Nationaal Lucht- en Ruimtevaartlaboratorium
National Aerospace Laboratory NLR

NLR-TP-99142

Component based software development at NLR

Assembling aerospace applications

E. Kesseler and E.H. Baalbergen

Nationaal Lucht- en Ruimtevaartlaboratorium
National Aerospace Laboratory NLR

NLR-TP-99142

Component based software development at NLR

Assembling aerospace applications

E. Kesseler and E.H. Baalbergen

This report is based on a presentation held at the Euroforum Software Congress "Component based development", Utrecht, The Netherlands, December 15, 1998.

Division:	Information and Communication Technology
Issued:	March 1999
Classification of title:	unclassified

Contents

(16 pages in total)

Abbreviations

AOM	Application Object Manager
CFD	Computational Fluid Dynamics
CORBA	Common Object Request Broker Architecture
COTS	Commercial Of The Shelf
CSCI	Computer Software Configuration Item
FAR	Federal Airworthiness Requirement
ILU	Inter-Language Unification
I/O	Input/Output
IMC	Instrument Meteorological Conditions
ISNaS	Information System for Navier-Stokes Equations
JAR	Joint Aviation Requirement
MC/DC	Modified Condition/Decision Coverage
NTSB	National Transport safety Board
OIL	Object Interface Layer
OMG	Object Management Group
ORB	Object Request Broker
TCD	Test Case Definition
VMC	Visual Meteorological Conditions
US	User Shell

1 Introduction

The classical approach to the production of software is to write a tailored solution dedicated to each application. For each new application the production process starts from scratch. This approach is becoming increasingly infeasible due to the commercial realities of unaffordable costs combined with an unaffordable time-to-market. Furthermore experience has shown that in general this labour intensive approach can not guarantee the application's quality, i.e. that the application performs its intended functions correctly. Re-use has been seen as a solution to some of these problems for a long time. Due to problems related with finding which modules to re-use, ambiguities in the documentation of the module's behaviour combined with an unspecified and hence unknown reliability of the module, re-use has not achieved the success expected from it. Component based development is the latest attempt to improve the software production process on all of the three items mentioned.

NLR operates in an aerospace environment, in which the application's technical characteristics pose additional constraints on the software. These constraints are reflected in the software development process. NLR's strategy to promote re-use and component based development will be illustrated with two cases. Of each case different characteristics will be discussed, exemplifying the variety of issues encountered in aerospace software development.

The following two chapters will describe one case each. The first case employs standard component based development techniques to produce an environment for metacomputing (SPINEware). The second case applies re-use for an embedded application which performs safety critical functions in an airframe. In this case safety concerns and certifiability are added. The conclusions in the last chapter summarise the experience gained in component based software development

2 Case 1: SPINEware

The first case concerns SPINEware, a facility that supports development and use of application and user oriented working environments on top of heterogeneous computer networks [Baal, 1998].First a concise description of SPINEware is presented. Next the major constraints that motivated the components based development of the latest version of the product are discussed. Also described is how components based techniques are applied to the development of SPINEware. The case description concludes with a short summary of experiences with the application of components based techniques.

2.1 Description of SPINEware

SPINEware is a collection of tools and software modules that supports the implementation of user and application area oriented metacomputers on top of heterogeneous computer networks. Such metacomputer provides the end user with a single and coherent application environment, along with a powerful user-oriented and tailorable graphical desktop system. The application environment (henceforth called working environment) gives access to potentially all resources from the computer network. The most important characteristic of a SPINEware-based working environment, which actually motivates use of the term "metacomputer", is that the user sees and operates a single computer. All details concerned with the heterogeneity of the network, remote access of resources, and information transfer among individual computers are hidden from the user. SPINEware allows the creation of such metacomputer and its tailoring to the specific wishes of the end users, working in specific application areas. The emphasis of SPINEware lies on detailed elaboration on the end-user viewpoint on the one hand, and on the provision of a single through-the-desktop environment of the heterogeneous computer network resources on the other hand.

The idea of providing the end user with a working environment that hides details emerging from computer and network usage, was first elaborated in the ISNaS project [Voge], aimed at the development of an information system for Computational Fluid Dynamics (CFD). The development of SPINEware started early 1992 at the National Aerospace Laboratory NLR as product called "SPINE". SPINEware is being further developed since 1996 jointly by NLR and the Japanese computer manufacturer NEC, who recognised SPINEware as a valuable tool for supporting their own supercomputer products in existing computer networks. The first commercial version of SPINEware, version 2.0, was announced by NEC early 1998, and is currently available for most popular UNIX workstations. New requirements, and experiences gained through version 2.0 has lead to development of version 3.0, which started by the end of 1997. This version is targeted for UNIX computers as well as Windows (95/98/NT) PCs.

2.2 Why components based development

Several constraints and experiences have motivated the application of components based techniques to the development of SPINEware version 3.0. In this section, the four most important motivations are discussed.

First, timely availability, giving a fixed budget is an important requirement. Development of version 3.0 concerned a redesign of the system, giving rise to reimplementation of large parts of the system. On the one hand it is recognised that a short time-to-market is of paramount importance for such system as SPINEware to become successful. On the other hand, for contractual reasons, the system must be realised within a fixed budget. It simply turned out that we could not afford development of the entire system from scratch. Hence, it was decided to potentially re-use existing, good, off-the-shelf software components, comprising commercial as well as non-commercial, and (de-facto) standard as well as "own" software.

A second motivation originates from a set of requirements concerned with the system being distributed. Presenting the resources available from a set of computers in a single and coherent environment, requires at least one software component to be installed on each of the individual computer, in order to manage and manipulate the resources on that computer on behalf of the environment. For this reason, a SPINEware-based working environment is realised as a distributed system. Extra requirements applying to this system are:

- Σ openness in the sense that new software components can easily be added;
- Σ flexibility in the sense that software components can be replaced by other; and
- Σ configurability in the sense that software components may easily migrate to other computers.

Implementing the system as a set of loosely-coupled components contributes to realisation of these requirements.

Another motivation for components based development of SPINEware version 3.0 is application of the good old "divide-and-conquer" principle to system development. The system, recognised as rather large and complex in its whole, has been subdivided into a set of smaller and simpler components, with well-defined interfaces among them. Development of each individual component can be carried out by a small number of people, and requires hardly interaction with the rest of the development team. Also, each component can be tested separately, making the verification of the quality of the entire system manageable.

Also, the realisation of SPINEware version 3.0 as an object-based system gives rise to components based development of the system. The resources, such as files, directories, tools, and printers, are presented to the user as objects. The user may access and manipulate an

object through a set of operations (methods) defined for the objects. The graphical user interface (i.e., the desktop) of a SPINEware-based working environment provides intuitive mouse-based operations, such as select icons in windows, drag-and-drop icons, and pull-down menus, to facilitate the invocation of methods. The object-based model gives rise to a decomposition of the software into components, comprising implementation of individual objects and modules for management of the objects.

2.3 Approach

As pointed out in the previous section, SPINEware version 3.0 has been designed, and currently is being implemented as an object-based, distributed system. It consists of the following components:

- Application objects modelling the native computers' resources, such as File and Directory (file-system objects), Tool (program or utility), WorkFlow and DataContainer (tool chains, work flows, and data sets involved with these), Printer (printer and related utilities), and ObjectFolder (set of application objects);
- Application Object Manager (AOM), responsible for administration (e.g., the interface in terms of methods and attributes) of the application objects, and the invocation of methods;
- Object Interface Layer (OIL), responsible for all communication – either or not via the network - among the components;
- User Shell (US), providing the graphical user interface, the desktop, of a working environment.

AOM, OIL, US, and each application object is implemented as a separate component. A component may be further subdivided into software items, mainly for being able to re-use existing software, to replace a piece of software, or just to divide-and-conquer.

For definition of the interfaces among the components, the CORBA (Common Object Request Broker) standard from the Object Management Group (OMG) is applied [OMG, 1995]. The OMG is an international consortium from industry, including system vendors, software developers, and users, that promotes the theory and practice of object-oriented technology in software development. The goal is to provide a common architectural framework for communication among application objects across heterogeneous computer systems. Applying the CORBA standard in development of SPINEware version 3.0 has two important advantages:

- CORBA provides a standard means for definition of interfaces among objects implemented as software components. This means that, if your system is so-called CORBA-compliant, integration of, or interoperability with other CORBA-compliant software is feasible with minimum effort;

- For realisation of communication among objects, use can be made of existing CORBA products. For SPINEware, the product ILU (Inter-Language Unification) from Xerox Corporation has been selected [Xero, 1997]. Such a product allows you to generate code from the object's interface definition and to implement the object as part of a implementation of the distributed system.

The CORBA compliance has made a first implementation of the system within short time feasible. Also, with the software components ported to all of the target UNIX and Windows platforms, implementation of the entire system with components running on UNIX as well as Windows was feasible within short time. In future, realisation of, for example, WorkFlow objects will be accomplished by "plugging in" an existing and CORBA-compliant work flow management system.
Reuse of existing software components is also made, or planned in other areas. For example, the US component is realised using an existing public-domain product for building graphical user interfaces, Tcl/Tk. To facilitate use of, for example, a Web-browser based, or Java-based interface in the future, the present US may be simply replaced by one based on a Web browser or Java.

2.4 Experiences

Applying components based technology to development of SPINEware version 3.0 has indeed helped us in realisation of the system, as described in sections 2.2 and 2.3. Components-based development principles such as "re-use" and "divide-and-conquer", have been, and yet are valuable contributions in the development of SPINEware version 3.0, especially with limited resources (budget, humans, time-to-market).

We consider the "divide-and-conquer" principle, originating from the early days of computer programming, mature; sufficient methods and expertise exist to apply this principle in system development. The "re-use" principle is yet immature. The ***use*** of re-usable software components is nowadays facilitated through indices, manual pages, libraries, and (Ada) packages. The ***production*** of re-usable software components, however, deserves attention. In practice, software development projects leave no, or occasionally only little room to put extra effort in making a system component that is reusable as well. In addition to interface definition, documentation, and verification, extra attention needs to be paid to generalisation and abstraction in order to make a software component reusable.

3 Case 2: Safety critical embedded software

The second case concerns safety critical embedded software. First a short description of the application is provided, followed by the safety requirements and the official safety classification definition. After a section on verification NLR's experience is provided, emphasising the issues related with component based development. The case description ends with some concluding remarks.

3.1 Application description

To fly aircraft under all (adverse) conditions, pilots must fully rely on the data presented to them, and on the correct and timely forwarding of their commands to the relevant aircraft subsystems. The embedded application discussed connects these subsystems with the aircraft flight deck by means of modern digital data buses. It combines, controls, processes and forwards the data between the subsystems and the flight deck. The embedded application is designed to operate in both Visual Meteorological Conditions (VMC) and Instrument Meteorological Conditions (IMC). Under the latter conditions the displays of the flight deck are needed by the pilot to fly, rendering the correct functioning of the displays safety critical. A number of equipment items needs to be duplicated to achieve the required failure probability.

During normal operation the embedded application processes about 100 different flight parameters, originating from 10 different sensors. Two processors are used in each of the duplicated hardware units. The delay times within the entire embedded application should be guaranteed to be less then 30 milliseconds with a cycle time of 20 milliseconds for the main processor. Due to the many changes expected during the operational life of the embedded software 50% spare processor time shall be allowed for. The I/O processor has a cycle time of 360 microseconds. Due to the available processing power these timing constraints alone prevent the use of standard component based development techniques.

The influence of safety on the embedded application's function will be illustrated for data input, the equivalent of the standard read statement. Depending on the criticality of the flight parameter, the software validates it in up to four complementary ways:

- coherency test: a check on correct length and parity of the data;
- reception test: a check on the timely arrival of the data;
- sensor discrepancy test: a comparison between the two data values produced by the two independent redundant sensors and;
- module discrepancy test: a comparison between the two parameter values produced by the same sensor; one value directly read by the system from the sensor, and one obtained from the redundant system via a cross-talk bus.

[Kess, Slui 1998] contains more information on the application.

3.2 Air transport safety requirements

For safety critical software in airborne equipment the [DO-178B] standard has been developed. The aim of this document is to provide guidance to both the software developers and the certification authorities. Usually acceptance of software is based on an agreement between the developer and the customer. In civil avionics an independent third party, the certification authority, performs the ultimate system acceptance by certifying the entire aircraft. Only then the constituent software is airworthy and can be considered ready for use in the aircraft concerned. [DO-178B] provides a world wide "level playing field" for the competing industries as well as a world wide protection of the air traveller, which are important due to the international character of the industry. The certification authority is a national governmental institution which in NLR's case delegated some of its technical activities to a specialised company.

As the entire aircraft is certified this implies that when an operator wants an aircraft with substantial modifications, the aircraft including its embedded software has to be re-certified. Substantial modifications are, for example, modifications which can not be accommodated by changing the certified configuration files. The influence of certification on the re-use of safety critical software components will be elucidated, based on NLR's experience with the embedded application.

3.3 Safety classification

Based on the impact of the system failure the software failure can contribute to, the software is classified into 5 levels. The failure probability in flight hours (i.e. actual operating hours) according to the Federal Airworthiness Requirements /Joint Aviation Requirements [FAR/JAR-25] has been added.

Level A: Catastrophic failure
Failure conditions which would prevent continued safe flight and landing.
[FAR/JAR-25] extremely improbable, catastrophic failure $< 1x10^{-9}$

Level B: Hazardous/Severe-Major failure
Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be:

- a large reduction in safety margins or functional capabilities;
- physical distress or higher workload such that the flight crew could not be relied on to perform their tasks accurately or completely;
- adverse effect on occupants including serious or potentially fatal injuries to a small number of those occupants.

[FAR/JAR-25] extremely remote, $1x10^{-9} <$ hazardous failure $< 1x10^{-7}$

Level C: Major failure
Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example,

- a significant reduction in safety margins or functional capabilities;
- a significant increase in crew workload or in conditions impairing crew efficiency; or
- discomfort to occupants, possibly including injuries.

[FAR/JAR-25] remote, $1x10^{-7} < \text{major failure} < 1x10^{-5}$

Level D: Minor failure
Failure conditions which would not significantly reduce aircraft safety and which would involve crew actions that are well within their capabilities. Minor failure conditions may include for example,

- a slight reduction in safety margins or functional capabilities;
- a slight increase in crew workload, such as, routine flight plan changes or some inconvenience to occupants.

[FAR/JAR-25] probable, minor failure $> 1x10^{-5}$

Level E: No Effect
Failure conditions which do not affect the operational capability of the aircraft or increase crew workload

.

3.4 Verification

In order to provide the developer with maximum flexibility, [DO-178B] allows the developer to choose the software life cycle. It enforces traceability to its general requirements. This life cycle will have to be approved by the certifying authorities. Each constituent software development process has to be traceable, verifiable and consistent. Transition criteria need to be defined by the developer to determine whether the next software development process may be started.

Verification is defined in [DO-178B] as "the evaluation of the results of a process to ensure correctness and consistency with respect to the inputs and standards to that process". Verification can be accomplished by review, analysis, test or any combination of these three activities.

Review provides a qualitative assessment of correctness.
Analysis is a detailed examination of a software component. It is a repeatable process that can be supported by tools. Every tool needs to be verified against the Tool Operational

Requirements, the contents of which is prescribed in [DO-178B]. For software tools the same documentation and configuration control procedures apply as for the airborne software. Every software tool needs approval of the certification authority.

Testing is "the process of exercising a system or system components to verify that it satisfies specified requirements and to detect errors". By definition the actual testing of deliverable software forms only part of the verification of the coding and integration processes. For software classified at [DO-178B] level A, a mandatory 100% code coverage applies. This code coverage consists of:

- statement coverage (every statement executed);
- decision coverage (every decision executed for pass and fail) and;
- the modified condition/ decision coverage (mc/dc). Mc/dc requires that for every condition in a decision, its effect on the outcome of the decision is demonstrated.

3.5 Experience

Modern aircraft contain huge amounts of software, supplied by numerous independent suppliers world wide. Even a single aircraft contains software of many different suppliers. According to the US National Transport Safety Board (NTSB), [DO-178B] works well as up to now no catastrophic failure (i.e. fatalities or hull losses) can be directly attributed to a software failure [IEEE, 1998]. An independent software engineering experiment using an [DO-178B] compliant software development process by NASA confirms that no errors were identified in the developed software [Hayh, 1998].

In the embedded application, software classified at levels A, B and E has been realised. For the launching customer of the product a first certifiable release has been produced. The time-to-market did not allow all requirements to be implemented. Subsequently considerable extensions had to be realised for the second group of customers.

The application's safety critical nature mandates that every requirement has to be worded such that independent verification and validation are possible. The users, amongst others pilots, have to understand the requirements, which prevents the use of formal specification methods. The baseline for the first version of the embedded application contained 207 requirements, of which 7 were not yet to be implemented. The second certifiable version is to be delivered 5 months later. The corresponding baseline contains 98 requirements changes (48%) which impact the software code plus an additional 55 requirements changes which (28%) do not impact the code. The latter changes are mostly clarifications and splitting of combined requirements. The lessons learned are to specify as precisely as possible, and to allocate, a separate identification for each verifiable item. This facilitates requirement management,

which conclusion [Hayh, 1998] reached independently. A formal definition method for the interfaces between the various applications running on different hardware, including its timing characteristics, would be helpful.

Between the first and the second certification 48% of the code changed, which implies a re-use of 52%. The 48% code changes breaks down in 28% modified lines, 14 % deleted lines and 6% added lines. The application consisted of four Computer Software Configuration Items (CSCI). Nearly all changes were applied to one CSCI. Between the first and the second certifiable release the total code size reduced slightly (8 %) despite the added functionality. This implies that more general solutions have been implemented.

The testing is partly automated using the Commercial Of The Shelf software (COTS) tool CANTATA. The actual test code is generated from Test Case Definition (TCD) files. The total non-comment size of these files is 1.2 times the non-comment source code size. This non-comment source code size includes non-executable statements like definitions etc. For each statement on average 4.5 tests are executed. Dividing the number of conditional statement (decision or loop) by the number of tests yields an average of 31 tests per condition. As most conditions contain a limited number of operands this illustrates that many tests are to verify assignments. The re-use of code also pays of in the re-use of tests. Note that [DO-178B] requires to repeat all tests for the second certification (full regression testing).

[DO-178B] requires traceability from requirements to code and vice versa. For the second certification full traceability using the entire updated requirement baseline has to be performed. This traceability requirement removes code implementing "obsolete" requirements in "unused code". Such obsolete requirements in supposedly unused code caused the Ariane 5 disaster [IEEE,1998]. [DO-178B]'s justified traceability requirement has consequences for the re-use of industrial standard components in a safety critical environment. Component re-use between companies is close to infeasible for the moment due to lack of universal specification techniques.

Currently, use of industry standard component based software architectures, such as CORBA's Object Request Brooker (ORB), Microsoft Com/ActiveX, or Java beans, for safety critical applications will be unlikely due to the same problems with precise specification, exhaustive testing and traceability of unambiguous requirements to code. As the commercial incentives for component based development apply even more for expensive safety critical embedded software, additional work in this field is needed.

3.6 Case conclusions

Standard use of component based development for safety critical embedded software is not yet possible due to technical limitations in producing unambiguous specifications, requirement traceability and resource usage. Some elements of component based software development like object orientation and re-use have been successfully used in a project at NLR. Other elements, like formal interface definition could be added.

There are huge commercial incentives to extend current component based development methods for safety critical embedded software development.

4 Conclusions

Component based development is an attempt to improve the software development process on three key characteristics, cost reduction, reduction of the time-to-market, and product quality improvement (i.e. the application is to perform its intended functions correctly).

In the aerospace environment the technical characteristics of the domain impose additional constraints of the software development process.

SPINEware, the first case, shows that applying components based technology has indeed helped in realisation of the system within the limited resources available (budget, humans, time-to-market). The divide-and-conquer principle is mature, sufficient methods and expertise exist to apply this principle in real life system development. The re-use principle is still immature. The ***use*** of re-usable software components is nowadays facilitated through indices, manual pages, libraries, and (Ada) packages. The ***production*** of re-usable software components, however, requires additional attention to generalisation and abstraction in order to make a software component reusable. In daily practice, software development projects do not have the extra effort to make their system components reusable.

For safety critical embedded software, it is not yet possible to use standard component based development techniques, due to technical limitations in producing unambiguous specifications, requirement traceability and resource usage. Some elements of component based software development like object orientation and re-use have been successfully used in a safety critical embedded software project at NLR. Other elements, like formal interface definition between several aircraft subsystems could be added.

Safety critical embedded software is expensive and time consuming to produce. Consequently there are huge commercial incentives to extend current component based development methods for this type of application

References

[Baal, 1998] *SPINEware: A practical and holistic approach to metacomputing*, proc. High-Performance Computing and Networking Europe 1998, Amsterdam, The Netherlands, April 1998.
E.H. Baalbergen

[DO-178B, 1992] *Software Considerations in Airborne Systems and Equipment Certification*

[IEEE, 1998] *IEEE, Developing software for safety critical systems*, J. Besnard, M. DeWalt, J. Voas, S. Keene

[Hayh, 1998] *Framework for small-scale experiments in software engineering*, K. J. Hayhurst

[JAR/FAR-25] *Federal Airworthiness Requirements/Joint Aviation Requirements*

[Kess, Slui, 1998] *Reliability, maintainability and safety applied to a real world avionics application,*
E. Kesseler, E. van de Sluis NLR TP-98037

[OMG, 1995] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, July 1995.

[Voge] *Development of ISNaS: an information system for flow simulation in design, in: Computer Applications in Production and Design*, F. Kimural (ed.), North Holland, ISBN 0 444 88089 5. M.E.S. Vogels, W. Loeve

[Xero, 1997] *ILU 2.0alpha12 Reference Manual*, Xerox Corporation, November 1997.