without to much effort by reusing existing functions for supervision of hardware components.

Maintainability. The maintainability of the system, defined as the ease of making corrections, adaptations, and extensions to the system, should be positively affected by the adoption of a component-based architecture. Changes made to a component that only interacts with the rest of the system through well-defined interfaces, is less likely to have unforeseen consequences for other parts of the system than changes made to a module with many visible and invisible interdependencies with other modules.

Scalability. One aspect of scalability, the possibility to deploy the software on platforms of varying size and performance, is an important concern for the controller system software. The component-based architecture is expected to have a positive affect on this attribute, since protocol handlers can easily be left out on platform where they will not be used. The possibility of using the generic interfaces without relying on COM and dynamic linking makes it easy to deploy the software on platforms where the overhead of a component technology cannot be afforded or where COM support is not available.

4. Lessons learned

The definitive measure of the success of the project described in this paper will be how large the effort required to redesign the software architecture has been compared to the effort saved by the new way of adding I/O and communication support. It is important to remember, however, that in addition to this cost balance, the business benefits gained by shortening the time to market must be taken into account. Also important, although harder to assess, are the long time advantages of the increased flexibility that the component-based software architecture is hoped to provide.

At the time of writing, the design of the framework, including the specification of interfaces, is largely completed and implementation has started. It is thus too early to say exactly how much work has been needed, but it seems safe to conclude that the efforts are of the same order of magnitude as the work required to add support for an advanced I/O or communication system the old way, that is by adding code to the affected modules. From this we can infer, that if the new software architecture makes it substantially easier to add support for such systems, the effort has been worthwhile. We therefore find that the experiences with the ABB control system supports our hypothesis that a component-based software architecture is an efficient means for supporting distributed development of complex systems.

Another lesson of general value is that it seems that a component technology, such as COM, can very well be used on embedded platforms and even platforms where run-time support for the technology is not available. Firstly, we have seen that the overhead that follows from using COM is not larger than what can be afforded in many embedded systems. In fact, used with some care, COM does not introduce much more overhead than do virtual methods in C++. Secondly, in systems where no such

overhead can be allowed, or systems that run on platforms without support for COM, IDL can still be used to define interfaces between components, thus making a future transition to COM straightforward. This takes advantage of the fact that the Microsoft IDL compiler generates C and C++ code corresponding to the interfaces defined in an IDL file as well as COM type libraries. Thus, the same interface definitions can be used with systems of separately linked COM components and statically linked systems where each component is realized as a C++ class or C module.

An interesting experience from the project is that techniques that were originally developed to deal with dynamic hardware configurations have been successfully extended to cover dynamic configuration of software components. In the ABB control system, hardware definition files are used to specify what hardware components a controller may be equipped with and how the system software should interact with different types of components. In the redesigned system, the format of these files has been extended to specify which software components may be used in the system. The true power of this commonality is that existing mechanisms for handling hardware configurations, such as manipulating configuration trees in the Control Builder, downloading configuration information to a control system, and dealing with invalid configurations, can be reused largely as is. The idea that component-based software systems can benefit by learning from hardware design is also aired in [1].

5. Related work

The use of component-based software architecture in real-time, industrial control has not been extensively studied, as far as we know. One example is documented in [8]. This work is not based on experiences from industrial development, however, but rather from the construction of a prototype, developed in academia for non-real-time platforms with input from industry. It also differs from our work in that it focuses on the possibility of replacing the multiple controllers usually found in a production cell with a single controller, rather than on supporting distributed development.

The use of software components in embedded systems is also discussed in [9]. This work is more ambitious than ours in one sense, as it focuses on techniques and tools to ensure correct composition of components. It is more limited in another way, however, since dynamic configuration is not handled by the suggested techniques.

An example of a commercial system that supports component-based development of control systems is ControlShell [10]. This system is, however, substantially different from the system described in this paper, since ControlShell focuses on constructing control systems from re-usable components, using a graphical editor and automatic code generation, and is not concerned with independently deployable components and dynamic system configuration.

6. Conclusions and future work

The initial experiences from the effort to redesign the software architecture of ABB's control system to support component-based development are promising, in that the developers have managed to define interfaces between the framework and the protocol handlers. Since the effort to redesign the system has not been too extensive, we conclude that the project has met its first challenge successfully. Preliminary results using emulated COM suggest that the performance of the systems will be acceptable. A solution based on COM has yet to be implemented.

An issue that may be addressed in the future development at ABB is richer specifications of interfaces. COM IDL only specifies the syntax of interfaces, but it is also useful to specify loose semantics, such as the allowed parameters and possible return values of methods, and timing constraints. Since UML has already been adopted as a design notation, one possibility is to use the specification style suggested in [11]. One concern, however, is the lack of support for specifying timing constraints in UML [12]. Another continuation of the work presented here, would be to extend the component approach beyond I/O and communication. An architecture were general functionality can be easily integrated by adding independently developed components, would be a great benefit to this type of system, which is intended for a large range of control applications.

In our continued research concerning this effort we plan to study in more detail how different quality attributes are addressed by the software architecture. We will, for instance, look at reliability, which is an obvious concern when externally developed software components are integrated into an industrial system. We have already claimed that the experiences recorded in this paper support our hypothesis that component-based software architectures is a good alternative to monolithic architectures for complex systems developed in distributed organizations. It will be a primary goal of our future work to strengthen this claim by presenting data that verifies that the development of I/O and communication support is made substantially easier by the new architecture.

7. Acknowledgements

The project described in this paper is carried out at ABB Automation Technology Products in Malmö, Sweden. We gratefully acknowledge the financial support of ABB and the Swedish KK Foundation.

8. References

[1] C. Szyperski, *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

[2] H. Hermansson, M. Johansson, L. Lundberg, "A Distributed Component Architecture for a Large Telecommunication Application", *Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, December 2000.

[3] I. Crnkovic, M. Larsson, "A Case Study: Demands on Component-Based Development", *Proceedings of 22nd International Conference of Software Engineering*, May 2000.

[4] International Electrotechnical Commission, *Programmable controllers - Part 3: Programming languages*, IEC Standard 61131-3, 1993.

[5] J. Bosch, *Design and Use of Software Architectures – Adopting and Evolving a Product Line Approach*, Addison-Wesley, 2000.

[6] Microsoft Corporation, *The Component Object Model Specification, Version 0.9*, October 1995.

[7] L. Bass, P. Clements, R. Katzman, *Software Architecture in Pracice*, Addison-Wesley, 1998.

[8] A. Speck, "Component-Based Control System", *Proceedings of the Seventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, April 2000.

[9] T. Genssler, C. Zeidler, "Rule-Driven Component Composition for Embedded Systems", *Proceedings of the Fourth ICSE Workshop on Component-Based Software Engineering*, May 2001.

[10] S. A. Schneider, V. W. Chen, G. Pardo-Castellote, "ControlShell: Component-Based Real-Time Programming", *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, May 1995.

[11] J. Cheesman, J. Daniels, *UML Components – A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2001.

[12] The Object Management Group, *UML Profile for Scheduling, Performance, and Time: Request for Proposal*, December 1999.