A Component-Based Software Architecture for Industrial Control

Frank Lüders, Ivica Crnkovic, Andreas Sjögren
ABB Automation Technology Products, Mälardalen University

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.

Key words: software architecture, component-based software development, industrial control systems

1. INTRODUCTION

Increased globalisation and the more competitive climate make it necessary for international companies to work in new ways that maximize the synergies between business units around the world. Interestingly, this may also require the software architecture [1] of the developed systems to be rethought. In a case where different development centres are responsible for different parts of the functionality of a large system, a component-based architecture may be a good alternative to 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 [2][3].

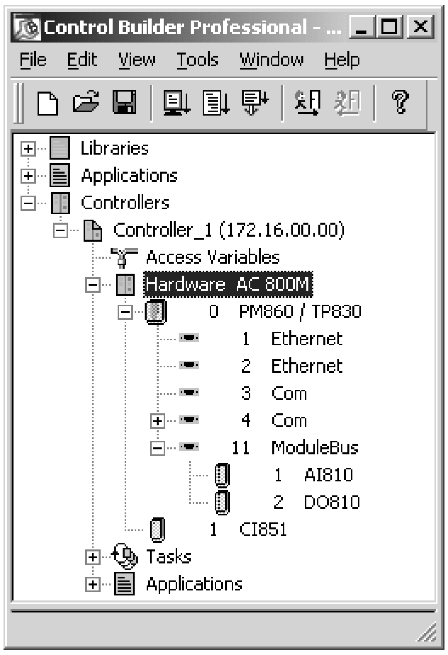
This paper reports on experiences from an ongoing project at ABB to redesign the software architecture of a control system to make it possible for different development centres to incorporate support for different I/O and communication systems. While it seems obvious that a component-based architecture brings advantages in the long run, it is clear that the redesign itself and the additional effort for designing components to be reusable require more costs in the beginning of the process [4]. 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. In section four, we analyse 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. 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.

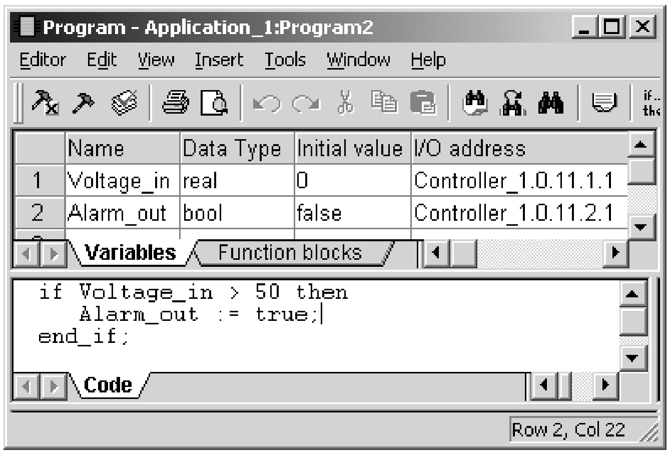
The ABB Control Builder is used to specify the hardware configuration of a control system, comprising one or more ABB Controllers, and to write the programs that will execute on the controllers. When the configuration and the control programs, commonly called a control project, are downloaded to the 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-DP communication interface.



```
Libraries
Applications
Controllers
  Controller_1 (172.16.00.00)
    Access Variables
    Hardware  AC 800M
      0   PM860 / TP830
            1   Ethernet
            2   Ethernet
            3   Com
            4   Com
           11   ModuleBus
                  1   AI810
                  2   DO810
      1   CI851
    Tasks
    Applications
```

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 program editor of the Control Builder. When the program executes, variables connected to input channels are set at the beginning of every execution cycle while the value of variables connected to output channels is transferred to the channel at the end of every execution cycle. Real-valued variables may be attached to analogue I/O channels and Boolean variables to digital I/O channels. *Figure 2* shows the 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*.


Program - Application_1:Program2
Editor Edit View Insert Tools Window Help
Variables
Function blocks
Code
Row 2, Col 22

	Name	Data Type	Initial value	I/O address
1	Voltage_in	real	0	Controller_1.0.11.1.1
2	Alarm_out	bool	false	Controller_1.0.11.2.1

```
if Voltage_in > 50 then
   Alarm_out := true;
end_if;
```

Figure 2. The program editor of the ABB 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. A 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.

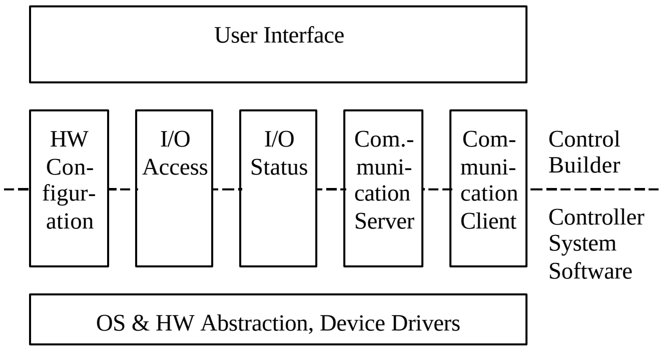


Figure 3. The current software architecture

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.

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 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*. 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], since it provides suitable formats both for writing interface specification, using the COM Interface Definition 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 (GUIDs) of the COM classes that implement them. 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. In the first release of the system, which will be on a platform without COM support, the protocol handlers will be implemented as C++ classes, which will be linked statically with the framework. This works because the Microsoft IDL compiler generates C and C++ code corresponding to the interfaces defined in an IDL file as well as COM type libraries.

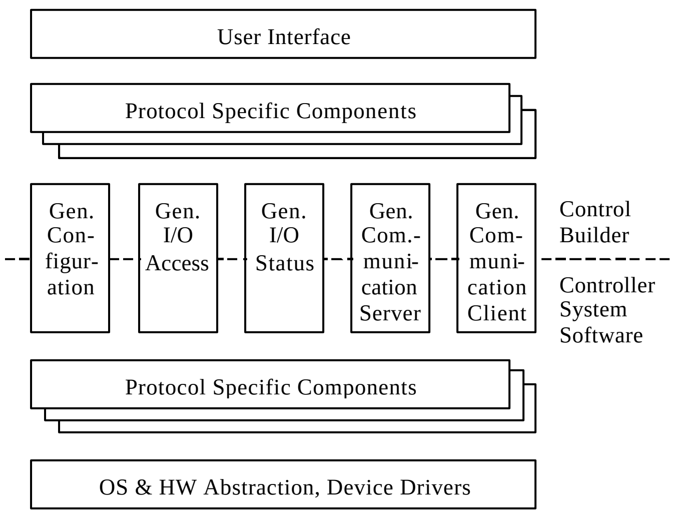

User Interface
Protocol Specific Components
Gen. Con-figur-ation
Gen. I/O Access
Gen. I/O Status
Gen. Com.-muni-cation Server
Gen. Com-muni-cation Client
Control Builder
Controller System Software
Protocol Specific Components
OS & HW Abstraction, Device Drivers

Figure 4. Component-based software architecture

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 as part of the 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, just as if one tries to download a configuration with 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.

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.

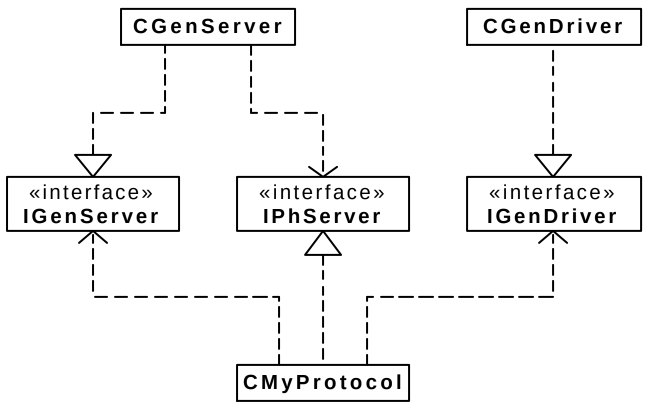


Figure 5. Interfaces for communication servers.

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.

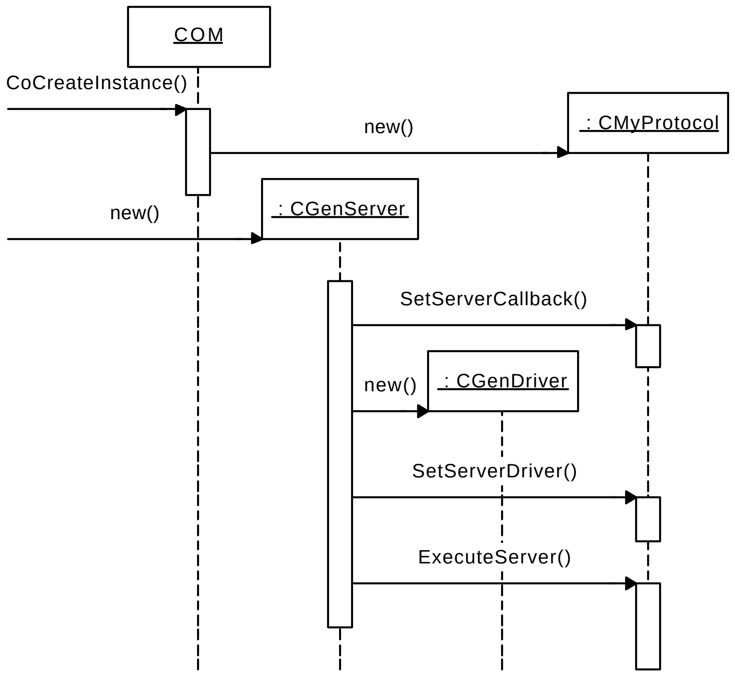


Figure 6. Call sequence to set up connections.

The scenario presented here is related to the use of a protocol handler on a controller that acts as a server for some communication protocol. However, many of the principles are the same as for other parts of the system. Examples of such princ iples are those used for setting up and taking down connections between the framework and a protocol handler and the execution of a protocol handler in a separate thread.

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. At the time of writing, the specification of generic interfaces and the implementation the framework are largely completed, and it we can 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. It is an interesting fact that more efforts than originally planned were required to understand the architecture of the old controller. It also took additional efforts for training of developers to start with the re-design using UML and to analyse the system from the architecture point of view before starting with the detailed design and the implementation. Another additional effort was required to get familiar with COM technology and IDL. Once the overall system architecture was specified and the developers get used with UML, the efficiency of the development was considerably higher. 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. From the architecture and modifiability point of view, the system is significantly improved. The component developers (when introducing a new protocol or new I/O modules) do not need to be familiar with implementation details of the entire systems, but must strictly follow the interface specification. 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.

A 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, the same interface definitions can be used with protocol handlers implemented as dynamically linked COM components and statically linked C++ classes or C modules.

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 can be reused largely as is. The idea that component-based software systems can benefit by learning from hardware design is also aired in [2].

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 [7], which describes work not based on industrial experiences, but from the construction of a prototype, developed in academia for non-real-time platforms with input from industry. A research project focusing on tools end techniques for ensuring correct composition of components in embedded systems is described in [8]. An example of a commercial system for component-based development of real-time control systems is ControlShell [9], which supports construction from re-usable components using a graphical editor and automatic code generation.

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. Since the effort to redesign the system has not been too extensive, we conclude that the project has met its first challenge successfully. An assessment of how the remaining challenges of achieving sufficiently general interfaces while maintaining an acceptable performance have been met would be premature at this point.

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. In addition, we plan to study in more detail how non-functional requirements are addressed by the software architecture, since the architecture of a system is often seen as a primary means for meeting such requirements [1]. We will, for instance, look at reliability, which is an obvious concern when externally developed software components are integrated into an industrial system. Other goals are to investigate the additional expected benefits of increased flexibility and ease of maintenance and to compare the performance of the system after the redesign to that of the current system.

7. REFERENCES

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

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

[3] 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.

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

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

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

[9] 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.