Adopting a Software Component Model in Real-Time Systems Development

Frank Lüders
ABB Automation Technologies, Control Platform Products
Lugna Gatan, Building 357, SE-721 59 Västerås, Sweden
frank.a.luders@se.abb.com

Abstract

Component-based software engineering (CBSE) and the use of (de-facto) standard component models have gained popularity in recent years, particularly in the development of desktop and server-side software. This paper presents a motivation for applying CBSE to real-time systems and discusses the consequences of adopting a software component model in the development of such systems. Specifically, the consequences of adopting Microsoft's COM, DCOM, and .NET models are analyzed. The most important aspects of these models are discussed in an incremental fashion. The analysis considers both real-time systems in general, and a real-life industrial control system where some aspects the COM model have been adopted. It is concluded that adopting these models makes it possible to meet real-time requirements, but that some overhead must be expected and that special precautions may have to be taken to prevent loss of real-time predictability.

1. Introduction

Component-based software engineering (CBSE) denotes the assembling of software products from pre-existing smaller products, generally called components. In particular when this is done using standard or de-facto standard component models and supporting technologies [1]. A component model generally defines a concept of components and rules for their design-time composition and/or run-time interaction, and is usually accompanied by one or more component technologies, implementing support for composition and/or interoperation.

In recent years, the use of component models has gained popularity in the development of desktop and server-side software. Two popular models in desktop applications are Sun's *JavaBeans* [2] and Microsoft's *ActiveX controls* [3], where the latter is built on top of the more basic *Component Object Model* (COM) [4]. Both of these are particularly suited for components to be used with visual composition tools. The best-known models in the server domain are Sun's *Enterprise JavaBeans* (EJB) [5], Microsoft's COM extension *COM+* [6], and the Object Managements Group's new *CORBA Component Model* (CCM) [6]. These models offer similar support for transactional processing and persistent data management.

This paper discusses the possibilities of using such component models in real-time systems. In particular, the feasibility of using COM, the most basic of these models, and its distributed extension is analyzed and illustrated through a case study. Microsoft's latest model *.NET* [8] is also briefly discussed. Section two presents motivations for adopting a component model, both in real-time systems generally and in a real-world industrial control system. Section three discusses the implications of adopting different aspects of a particular component model. An overview of related work is given in Section four. Finally, Section five concludes the paper and outlines future work.

2. Motivation

The general motivation for component-based software engineering is the prospect of increased productivity and timeliness of software development projects. Indeed, this is as desirable for real-time and embedded software as for any other application. It could also be argued that some characteristics of CBSE make it particularly attractive for real-time systems. For instance, real-time software often requires more extensive testing, so the use of pre-tested components may be particularly time saving in the development of such system. Another example is that many embedded systems, such as mobile telephones, could benefit from reuse of components across products and models. Conversely, there are also barriers to CBSE particular to real-time and embedded systems. Most obviously, there may be a risk that component models and technologies may introduce unacceptable overhead or loss of predictability.

An example of a real-time system where the use of a component model has been useful is the industrial control system by ABB called Control^{IT} (http://www.abb.com). This product is a modular controller consisting of a central processing unit with two expansion buses. One bus is for I/O modules of different types and is used to connect the controller to physical signals. The other bus is for communication interfaces and allows the controller to communicate with other devices using different media

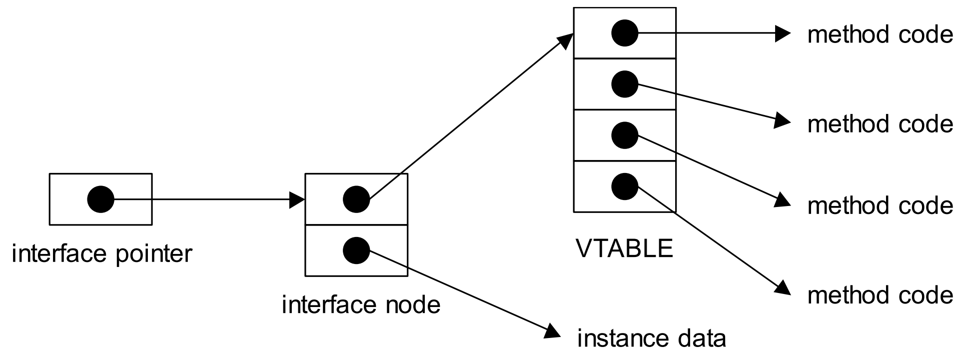

Figure 1. Typical format of COM interface nodes

and protocols. The controller also has two built-in serial ports and redundant Ethernet ports.

ABB's development organization is globally distributed, and the interest in component models first arose from a wish to make it easier for different development centers to add I/O and communication support to the system. It was decided to redesign the system's architecture so that all code particular to a certain I/O module, communication interface, or protocol resides in a separate component called a protocol handler. To achieve this, rules and formats for interaction between these protocol handlers and the rest of the system had to be decided on. In other words, a component model was needed. In the following analysis of adopting different aspects of a component model, the usefulness and liabilities of each particular aspect in connection with protocol handlers will be discussed. The use of a component model to support integration of protocol handlers in ABB's control system is further described in [9], where it is demonstrated that the new architecture supports distributed development and reduces the time required to implement I/O and communication support.

3. Adopting Microsoft Models

Among the most commonly used component models for desktop and server applications are Microsoft's Component Object Model (COM) and its extension Distributed COM (DCOM) [10]. There is also great interest in the company's new generation of technologies, commonly denoted .NET, which also defines a component model [8]. This section explores the possibilities of using these models in real-time systems. The most important aspects of these models will be discussed in an incremental fashion, assuming that it may be desirable in some situations also to adopt the models in such a fashion.

3.1. COM Interfaces

A key principle of COM and other component models is that interfaces are specified separately from both the components that implement them and those that use them. COM defines a dialect of the *Interface Definition Language* (IDL) that is used to specify object-oriented interfaces. Interfaces are object-oriented in the sense that their operations are to be implemented by a class and passed a reference to a particular instance of that class when invoked. The code that uses a component does not refer directly to any objects, however. Instead, the operations of an interface supported by an object are invoked via what is known as an *interface pointer*. A concept known as interface navigation makes it possible for the user to obtain a pointer to every interface supported by the object. For a further description of this topic, see e.g. [10].

COM also defines a run-time format for interface pointers. What an interface pointer really references is an *interface node*, which in turn, contains a pointer to a table of function pointers, called a *VTABLE*. Typically, the node also contains a pointer to an object's instance data, although this is up to the implementation (of the supporting component technology). This use of VTABLEs is identical to the way that many C++ compilers implement virtual functions. Thus, the time and space overhead associated with accessing an object through an interface pointer is the same as that incurred with virtual C++ functions. This time overhead is very modest. The memory overhead should also be acceptable, perhaps except for the most resource constrained embedded systems. Figure 1 illustrates the typical format of interface nodes.

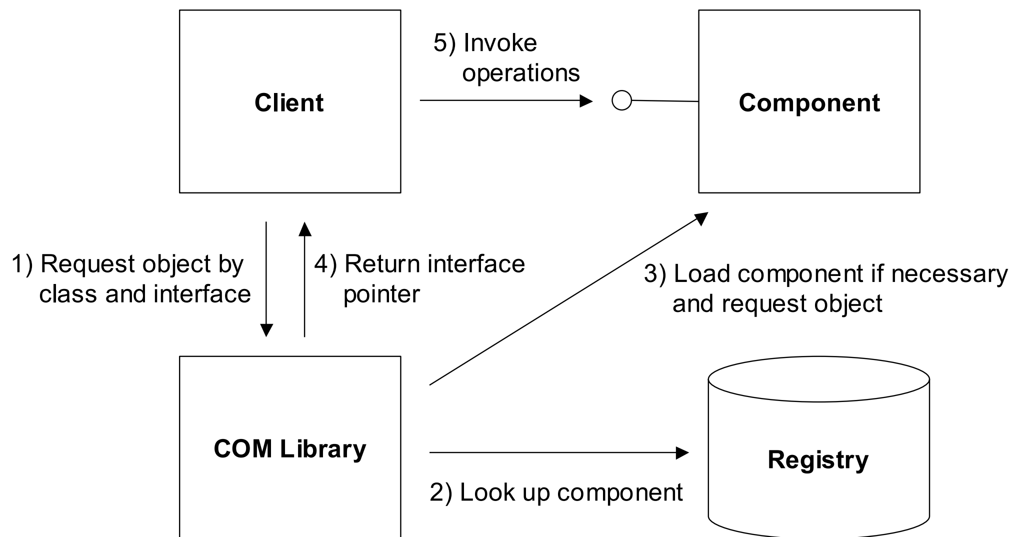

Figure 2. Instance creation and dynamic loading of code in COM

For most real-time systems, a more serious concern than these modest overheads is that interface navigation introduces a possible source of run-time errors. If the user of a component asks an object for a pointer to an interface that the object does not support, this will not be detected during compilation. It may be argued, in fact, that this is the principal difference between interface navigation and interface inheritance in traditional object-oriented programming. This can be seen as a necessary price to pay for the otherwise desirable reduced compile-time dependence between components.

Most real-time systems are based on multi-tasking and are often built on top of a real-time operating system (RTOS) using some kind of priority-based scheduling. Developers of components for real-time systems will generally need to make assumptions about how their components will be used in a multi-tasking environment. The safest option will be always to assume that an object can be concurrently used by several tasks, and guard all methods with the necessary synchronization. For reasons of efficiency, however, it may be more desirable to require the code that uses the component to provide any necessary synchronization. The exact circumstances under which such protection is necessary are thus an important part of the component's documentation.

The use of COM IDL to specify interfaces and VTABLEs to implement interface pointers work well for protocol handlers. The concept of multiple interfaces per object with interface navigation is useful since different protocol handlers must provide different functionality. The object-oriented nature of COM interfaces where each interface pointer refers to a particular instance of a class also matches the needs of the ABB control system. Multiple instances of the same protocol handler are useful, e.g. when a controller is equipped with two identical communication interfaces, linking it to two separate networks of the same type. The latest version of the control system uses COM interfaces, but not the other parts of COM discussed below.

3.2. Instantiation and Dynamic Linking

The previous section stated that the code of a COM component is implemented in classes, without discussing how instances are created. Also, nothing was said about how and when the code in different components is linked together. COM defines a policy for instantiation, which is intended to ensure that different components can be installed in a system at different times. When a component is installed, information about it must be registered somewhere in the system, linking the identity of its classes to the code that implement these. COM also requires a run-time library, called the COM library, to be installed on the system. When some code wants to use a component, it uses an operation provided by the COM library to ask for an instance of a class and an initial interface pointer to it. If the code of the component is not already loaded into memory, the COM library uses the registered information to locate the code and load it before an instance is created. This process is illustrated in Figure 2.

Thus, creation of an instance involves searching the information about registered classes and possibly loading of code. This leads to a noticeable overhead when compared to instantiation in for instance C++. Furthermore, this overhead will vary, depending on

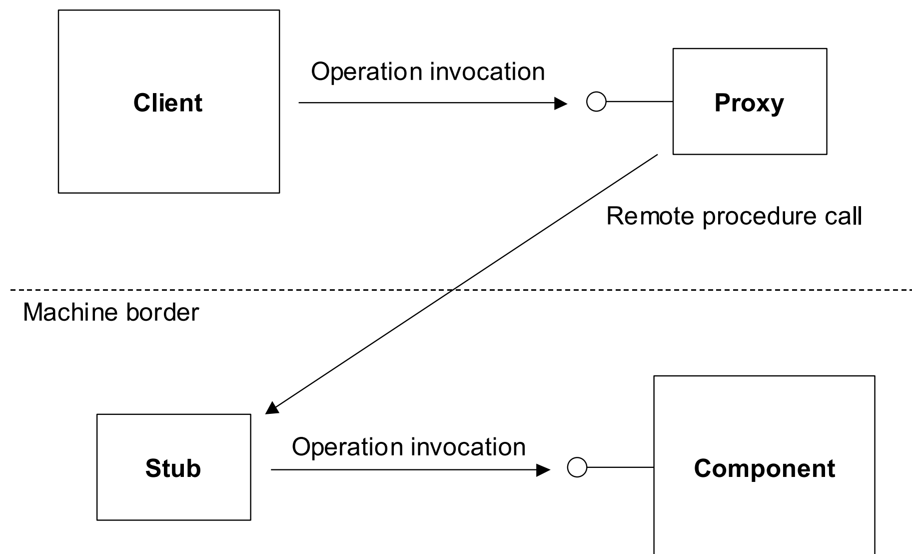

Figure 3. Use of proxy and stub objects in DCOM

whether the code implementing a class has already been loaded or not. This variability can be eliminated, however, by designing the software such that all components that may be used will be loaded at start-up. Note that removal of instances is subject to the same variability, since the COM standard states that code can be unloaded when the last instance that rely on it is removed.

A benefit that follows from COM's way of creating instances is that the code that implements a component can be built independently of any code that uses the component. Since instantiation involves passing the identity of the desired class as a parameter to a system operation, it is a possible source of run-time errors, which is not present during instantiation in traditional object-oriented programming, since attempting to instantiate a class that does not exist will result in a compilation error in this case. Again, this is a necessary price to be paid for decreased coupling.

COM's principle of instantiation is well suited for creating instances of protocol handlers, since no knowledge of the set of available protocol handlers should be built into the system. The overhead associated with looking up classes and dynamic loading of code is expected to be tolerable, especially since the software is designed such that protocol handlers need only be instantiated and deleted during program download. Thus, the extra time taken by this way of instantiation will not interfere with the continuous operation of the system. An additional benefit of using this technique for instantiation is that protocol handlers can be deployed (and updated) independently of the rest of the system. Future versions of the control system may include a COM library and employ dynamic linking of components. It is possible that a commercial component technology, such as WindRiver's implementation of COM for the VxWorks RTOS (http://www.windriver.com) will be used.

3.3. Location Transparency with DCOM

DCOM is an extension of COM, which allows component-based applications to be distributed across memory spaces or physical machines. This is realized using auxiliary objects known as proxies and stubs. When some code asks the COM library to create an instance of a class that is implemented in a component in a different location, the instance is created in the remote location along with a stub. The code that asked for the instance is passed an interface pointer to a proxy object, created on its side. When an operation is invoked via this interface pointer, the proxy translates this to a remote procedure call (RPC) to the remote stub, which in turn invokes the corresponding operation on the real object. It may also be necessary to create a proxy-stub pair at other times than object instantiation. This happens when an interface pointer is passed as a parameter to an operation of an object in a remote location. This process is known as marshalling. Proxy and stub code is usually generated automatically from IDL specifications. Figure 3 illustrates the use of proxy and stub objects

The ability to deal with memory spaces may not be of great consequence to real-time systems, since real-time operating systems do not traditionally use memory spaces. The ability to deal with such may, however, be useful in parallel processor architectures. DCOM may be useful in simplifying the implementation of distributed real-time systems. The transparency to the programmer of accessing remote objects is not completely valid for real-

time systems, however. Since the timing of object operations will differ between local and remote invocations, real-time software developers will still need to consider whether their code uses components in another location or not. It is also useful for developers of components to be aware of whether their components will be remotely accessed. For instance, one may consider exploiting the ability to define asynchronous interfaces for such components. An additional benefit of using DCOM in real-time systems is that it may simplify the implementation of communication between these systems and COM-based desktop applications, such as operator stations.

In addition to the extra time overhead associated with remote invocation and marshalling, DCOM also requires more space than COM, to store the proxy and stub code as well as the RPC mechanism. The proxy and stub are generally quite small and executes relatively quickly, however, so the time and space overhead is mostly due to the RPC mechanism and underlying protocol stack. Therefore, using DCOM does not result in much of an overhead for distributed real-time systems, where RPC or some other communication mechanism would be needed anyway.

A possible reason for using DCOM in ABB's control system, is that protocol handlers could be located in the communication interfaces or I/O modules they support, rather than in the central processing unit. The usefulness of this is not obvious, however, especially when considering the required additional overhead. Thus, there are no current plans to adopt DCOM in the system.

3.4. The Next Generation: .NET

The name .NET is used by Microsoft to denote a comprehensive set of new technologies. This includes a new component model, intended to replace COM/DCOM. A notable development is that .NET moves the responsibility of providing certain functionality from the components to a more sophisticated run-time system. In particular, COM/DCOM requires components to provide a considerable amount of "house-keeping" functionality that is taken care of by the .NET run-time. Much of the flexibility that follows from having such implementations in each component is maintained under .NET, where the operation of the run-time system with respect to individual components can be affected by setting declarative attributes.

A potential advantage of this development is increased reliability, since it may be assumed that more effort may be invested in ensuring the quality of a run-time system to be re-used in a large number of systems. Another attractive consequence of having more code in a common run-time is that the total size of the software may decrease. Obviously, this advantage of grows with the number of components in the system. On the other hand, using a sophisticated run-time system, possibly without using much of its functionality, may lead to unnecessarily large software. This is a particular problem for resource constrained embedded systems. Fortunately, Microsoft has defined a special compact version of .NET that limits this problem somewhat. What is assumed to be the greatest strength of .NET is the potential for increased development productivity. This relies both on the aforementioned run-time system with its associated libraries, and on advanced development tools. As usual, this gain is achieved at the expense of some run-time overhead. While it seams clear that this cost is acceptable for desktop software, the corresponding question for real-time systems is more open.

4. Related Work

There are some work on software component models and real-time or embedded systems in recent literature. This work is dominated by efforts to define component models particularly targeted at real-time embedded systems or even narrower application domains. Examples include Philip's Koala component model for consumer electronics [11], the component model for industrial field devices developed in the PECOS project [12], the commercial product ControllShell [13], which is based on visual composition and automatic code generation, and the more academic ACCORD approach [14] of aspect-oriented component-based development of real-time systems. Work on using "mainstream" component models in real-time systems is less common. One example is [15], which also discusses COM. This work, however, focuses on extensions to COM rather than the consequences of using the existing model in real-time systems.

5. Conclusion and Future Work

This paper has discussed the idea of using a software component model in real-time systems. In particular, using Microsoft models, both from the perspective of real-time systems in general and from that of ABB's control system. In general, it has been seen that each of the levels of adopting the models that have been discussed, introduces some degree of time and space overhead. In addition, new potential sources of run-time errors are introduced, corresponding to compilation errors in traditional object-oriented programming. It is concluded that COM/DCOM may be used for real-time systems, provided that any overhead is acceptable or can be compensated by hardware, and that the software designer takes care that the potential run-time errors are not allowed to materialize and result in a loss of predictability.

The major conclusions to be drawn from the discussions in this paper are as follows. COM interfaces, which provide a way to separate the specification of interfaces from component implementation, carry with them a very modest time and memory overhead. Compared to interface inheritance in object-oriented programming, COM interfaces introduce a potential source of run-time errors. COM's mechanism for instantiating objects and loading code at run-time has a considerable overhead when compared to instantiation in for example C++. This overhead is subject to a certain variability, which may be avoided by careful application design. DCOM is an extension of COM that allows applications to access COM objects across memory spaces and physical machine boundaries. The time and space overhead associated with this is dominated by the underlying communication mechanisms. The new .NET platform promises increased development productivity, but it remains to be seen to what extent it is suitable for real-time systems.

The immediate future work planned as a continuation of this paper is to strengthen the analyses with empirical evidence by conducting experiments and collecting measurements. Preferably, this should be done on a real-time platform using a commercial or self-made COM implementation. In the longer perspective, an intriguing idea is to develop a COM-based component model particularly intended for real-time software. This idea is primarily inspired by how COM+ supports the implementation of functionality such as transactional processing, which is considered a major challenge in distributed information systems. Corresponding challenges for real-time systems include issues such as concurrency, synchronization, and timing. In addition to easing the implementation it would be desirable for such a model to support compositional reasoning, i.e. the deduction of a system's properties from the known properties of its parts. A natural starting point for achieving this is the existing work on prediction enabled component technologies (PECT).

6. References

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

[2] Sun Microsystems, *JavaBeans Specification*, Version 1.01, 1997.

[3] D. Chappell, *Understanding ActiveX and OLE*, Microsoft Press, 1996.

[4] Microsoft Corporation, *The Component Object Model Specification*, v0.99, 1996.

[5] Sun Microsystems, *Enterprise JavaBeans Specification*, Version 2.0, 2001.

[6] D. S. Platt, *Understanding COM+*, Microsoft Press, 1999.

[7] Object Management Group, *CORBA Components*, Version 3.0, Report No. formal/02-06-65, 2002.

[8] J. Lowy, *Programming .NET Components*, O'Reilly & Associates, 2003.

[9] F. Lüders, *Use of Component-Based Software Architectures in Industrial Control Systems*, Technology Licentiate Thesis, Mälardalen University, Sweden, 2003.

[10] F. E. Redmond III, *DCOM – Microsoft Distributed Component Object Model*, IDG Books, 1997.

[11] R. van Ommering, J. Kramer, J. Magee, "The Koala Component Model for Consumer Electronics Software", *IEEE Computer*, March 2000, Vol. 33, No. 3.

[12] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, R. van den Born, "A Component Model for Field Devices", *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment*, June 2002.

[13] 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.

[14] A. Tešanović, D. Nyström, J. Hansson, C. Norström, "Towards Aspectual Component-Based Development of Real-Time Systems", *Proceedings of ölsakfjölskdjf alskdfj* 2001?.

[15] D. Chen, A. Mok, M. Nixon, "Real-Time Support in COM", *Proceedings of the 32nd Hawaii International Conference on System Sciences*, 1999.

[16] S. A. Hissam, G. A. Moreno, J. Stafford, K. C. Wallnau, "Enabling Predictable Assembly", *Journal of Systems and Software*, Volume 65, Issue 3, 2003.