The Journal of Systems and Software 61 (2002) 201–212

The Journal of Systems and Software

www.elsevier.com/locate/jss

Challenges of component-based development

Ivica Crnkovic[a,*], Magnus Larsson[b]

[a] Department of Computer Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden
[b] Research and Development, ABB Automation Products AB, 721 59 Västerås, Sweden

Received 1 March 2001; received in revised form 1 July 2001; accepted 1 September 2001

Abstract

It is generally understood that building software systems with components has many advantages but the difficulties of this approach should not be ignored. System evolution, maintenance, migration and compatibilities are some of the challenges met with when developing a component-based software system. Since most systems evolve over time, components must be maintained or replaced. The evolution of requirements affects not only specific system functions and particular components but also component-based architecture on all levels. Increased complexity is a consequence of different components and systems having different life cycles. In component-based systems it is easier to replace part of system with a commercial component. This process is however not straightforward and different factors such as requirements management, marketing issues, etc., must be taken into consideration. In this paper we discuss the issues and challenges encountered when developing and using an evolving component-based software system. An industrial control system has been used as a case study. © 2002 Elsevier Science Inc. All rights reserved.

Keywords: Reuse; Component-based development; Development environment; Architecture; Commercial components

1. Introduction

Systems that live over a longer period of time tend to be updated and changed many times during this period. Reuse and an open component-based architecture are the key to the success of systems with a long life cycle. Designing a system that supports this approach requires more effort in the design phase and the time to market might be longer, but in the long run, the reusable architecture will prove profitable. The reuse concept can be used on different levels: On a low level it is a reuse of source-code, and small-size components. More reuse is obtained with larger components encapsulating business functions. Finally, the integration of complete products in complex systems can be seen as the highest level of reuse. On each level of reuse there are specific demands on the reusable components, on the component management and on the integration process.

This paper describes important issues related to the development and maintenance of reusable components and as an example uses the ABB Advant industrial process control system. In Section 2 we give an overview of the Advant system design and the main characteristics of Advant reusable components. Section 3 outlines all the development and maintenance aspects of a component-based system which must comply with customer requirements. During evolution of the system new technologies were developed which resulted in the appearance on the market of many components with the same functionality as the proprietary ones. The fact that new components must be incorporated into the existing systems introduces new demands on the system development process. These new issues are discussed in Section 4.

2. Case study of an industrial automation system

2.1. Overview

ABB is a global electrical engineering and technology company, serving customers in power generation, transmission and distribution, in industrial automation products, etc. The ABB group is divided into companies, one of which, ABB Automation Products AB, is

* Corresponding author. Tel.: +46-21-103183; fax: +46-21-342666.
E-mail addresses: ivica.crnkovic@mdh.se (I. Crnkovic), magnus.larsson@mdh.se (M. Larsson).

responsible for the development of industrial automation products. The automation products encompass several families of industrial process-control systems including both software and hardware.

The main characteristics of these products are reliability, high quality and compatibility. These features are the result of responses to the main customers requirements: The customers require stable products, running around the clock, year after year, which can be easily upgraded without impact on the existing process. To achieve this, ABB uses a component-based system approach to design extendable and flexible systems.

The Advant open control system (OCS) (ABB, 2000) is component-based to suit different industrial applications. The range includes systems for Power Utilities, Power Plants and Infrastructure, Pulp and Paper, Metals and Minerals, Petroleum, Chemical and Consumer Industries, Transportation systems, etc. An overview of the Advant system is shown in Fig. 1.

Advant OCS performs process control and provides business information by assembling a system of different families of Advant products. Process information is managed at the level of process controllers. The process controllers are based on a real-time operating system and execute the control loops. The operator station (OS) and information management station (IMS) gather and supervise product information, while the business system provides analysis information for optimization of the entire processes. Advant products use standard and proprietary communication protocols to satisfy real-time requirements.

Advant OCS therefore includes information management functions with real-time insight into all aspects of the process controlled. Advant Information Management has an SQL-based relational database accessible to resident software and all connected computers. Historical data acquisition reports, versatile calculation packages and an application programming interface (API) for proprietary and third-party applications are examples of the functionality provided. Advant components have access to process, production and quality data from any process control unit in a plant or in an Intranet domain.

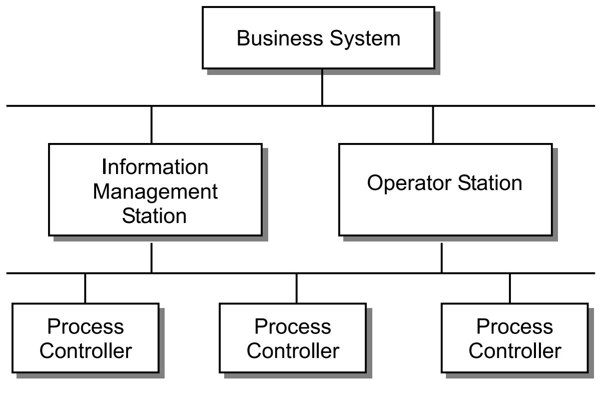

Fig. 1. An overview of the conceptual architecture of the Advant open control system.

2.2. Designing with reuse

Designing with reuse of existing components has many advantages (Sommerville, 1996). The software development time can be reduced and the reliability of the products increased. These were important prerequisites for the Advant OCS development.

Advant OCS products can be assembled in many different configurations for use in various branches of industry. Specific systems are designed with the reuse of Advant OCS products and other external products. This means customers get a tailor-made system that meets their needs. External products and components can be used together with the Advant OCS due to the openness of the system. For example, a satellite communication component, which is used to transmit data from the offshore station to the supervision system inland, can be integrated with the Advant OCS.

The Advant system architecture is designed for reuse. Different products such as Operator and Information Management Stations are used as system components in assembling complete systems. The two operator station versions, Master OS and MOD OS, are used in building different types of operator applications.

2.3. Scalability

Advant OCS can be configured in a multitude of ways, depending on the size and complexity of the process. The initial investment can consist of stand-alone process controllers and, optionally, local operator stations for control and supervision of separate machines and process sections. Subsequently, several process controllers can be interconnected and together with central operator and information management stations build up a control network. Several control networks can be interconnected to give a complete plant network which can share centrally located operator, information and engineering workplaces.

2.4. Openness

The system is further strengthened by the flexibility to add special hardware and software for specific applications such as weighing, fixed- and variable-speed motor drives, safety systems and product quality measurements and control in, for example, the paper industry. Second- and third-party administrative, information, and control can also be easily incorporated.

2.5. Cost-effectiveness

The step-by-step expansion capability of Advant OCS allows users to add new functionality without making existing equipment obsolete. The system's self-configuration capability eliminates the need for engineers to enter or edit topology descriptions when new stations are physically installed. New units can be added while the system is in full operation. With Advant OCS, system expansion is therefore easy and cost-effective.

2.6. Reusable components

The Advant OCS products are component-based to minimize the cost of maintenance and development. Fig. 2 shows the component architecture of the operator station assembled from components.

The operator station consists of a specific number of functional components and a set of standard Advant components. These components use the user interface system (UIS) component. Object management facility (OMF) is a component which handles the infrastructure and data management. OMF is similar to CORBA (OMG, 2000) in that it provides a distributed object model with data, operation and event services. The UxBase component provides drivers and other specific operating system functions. Helper classes for strings, lists, pointers, maps and other general-purpose classes are available in the C++_complib library component.

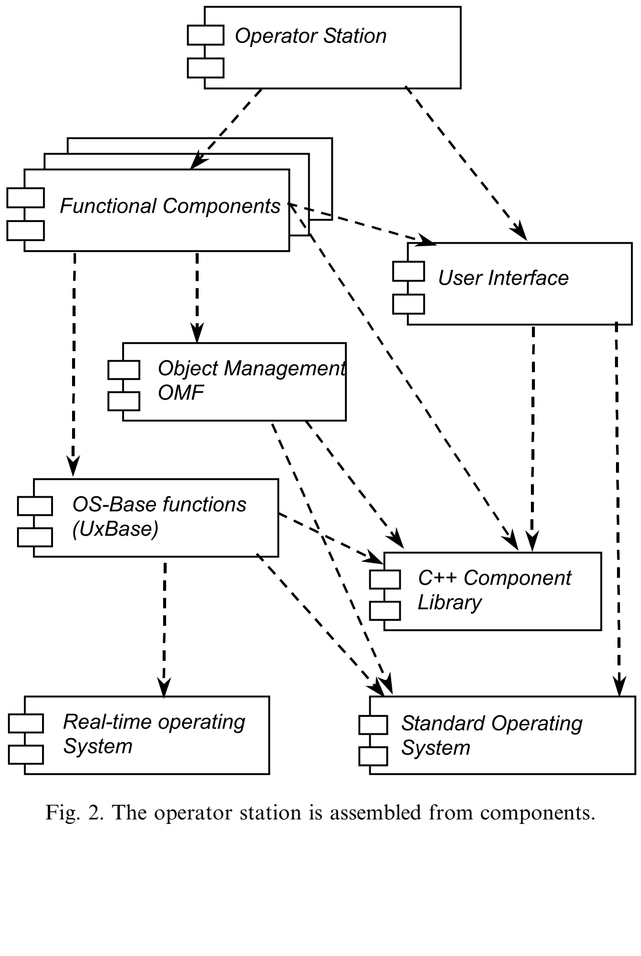

Fig. 2. The operator station is assembled from components.

The components are built upon operating systems, one, a standard system (such as Unix or Windows), and the other, a proprietary real-time system.

To illustrate different aspects of component-based development and maintenance, we shall further look at two components:

- OMF, a business type of component with a high level of functionality and a complex internal structure.
- C++_complib is a basic and a very general library component.

2.7. OMF

OMF (Nübling et al., 1999) is object-oriented middleware for industrial process automation. It encapsulates real-time process control entities of almost every conceivable description into objects that can be accessed from applications running on different platforms, for example, Unix and Windows NT. Programming interfaces are available for many languages, such as C, C++, Visual Basic, Java, Smalltalk and SQL while interfaces to the IEC 1131-3 (IEC, 1992) process control languages are under development. OMF is also adapted to Microsoft component object model (COM) via adapters and another component called OMF COM aware. The adapters for OPC (OLE for process control) (OPC, 1998) and OLE automation are also implemented. Thanks to all these software interfaces, OMF makes process and production data available to the majority of computer programmers and users, i.e., even to those not necessarily involved in the industrial control field. For instance, it is easy to develop applications in Microsoft Word, Excel and Access to access process information. OMF has been developed for demanding real-time applications, and incorporates features such as real-time response, asynchronous communications, standing queries and priority scheduling of data transfers. On one side OMF provides industry-standard interfaces to software applications, and on the other, it offers interfaces to many important communication protocols in the field including MasterNet, MOD DCN, TCP/IP and Fieldbus Foundation. These adapters make it possible to build homogeneous control systems out of heterogeneous field equipment and disparate system nodes.

OMF reduces the time and cost of software development by providing frameworks and tools for a wide range of platforms and environments. These utilities are well integrated into their respective surroundings, allowing developers to retain the tools and utilities they prefer to work with.

2.8. C++_complib

C++_complib is a class library that contains general-purpose classes such as containers, string management classes, file management classes, etc. The C++_complib library was developed when no standard libraries, such as STL (Austern, 1999), were available on the market. The main purpose of this library was to improve the efficiency and quality, and promote the uniform usage of the basic functions.

C++_complib is not a component according to the definition in Szyperski (1998), where a component is a unit of composition deployed independently of the product. However, in a development process C++_complib is treated in a very similar way as binary components with some restrictions such as dynamic configuration.

2.9. Experience

The Advant system is a successful system and the main reasons for its success are its component-based architecture giving flexibility, robustness, stability and compatibility, and effective build and integration procedures. This type of architecture is similar to product line architectures (Bass et al., 1999). Some case studies (Bosch, 1999) have shown that product-line architectures are successfully applied in small- and medium-sized enterprises although there exist a number of problems and challenging issues (organization, training, information distribution, product variants, etc.). The Advant experience shows that applying of product-line architectures can be successful for large organizations.

However, the cost of achieving these features has been high. To suit the requirements of an open system, new ABB products have always to be backward compatible. It would have been easier to develop a new system that did not require being compatible with the previous systems. A guarantee that the system is backward compatible is a warranty that an existing system will work with new products and this makes the system trustworthy.

Development with large components which are easy to reuse increases the efficiency significantly as compared with reusing a smaller component that could have been developed in-house at the same cost as its purchase price. Advant OCS products are examples of large components which have been used to assemble process automation systems.

3. Different reuse challenges

3.1. Component generality and efficiency

Reuse principles place high demands on reusable components. The components must be sufficiently general to cover the different aspects of their use. At the same time they must be concrete and simple enough to serve a particular requirement in an efficient way. Developing a reusable component requires three to four times more resources than developing a component, which serves a particular case (Szyperski, 1998). The fact that the requirements of the components are usually incomplete and not well understood (Sommerville, 1996) brings additional level of complexity. In the case of C++_complib, the situation was simpler, because the functional requirements were clear. It was relatively easy to define the interface, which was used by different components in the same way. The situation was more complicated with complex components such as OMF. Although the basic concept of component functionality was clear, the demands on the component interface and behavior were different in different components and products. Some components required a high level of abstraction, while others required the interface to be on a more detailed level. These different types of requirements have led to the creation of two levels of components: OMF base, including all low-level functions, and OMF framework, containing only a higher level of functions and with more pre-defined behavior and less flexibility. In general, requirements for generality and efficiency at the same time lead to the implementation of several variants of components which can be used on a different abstraction level. In some specific cases, a particular solution must be provided. This type of solution is usually beyond the object-oriented mechanisms, since such components are on the higher abstraction level.

3.2. System evolution

Long-life products are most often affected by evolution of different kinds:

- *Evolution of system requirements, functional and non-functional.* A consequence of a continually competitive market situation is a demand for continually improved system performance. The systems controlling and servicing business, industrial, and other processes should permanently increase the efficiency of these processes, improve the quality of the products, minimize the production and maintenance costs, etc.
- *Evolution of technology related to different domains.* The advance of technology in the different fields in which software is used requires improved software. The improvements may require a completely new approach to or new functions in software.
- *Evolution of technology used in software products.* Evolution in computer hardware and software technology is so fast that an organization manufacturing long-life and complex products must expect significant technology changes during the product life cycle. From the reliability and risk point of view, such organizations prefer not to use the latest technology, but because of the demands of a highly competitive mar-

ket, are forced to adopt new technology as it appears. The often unpredictable changes which must be made in products cause delivery delays and increased production costs.

- *Evolution of technology used for the product development*. As in the case of products themselves, new technology and tools used in the development process appear frequently on the market. Manufacturers are faced with a dilemma – to adopt the new technology and possibly improve the development process at the risk of short-term higher costs (for training and migration) or to continue using the existing technology and thereby miss an opportunity to lower development costs in the long run.
- *Evolution of society*. Changes in society (for example, environmental requirements or changes in the relations between countries – as in the EU) can have a considerable impact on the demands on products (for example, new standards, new currency, etc.) and on the development process (relations between employers and employees, working hours, etc.).
- *Business changes*. We face changes in government policies, business integration processes, deregulation, etc. These changes have an impact on the nature of business, resulting, for example, in a preference for short-term planning rather than long-term planning and more stringent time-to-market requirements.
- *Organizational changes*. Changes in society and business have direct effect on business organizations. We can see a globalization process, more abrupt changes in business operations and a demand for more flexible structures and management procedures, "just-in-time" deliveries of resources, services and skills. These changes require another, fast and flexible approach to the development process.

All these changes have a direct or indirect impact on the product life cycle. The ability to adapt to these changes becomes the crucial factor in achieving business success (Brown, 2000). In the following sections we discuss some of these changes and their consequences in the development process and product life cycle.

3.3. Evolution of functional requirements

The development of reusable components would be easier if functional requirements did not evolve during the time of development. As a result of new requirements for the products, new requirements for the components will be defined. The more reusable a component is, the more demands are placed on it. A number of requirements coming from different products may be the same or very similar, but this is not necessarily the case for all requirements passed to the components. This means that the number of requirements of reusable components grows faster than that of particular products or of a non-reusable piece of software. The relation between component requirements and the requirements from the products is expressed with the following equation:

$$R_C = R_{C0} + \sum a_i R_{p_i}, \quad 0 \leqslant a_i \leqslant 1,$$

where R_{C0} denotes direct requirements of the component, R_{p_i} requirements of the products P_i, a_i impact factors to the component and R_C is the total number of component requirements.

To satisfy these requirements the components must be updated more rapidly and the new versions must be released more frequently than the products using them.

The process of the change of components is more dynamic in the early stage of the component lives. In that stage the components are less general and cannot respond to the new requirements of the products without being changed. In later stages, their generality and adaptability increase, and the impact of the product requirements become less significant. In this period the products benefit from combinatorial and synergy effects of components reuse. In the last stage of its life, the components are getting out-of-date, until they finally become obsolete, because of different reasons: Introduction of new techniques, new development and run-time platforms, new development paradigms, new standards, etc. There is also a higher risk that the initial component cohesion degenerates when adding many changes, which in turn requires more efforts.

This process is illustrated in Fig. 3. The first graph shows the growing number of requirements for certain products and for a component being used by these products. The number of requirements of a common component grows faster in the beginning, saturates in the period $[t_0 - t_1]$, and grows again when the component features become inadequate. Some of the product requirements are satisfied with new releases of products and components, which are shown as steps on the second graph. The component implements the requirements by its releases, which normally precede the releases of the product if the requirements originated from the product requirements.

Indeed this was the case with both components we are analyzing here: New functions and classes were required from C++_complib, and new adapters and protocol support were required from OMF. The development time for these components was significantly shorter than for products: While new versions of a product are typically released every six months, new versions of components are released as least twice as often. After several years of intensive development and improvement, the components became more stable and required less effort for new changes. In that period the frequency of the releases has been lowered, and especially the effort has been significantly lower.

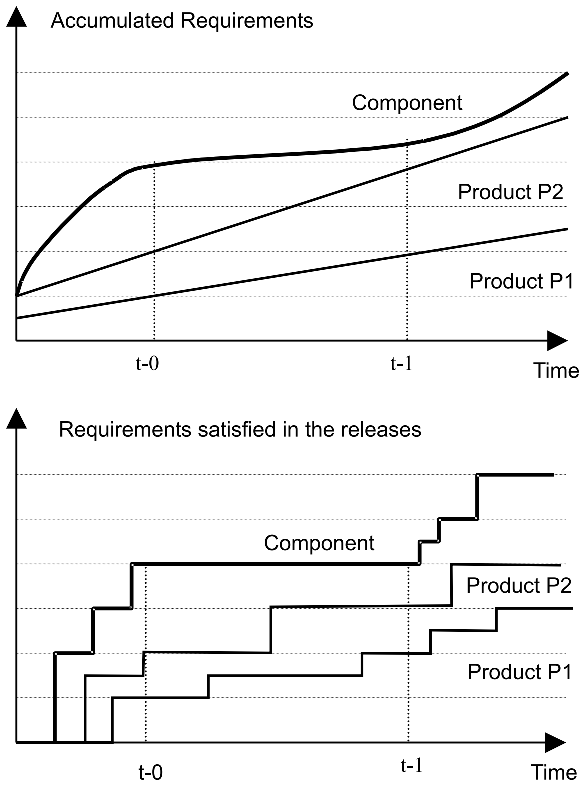

Fig. 3. To satisfy the requirements the reusable component must be modified more often in the beginning of their life.

New efforts for further development of components appeared with migration of products on different platforms and newer platform versions. Although the functions of the products and components did not change significantly a considerable amount of work was done on the component level.

3.4. Migration between different platforms

During their several years of development, Advant products have been ported to different platforms. The reasons for this were the customer requirement that the products should run on specific platforms, and general trends in the growing popularity of certain operating systems. Of course, at the same time, new versions and variants of the platform already used appeared, supporting new, better and cheaper hardware. The Advant products have migrated through different platforms: Starting on Unix HP-UX 8.x and continuing through new releases (HP-UX 9.x, 10.x), they have been ported to other Unix platforms such as Digital Unix, and also to completely different platforms such as Open VMS and Windows NT family (NT 3.5, NT 4.0 and Windows 2000). The products have been developed and maintained in parallel. The challenge with this multi-platform development was to keep the compatibility between the different variants of the products, and to maintain and improve them with the minimal efforts.

As an important part of the reuse concept was to keep the high-level components unchanged as far as possible it was decided to encapsulate the differences between operating systems in low-level components. This concept works, however, only to some extent. The minimal activity required for each platform is to rebuild the system for that platform. To make it possible to rebuild the software on every platform, standard-programming languages C and C++ have been used. Unfortunately, different implementations of the C++ standard in different compilers caused problems in the code interpretation and required the rewriting of certain parts of the code. To ensure that standard system services are available on all platforms, the POSIX standard has been used. POSIX worked quite well on different Unix platforms, but much less so on Windows NT. The second level of compatibility problem was graphical user interface (GUI). The main dilemma was whether to use exactly the same GUI on every platform or to use the standard "look and feel" GUI for each platform. This question applied particularly on NT in relation to Unix platforms. Experience has shown that it is not possible to give a definitive answer. In some cases it was possible to use the same GUI and the same graphical packages, but in general, different GUIs were implemented.

The main work regarding the reuse of code on different platforms was performed on low-level components such as UxBase and OMF. While UxBase provides different low-level packages for every platform (for example, different drivers), OMF capsulated the differences directly in the code using conditional compilation. OMF itself is designed in such a way that it was possible to divide the code into two layers. One layer is specific for each operating system, and the other layer, with the business logic, is implemented for all of the supported platforms. Reuse issues on different platforms for C++_complib were easier, strictly the package contains general algorithms, which are not depending on specific operating system. Some problems appeared, however, related to different characteristics of compilers on different platforms.

3.5. Compatibility

One of the most important factors for successful reusability is the compatibility between different versions of the components. A component can be replaced easily or added to new parts of a system if it is compatible with its previous version. The compatibility requirements are essential for Advant products, since smooth upgrading of systems, running for many years, is required. Compatibility issues are relatively simple when changes introduced in the products are of maintenance and improvement nature only. Using appropriate test plans, including regression tests, functional compatibility can be tested to a reasonable extent. More complicated problems occur when new changes introduced in a reusable component eliminate the compatibility. In such a case, additional software, which can manage both versions, must be written.

A typical example of such an incompatible change is a change in the communication protocol between OMF clients and servers. All different versions of OMF must be able to talk to each other to make the system flexible and open. It is possible to have different combinations of operating systems and versions of OMF and it still works. This has been solved with an algorithm that ensures the transmission of correct data format. If two OMF nodes have the same version, they talk in their native protocol.

If an old OMF node talks with a new, the new OMF is responsible for converting the data to the new format, this being designated RMIR ("receiver makes it right"). If a new OMF sends data to an older, the older OMF cannot convert the data since it is unaware of the new protocol. In this case the newer OMF must send in the old protocol format, SMIR ("sender makes it right"). This algorithm builds on that fact all machines know about each other and that they also know what protocol they talk. However, if an OMF-based node does not know of the other node then it can always send in a predefined protocol referred to as "well known format". All nodes do recognize this protocol and can translate from it. This algorithm minimizes the number of data conversions between the nodes.

In the case of C++_complib the problems with compatibility were somewhat different. New demands on the same classes and functions appeared because of new standards and technology. One example is the use of C++ templates. When the template technology became sufficiently mature, the new requirements were placed for C++_complib: All the classes were to be reimplemented as template classes. The reason for this was the requirement for using basic classes in a more general and efficient way. Another example is Unicode support in addition to ASCII-support. These new functions were added by new member-functions in the existing classes and by adding new classes using the inheritance mechanism for reusing the already existing classes. The introduction of the same functions in different formats have led to additional efforts in reusing them. In most of the cases the old format has been replaced by the new one, with the help of simple tools built just for this purpose. In some other cases, due to non-proper planning and prioritizing the time-to-market requirements, both old and new formats have been used in the same source modules which have led to lower maintainability and to some extent to lower quality of the products.

3.6. Development environment

When developing reusable components several dimensions of the development process must be considered:

- Support for development of components on different platforms.
- Support for development of different variants of components for different products.
- Support for development and maintenance of different versions of components for different product versions.
- Independent development of components and products.

To cope with these types of problems, it is not sufficient to have appropriate product architecture and component design. Development environment support is also essential. The development environment must permit an efficient work in the project – editing, compiling, building, debugging and testing. Parallel and distributed development must also be supported, because the same components are to be developed and maintained at the same time on different platforms. This requires the use of a powerful configuration management (CM) tool, and definition of an advanced CM process.

The CM process support exists on two levels. First on the source-code level, where source-code files are under version management and binary files are built. The second level is the product integration phase. The product built must contain a consistent set of component versions. For example, Fig. 4 shows an inconsistent set of components. The product version P1-V2 uses the component versions C1-V2 and C2-V2. At the same time the component version C1-V2 uses the component version C2-V1, an older version. Integrating different

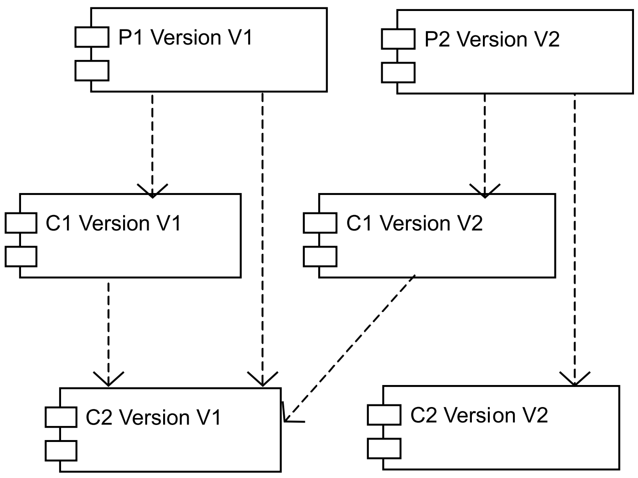

Fig. 4. An example of inconsistent component integration.

versions of the same component may cause unpredictable behavior of the product.

Another important aspect of CM in developing reusable components is change management. Change management keeps track of changes on the logical level, for example, error reports, and manages their relations with implemented physical changes (i.e., changes of documentation, source code, etc.). Because change requests (for example, functional requirements or error reports) come from different products, it is important to register information about the source of change requests. It is also important to relate a change request from one product to other products. The following questions must be answered: What impact can the implemented change have on other products? If an error appears in one product, does it appear in other products? Possible implications must be investigated, and if necessary, the users of the products concerned must be informed.

The development environment designated Software Development Environment (SDE) (Crnkovic, 1997) is used in developing Advant products. It is an internally built program package which encapsulates different tools, and provides support for parallel development. The CM tool, based on RCS (Tichy, 1985), provides support for all CM disciplines such as change management, works pace management, build management, etc. SDE runs on different platforms with slightly modified functions. For example, the build process is based on *Makefiles* and *autoconf* on Unix platforms, while Microsoft Developer Studio with additional *Project Settings* is used on Windows NT. The main objective of SDE is to keep the source-code in one place under version control. Different versions of components are managed using baselines, and change requests. Change requests are also under version control, which gives a possibility of acquiring information useful for project follow-up, for every change from registration to implementation and release (Crnkovic and Willför, 1998).

3.7. Independent component development

Component development independent of the products gives several advantages. The functions are broken down into smaller entities that are easier to construct, develop and maintain. The independent component development facilitates distributed development, which is common in large enterprises. Development of components independently of product or other component development also introduces a number of problems. The component and product test become more difficult. On the component level, a proper test environment must be built, which often must include a number of other components or even maybe the entire product.

Another problem is the integration and configuration problem. A situation shown in Fig. 4 must be avoided. When it is about complex products, it is impossible to manually track dependencies between the components, but a tool support for checking consistency must exist.

In the Advant development the components were treated as separate products even if they were developed within the enterprise. To have this approach helped when third-party components were used they were all managed in a uniform way. Every component contained a file called import file that included a specification of all component versions used to build the component. When the final product was assembled from the components, the import file has been used for integration and checking if the consistent sets of the components have been selected. The development environment, based on *make*, was set up to use the import files and the common product structure. All released components were stored in the product structure for availability to others. Another structure was used during development of a component. The component was exported to the product structure when the development was finished. Using this approach it was shown that the architecture design plays a crucial role. A good architecture with clear and distinguished relations between components facilitates the development process.

The whole development process is complex and requires organized and planned support, which is essential for efficient and successful development of reusable components and of applications using these.

3.8. The maintenance process

The maintenance process is also complex, because it must be handled on different levels: On the system level, where customers report their problems, on the product level, where errors detected in a specific product version are reported, and finally on the component level, where the fault is located. The modification of the component can have an impact on other components and other products, which can lead to an explosion of new versions of different products which already exist in several versions. To minimize this cumbersome process, ABB adopted a policy of avoiding the generation of and supply of specific patches to selected customers. Instead, revised products incorporating sets of patches were generated and delivered to all customers with maintenance contracts, to keep customer installations consistent.

The relations between components, products and systems must be carefully registered to make possible the tracing of errors on all levels. A systematic use of software configuration management has a crucial role in the maintenance process.

To support the maintenance process, Advant products and component specifications together with error reports are stored in several classes of repositories (see Fig. 5).

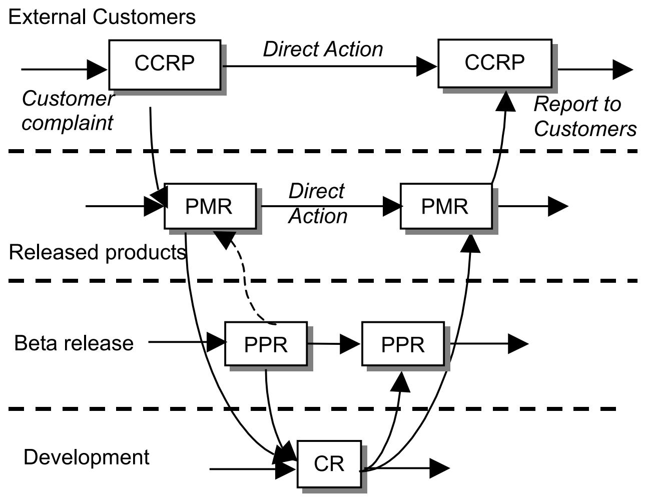

Fig. 5. Different levels of error report management.

On the highest level, the repository managing customer reports (CCRP) makes it possible for service personnel to provide customers with prompt support. Information saved on this level is customer- and product-oriented. Reports indicating a product problem are registered in the product maintenance report repository (PMR) where all known problems related to products and components are filed. Also, product structure information is stored on this level. The product structure, showing dependencies between products and components, provides product and component developers with assistance in relating error reports to the source of the problem on both product and component levels. A similar error management process is defined for products in the beta phase, i.e., not yet released. All of the problems identified in this phase (typically by test groups) are registered in the form of pre-release problem reports (PPR). These problems are either solved before the product is released or are reclassified as product error reports and saved in PMR. Any change applied in code or documentation is under change control, and each change is initiated by a change request. If a change required comes from an error report, a change request will be generated from a PMR. When a change made in a component is tested and verified, the action description is exported to the correlating PMR, propagated to the products involved and finally returned to the customer via the CCRP repository.

This procedure is not unique to component-based development. It is a means of managing complex products and of maintaining many products. What is specific to the component-based approach is the mapping between products and components and the management of error reports on product and component level, the most difficult part of the management. In this case the entire procedure is localized on the PMR level, i.e., product level. On the customer side, information with the highest priority is related to products and customers. On the development level, all changes registered are related primarily to components. Information about both products and components is stored on the development level. Error management on this level is the most complex. An error may be detected in a specific product version, but may also be present in other products and other product versions. The error may be discovered in one component, but it can be present in different versions of that component. The problem can be solved in one component version, but it also may be necessary to solve it in several. The revised component versions are eventually subsequently integrated into new versions of one or several products. This multi-dimensional problem (many error reports, impact on different versions of components and products, the solution included in different components and product versions) is only partially managed automatically, as many steps in the process require direct human decisions (for example, a decision if a solution to a problem will or will not be included in the next product release). Although the whole procedure is carefully designed and rigorously followed, it has happened on occasions that unexpected changes have been included, and that changes intended for inclusion were absent from new product releases. For more details of the entire maintenance process see Kajko-Mattson (1999a) and Kajko-Mattson (1999b).

Another important subject is the maintenance of external components. It has been shown that external components must be treated in the same way as internal components. All known errors and the complete error management process for internal and external components are treated in a similar way. The list of known, and corrected errors in external components is important for developers, product managers and service people. The cost of maintaining components, even those maintained by others, must be taken into consideration.

4. Integrating standard components

In recent years the demands of customers on systems have changed. Customers require integration with standard technologies and the use of standard applications in the products they buy. This is a definite trend on the market but there is little awareness of the possible problems involved. An improper use of standard components can cause severe problems, especially in distributed real-time and safety-critical systems, with long-period guarantees. In addition to these new requirements, time-to-market demands have become a very important factor.

These factors and other changes in software and hardware technology (Aoyama, 1998) have introduced a new paradigm in the development process. The development process is now focused on the use of standard and de facto standard components, outsourcing, COTS and the production of components. At the same time, final products are no longer closed, monolith systems, but are instead component-based products that can be integrated with other products available on the market.

This new paradigm in the development process and marketing strategy has introduced new problems and raised new questions (McKinney, 1999):

- The development process has been changed. Developers are now not only designers and programmers, they are also integrators and marketing investigators. Are the new development methods established? Are the developers properly educated?
- What are the criteria for the selection of a component? How can we guarantee that a standard component fulfills the product requirements?
- What are the maintenance aspects? Who is responsible for the maintenance? What can be expected of the updating and upgrading of components? How can we satisfy the compatibility and reliability requirements?
- What is the trend on the market? What can we expect to buy not only today but also on the day we begin delivering our product?
- When developing a component, how can we guarantee that the "proper" standard is used? Which standard will be valid in 5, 10 years?

All these questions must be considered before beginning a component-based development project. Josefsson (1999) presents certain recommendations to the component integrator for use as guidelines: Test the imported component in the environment where it is to run and limit the practical number of component suppliers to minimize the compatibility problems. Make sure that the supplier is evaluated before a long-term agreement is signed.

The focus of development environment support should be transferred from the "edit-build-test" cycle to the "component integration-test" cycle. Configuration management must give more consideration to run-time phase (Larsson and Crnkovic, 1999).

4.1. Replacing internal components with standard components

In the middle of 1980s, ABB Advant products were completely proprietary systems with internally developed hardware, basic and application software. In the beginning of 1990s, standard hardware components and software platforms were purchased while the real-time additions and application software were developed internally. The system is now developed further using components based on new, standard technologies.

During this development, further new components become available on the market. ABB faced this issue more than once. At one point in time, it was necessary to abandon the existing solutions in favor of new solutions based on existing components and technologies. To illustrate the migration process we discuss the possibility of replacing OMF and C++_complib with standard components.

Experience from these examples showed that it is easier to replace a component if the replacement process is made in small incremental steps. Allowing the new component to coexist with the old one makes it easier to be backward compatible and the change will be smooth.

4.2. Replacing OMF with DCOM

Moving from a UNIX-based system to a system based on Windows NT had serious effect on the system architecture. Microsoft components using a new object model were available, namely COM/DCOM (Box, 1998). DCOM has functionality similar to that of OMF and this became a new issue when DCOM was released. Should ABB continue to develop its proprietary OMF or change to a new standard component? The problem was that DCOM did not have all the functionality of OMF and vice versa. The domains overlap only partially.

A subscription of data with various capabilities can be made in OMF, and this subscription functionality is not supported by DOCM. On the other hand, DCOM can create objects when they are required and not like OMF where objects are created before the actual use of them. Both technologies support object communication and in this area it is easier to replace OMF with DCOM.

If the decision was made to continue with OMF, all the new components that run on top of COM could not be used, which would drastically reduce the possibilities of integration with other, third-party components. On the other hand, it would require considerable work to make the current system run on top of COM. This was the dilemma of COM versus OMF.

To begin with, OMF was adapted to COM with an adapter designated OMF COM aware. This functionality helped COM developers access OMF objects and vice versa. However, this solution to the problem using two different object models was not optimal since it added overhead in the communication. Nor was it possible to match the data types one to one, which made the solution limited. A decision was taken to build the new system on COM technologies with proprietary extensions adding the functions missing from COM. All the communication with the current system was to be made through the OMF COM. This solution made it easy to remove the old OMF and replace it with COM in small steps over time. Adapters are very useful when a new component is to used in parallel with an existing one (Rine et al., 1999). More adapters to other systems such as Orbix(CORBA) and Fieldbus Foundation were constructed. If the external systems have similar data types it is fairly straightforward to build a framework for adapters where the parts that take care of the proprietary system can be reused. New systems can be accessed by adding a server and client stub to the adapter framework. To be able to build functional adapters between two middleware components it is important to have the capability to create remote calls dynamically. For instance the dynamic invocation interface (DII) in CORBA can be used. If the middleware does not have this possibility it might be possible to generate code automatically that takes care of the different types of calls which are going to be placed through the adapter to the other system.

4.3. Replacing C++_complib with STL

To switch from C++_complib to STL (Austern, 1999) was much easier because STL covers almost all the C++_complib functions and provides additional functionality. Still, much work reminded to be done, since all the codes using C++_complib had to be changed to be able to use STL instead. The decision was taken to continue using both components and to use STL whenever a new functionality was added. After a time the use of old components was reduced and the internal maintenance cost reduced. In some cases in the same components both libraries were used, which gave some disadvantages, especially in the maintenance process.

4.4. Managing evolution of standard components

Use of standard components implies less control on them (Larsson and Crnkovic, 1999, 2000; Cook and Dage, 1999), especially if the components are updated at run-time. A system of components is usually configured once only during the build-time when known and tested versions of components are used. Later, when the system evolves with new versions of components, the system itself has no mechanism to detect if new components have been installed. There might be a check that the version of replacement component is at least the same as or newer than the original version. This approach prevents the system from using old components, but it does not guarantee its functionality when new components are installed. Applying ideas from configuration management, such as version and change management, in managing components is an approach which can be used to solve some of the problems.

A certain level of configuration control will be achieved when it is possible to identify components with their versions and dependencies on other components. Information about a system can be placed under version control for later retrieval. This makes it possible to compare different baselines of a system configuration. To manage dependencies, a graphic representation of the configuration is introduced. The graphs are then placed under version control. This information can be used to predict which components will be affected by a replacement or installation of a new component.

It is generally difficult to identify components during run-time and to obtain their version information. When the components are identified it is possible to build graphs of dependencies, which can be represented in various ways and placed under configuration control (Larsson, 2000).

To improve the control of external components, they can be placed under change management to permit the monitoring of changes and bugs. Instead of attaching source code files to change requests, which is common in change management, the name and version of the component can be used to track changes. When a problem report is analyzed, the outcome can be a change request for each component involved. Each such change request can contain a list of all the changed source files or a description of the patches if the component is external. Patches from the component vendor must be stored to permit recreation of the same configuration later. In cases where the high quality of products must be assured, the enterprise developing products must have special, well-defined relations to the component vendors for the support and maintenance.

5. Conclusion

We have presented the ABB Advant control systems (OCS) as a successful example of the development of a component-based system. The success of these systems on the market has been primarily the result of appropriate functionality and quality. Success in development, maintenance and continued improvement of the systems has been achieved by a careful architecture design, where the main principle is the reuse of components. The reuse orientation provides many advantages, but it also requires systematic approach in design planning, extensive development, support of a more complex maintenance process, and in general more consideration being given to components. It is not certain that an otherwise successful development organization can succeed in the development of reusable components or products based on reusable components. The more a reusable component is developed, the more complex is the development process, and more support is required from the organization.

Even when all these requirements are satisfied, it can happen that there are unpredictable extra costs. One example illustrate this: In the early stage of the ABB Advant OCS development, insufficient consideration was given to Windows NT and ABB had to pay the price for this oversight when it suddenly became clear that Windows NT would be the next operating platform. The new product versions on the new platform have been developed by porting the software from the old platform, but the costs were significantly greater than if the design had been done more independent of the first platform.

Another problem we have addressed is the question of moving to new technologies which require the recreation of the components or the inclusion of standard components available on the market. In both cases it can be difficult to keep or achieve the same functionality as the original components had. However, it seems that the process of replacing proprietary components by standard components available from third parties is inevitable and then it is important to have a proper strategy for migrating from old components to the new ones.

References

ABB, ABB Automation Products, Advant. Available from http://www.advantocs.com.

Aoyama, M., 1998. New age of software development: How component-based software engineering changes the way of software development. In: Proceedings of the 1st workshop on Component Based Software Engineering.

Austern, M., 1999. Generic Programming and STL. Addison-Wesley, Reading, MA.

Bass, L., Campbell, G., Clements, P., Northrop, L., Smith, D., 1999. Third product line practice report, Technical Report, CMU/SEI-99-TR.003, Software Engineering Institute.

Bosch, J., 1999. Product-line architectures in industry: A case study. In: Proceedings of 21st International Conference on Software Engineering. ACM Press, New York.

Box, D., 1998. Essential COM. Addison-Wesley, Reading, MA.

Brown, A., 2000. Large-scale Component-based Development. Prentice-Hall, Englewood Cliffs, NJ.

Cook, J.E., Dage, J.A., 1999. Highly reliable upgrading of components. In: Proceedings of 21st International Conference on Software Engineering. ACM Press, New York.

Crnkovic, I., 1997. Experience with Change-oriented SCM Tools. In: Proceedings of the 7th Symposium on Software Configuration Management. Lecture Notes in Computer Science, vol. 1235. Springer, Berlin.

Crnkovic, I., Willför, P., 1998. Change measurements in an SCM Process. In: Proceedings of the 8th Symposium on Software Configuration Management. Lecture Notes in Computer Science. Springer, Berlin.

IEC, 1992. Programmable Controllers Part 3, Programming Languages, IEC 1131.-3, IEC, Geneva.

Josefsson, M., 1999. Program varukomponenter i praktiken -attköpa tid och prestera mer, Report V0400.78, Sveriges Verkstadsindustrier.

Kajko-Mattson, M., 1999a. Maintenance at ABB (I): Software problem administration processes (the state of practice). In: Proceedings of IEEE International Conference on Software Maintenance. ACM Press, New York.

Kajko-Mattson, M., 1999b. Maintenance at ABB (II): Change execution processes (the state of practice). In: Proceedings of IEEE International Conference on Software Maintenance. ACM Press, New York.

Larsson, M., 2000. Applying configuration management techniques to component-based systems, Licentiate Thesis Dissertation 2000-007, Department of Information Technology, Uppsala University.

Larsson, M., Crnkovic, I., 1999. New challenges for configuration management. In: Proceedings of the 9th Symposium on System Configuration Management. Lecture Notes in Computer Science, vol. 1675. Springer, New York.

Larsson, M., Crnkovic, I., 2000. Component configuration management. In: Proceedings of the 5th Workshop on Component Oriented Programming.

McKinney, D., 1999. Impact of commercial off-the-shelf (COTS) software on the interface between systems and software engineering. In: Proceedings of the 21st International Conference on Software Engineering. ACM Press, New York.

Nübling, M., Popp, C., Zeidler, C., 1999. OMF – an object request broker for the process control application domain. In: Proceedings of the 3rd International Conference on Enterprise Distributed Object Computing EDOC. IEEE Computer Society, Silver Spring.

OMG, 2000. The common object request broker: Architecture and specification, Report v2.4, OMG Standards Collection, OMG.

OPC, 1998. OLE for process control, Report v1.0, OPC Standards Collection, OPC Foundation.

Rine, D., Nada, N., Jaber, K., 1999. Using adapters to reduce interaction complexity in reusable component-based software development. In: Proceedings of the 5th Symposium on Software Reusability. ACM Press, New York.

Sommerville, I., 1996. Software Engineering. Addison-Wesley, New York.

Szyperski, C., 1998. Component Software Beyond Object-Oriented Programming. Addison-Wesley, New York.

Tichy, W., 1985. RCS – A system for version control. IEEE Software and Practice Experiance 15 (7).

Ivica Crnkovic is a professor of Industrial Software Engineering at the Mälardalen University, Sweden. He received an M.Sc. in Computer Science 1979, an M.Sc. in Theoretical Physics 1984, and a Ph.D. in Computer Science 1991, all at the University of Zagreb, Croatia. He worked at ABB during 1985–1997, where he was responsible for software development environments. He was a project leader and manager of a group which developed Software Development Environment tools and methods for distributed development and maintenance of real-time systems. He is the Computer Science Laboratory leader at the Mälardalen University and he leads the Industrial IT research group at Mälardalen University. He is the co-organizer and a member of the program committee of several workshops related to Software Engineering and Configuration Management. His main research interests are Software Configuration Management, Component-based Development and in general Software Engineering.

Magnus Larsson is an industrial Ph.D. student employed by ABB Automation Products at the research and development department since 1993. He received a B.Sc. at Mälardalen University 1993 and an M.Sc. in computer science at Uppsala University 1995. He is interested in Component-based development, Software Configuration Management and real-time systems. He is a member of the Configuration Management group at the association of Swedish Engineering Industries. He presented the licentiate thesis "Applying Configuration Management Techniques to Component-based Systems" in December 2000.