Reuse and DO-178B Certified Software: Beginning With Reuse Basics

Hoyt Lougee
Foliage Software Systems

To successfully approach reuse and the associated certification considerations, a rigorous understanding of reuse is important. This article, from a certifiability perspective, defines reuse, discusses reuse drivers and typical reuse scenarios, and details the various types of reuse. In addition, a brief overview of a reuse analysis and implementation approach will be presented.

Reuse has been defined variously: Definitions range from "the systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality, and business performance" [1] to "the process of creating software systems from predefined software components" [2].

The first definition is seemingly more complete, but the second definition is less restrictive and more useful. Too often in literature, a purist attitude is taken toward reuse. For example, often the term reuse is applied to only those elements that can be used *without change* or to only those elements that have been designed and configured for reuse. For this article, therefore, reuse is defined simply as *using previously existing software artifacts*. Artifacts include all products of a certification development process and include planning data, requirements data, design data, source code, configuration management records, quality assurance records, and verification data.

Reuse Factors

Functional Alignment

Two aspects of functional alignment can affect the reuse strategy adopted. The first aspect, applicability, is a determination of how well the existing requirements/functionality align with the requirements of the target application. Do the artifacts serve the intended purpose? How much must the artifacts be modified to accommodate any new functionality? The second aspect also concerns the alignment of new functionality to existing functionality, although in the opposite respect. Does the existing configuration contain *more* functionality than is needed for the targeted application? What must be done to accommodate this additional functionality?

The issues surrounding extra functionality are prevalent with design-for-reuse component libraries. These libraries are designed to include all possible future functionality needs and, by their very nature, include additional functionality. These existing configurations typically contain *more* functionality than is needed for the targeted application. How, then, must this additional functionality be accommodated in a certifiable software system?

A variety of strategies are available to handle extra functionality. Seemingly, the most simple is to strip the unnecessary functionality from the configuration (from requirements through verification artifacts). Conceptually, this is the simplest approach, but this may not be the most cost-effective approach.

"Understanding the rigor with which previous development was performed is critical in determining the amount of effort that will be required to incorporate existing artifacts into a new configuration."

Additional unused functionality may be retained in the code as long as the mechanism by which such code could be inadvertently executed is prevented, isolated, or eliminated [3] is verified. In other words, although the code is present, its non-availability within a specific application must be demonstrated. Typically, this means that the unused functional interface must be verified to ensure that the unused software is not used in a particular configuration.

These configuration mechanisms can entail a hardware switch such as jumper-pin settings, or can be performed purely in software. For example, if a software function is included in the object module but the entry point (the call to the particular routine) is not invoked, the software function can be shown as not accessed. Alternatively, if a routine includes a parameter switch to *turn off* parts of the routine's functionality, the switch mechanism can be verified and the software reviewed to ensure that the switch is always set appropriately.

Requirements Volatility

The ability to identify and isolate volatile requirements can maximize the ability to reuse. For example, if control logic were historically a primary source of change, an appropriate reuse strategy would dictate that the control logic is separated from non-volatile areas. This separation would enhance the ability to reuse non-volatile areas.

Both historical metrics as well as application-specific projections of change are important when considering requirements volatility. Applications may have inherent areas of instability that, by design, will always result in functional modifications; for example, application control laws that must be tuned for each targeted application. On the other hand, past areas of instability may have been resolved in the existing software baseline and application-specific changes indicating other *hot spots* are likely. In any event, a careful analysis of requirements volatility is vital in developing an appropriate reuse strategy.

Previous Development Rigor

Understanding the rigor with which previous development was performed is critical in determining the amount of effort that will be required to incorporate existing artifacts into a new configuration. When previous certification treatment is insufficient for the current application, whether the software to be reused is commercial off-the-shelf (COTS), software developed to other guidelines (for example, military guidelines), software certified to DO-178 or DO-178A, or software developed to DO-178B but to a lower software criticality level, effort must be expended to pro-

RTCA DO-178B: Software Considerations in Airborne Systems and Equipment Certification

Published by the Radio Technical Commission for Aeronautics, Inc. (RTCA) and adopted by the Federal Aviation Administration (FAA) Advisory Circular AC20-115B, DO-178B provides guidance in meeting airborne-product airworthiness requirements associated with software. Adherence to DO-178B adds an extra level of difficulty to the already challenging undertaking of embedded software development. The guidelines are not straightforward; interpretations vary, and acceptance is not always impartial.

DO-178B defines the objectives and activities that must be performed in developing and verifying airborne software systems. The specific objectives and the resulting rigor varies according to the criticality of the software, ranging from the most rigorous Level A for software whose failure can have catastrophic consequences to Level E for software whose failure has no effect on the aircraft's continued safe flight and landing.

Adherence to DO-178B, therefore, will produce evidence by which the applicant can instill confidence in the FAA that the software embedded in airborne equipment is safe for its intended use. The software development and verification processes necessary to generate this evidence can be costly and time consuming. As a consequence, avionics manufacturers, struggling with their cost and schedule constraints, often turn to reuse.

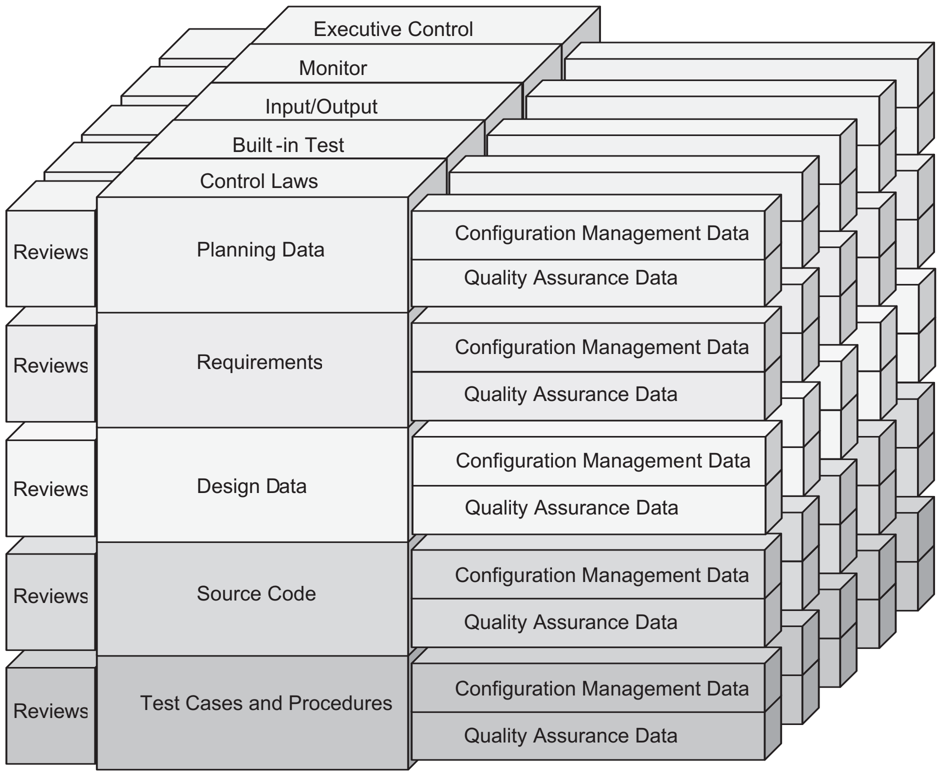
vide assurance that the software is suitable for the target certification effort.

If the previous software development was not certified with DO-178B, the existing development artifacts must be analyzed and mapped to the objectives of DO-178B. As a guideline, DO-178B does not dictate specifics with respect to data items or specific development processes. Instead, DO-178B details objectives that must be satisfied. Often, especially with military applications, a great deal of rigor was applied to the development process and a wealth of reusable artifacts is available.

On the other hand, if the previous development was certified with DO-178B, the previous development criticality level will determine the types of artifacts created, how the artifacts were configured and the type of change control provided, and the extent to which verification was performed.

Verification independence is also driven by the software criticality level. Higher criticality levels require greater levels of independence; therefore, the impacts of resolving independence issues must be considered. For example, criticality levels A and B require independence in assuring that the software high-level requirements comply with the system requirements and that the high-level requirements are accurate and consistent. These reviews must be re-addressed if the requirements are to be reused.

Figure 1: *Example Structured-Design-Based Configuration*



Maturity of Existing Artifacts

As a rule of thumb, reuse of *buggy* code is not a good idea – especially if the functionality is to be modified. Debugging modifications of *buggy* code compounds the complexity of the development process. As software issues arise during development, the source of the issues is not clear. Was the problem related to recent changes, was the problem related to reused elements, or was the problem related to a combination of the two? Moreover, the pedigree of buggy code may not be clear: Some bugs may necessitate major architectural changes – changes that, unfortunately, were not factored into the initial reuse analysis.

Defect history can be analyzed in several ways to *get a feel for the bugginess* of the previous software. The number and character of the defects found in the previous development effort can identify problem areas and provide insight into the amount and types of problems that can be expected. Analysis of the overall defect trending is also important. If the software was released and the defect-identification rate was still increasing, the software is sure to have undetected defects. On the other hand, if the defect-find rate was asymptotically approaching zero defects, the probability of a large number of undetected defects is lower.

Targeted Platform Changes

Often the ability to reuse software and reap the initial considerable investment in certifiable-software development is hampered by changing hardware platforms. Platforms change for many reasons ranging from strategic technology migration to obsolescence issues. Regardless of the motivation for changing the platform, the effects on software reuse are critical to the overall project impact.

Too often, the decision to update the hardware platform is performed without considering software reuse – disastrous effects on project schedules and budgets typically result. Since software development increasingly requires the lion's share of project budgets, software reuse should be a central consideration when developing a hardware migration strategy.

The target platforms must be analyzed in terms of concurrent multiple-platform support and the anticipated platform life span. The overlying product/business strategy must be examined to determine the need to support multiple-concurrent platforms. Questions to ask include the following:

- Is the software intended for use on varying concurrent platforms?
- What is the anticipated life span of targeted platforms?
- Is there a hardware migration plan (and if not, why not)?
- What are the characteristics of anticipated future platforms?
- Will future platforms be based on the same family of processors?
- Will the same basic hardware design/interface be retained?

Reuse Strategies

Full Vertical Reuse Versus Partial Vertical Reuse

Reuse can entail the entire life-cycle artifact set or subset. Full vertical reuse includes all life-cycle artifacts related to specific functionality. With certifiable aviation-software configurations, vertical reuse would entail all software life-cycle data: planning data, requirements data, design data, verification data, as well as configuration management data and quality assurance data. Partial vertical reuse would include a subset of this data: Perhaps only the requirements and design data would be appropriate for reuse. Clearly, full vertical reuse is preferred, but significant cost and schedule savings can still be accomplished by analyzing existing software systems for partial vertical reuse opportunities. Figure 1 illustrates a simple configuration based on a structured design.

As shown in Figure 2, full vertical reuse includes all life-cycle artifacts of the development, whereas in this example, partial vertical reuse only includes the requirements and design. Note that the associated quality assurance data and configuration management data, as well as the associated review data, are included with each vertical layer.

Even with full vertical reuse, however, there is still work to be performed to incorporate the reuse within a new application. Suppose that a feature whose functionality is unchanged is to be reused. Furthermore, suppose that all associated life-cycle data is expected to be accurate and appropriate with respect to the targeted application. A finite amount of work must still be performed and documented to ensure that the artifacts are indeed appropriate for use in the targeted application.

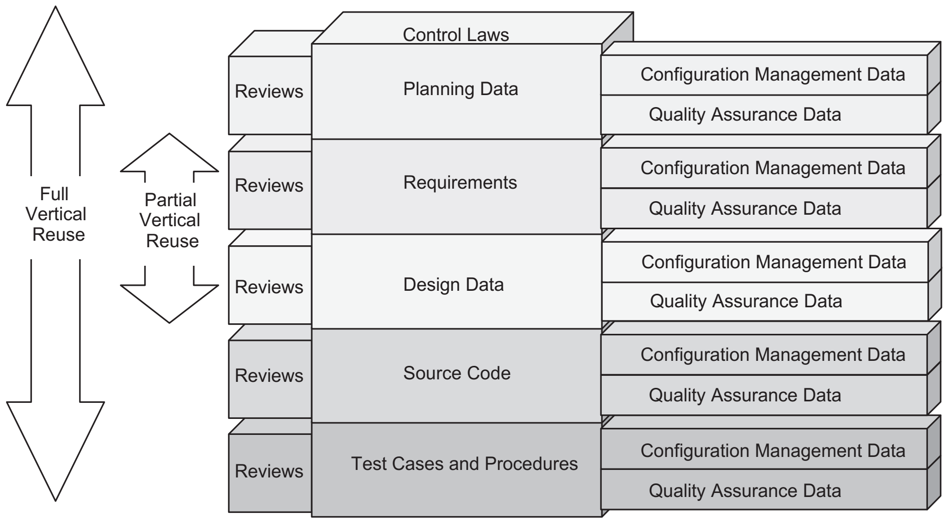


Figure 2: *Partial Versus Full Vertical Reuse*

With all this extra activity, what is gained? The reviews and analyses performed are typically neither as extensive nor time consuming as the initial *from scratch* reviews. These reviews and analyses are specific and focused on the integration of the reused artifacts into the target application. With respect to the design/architecture and code reviews, the focus is on the external interface to the reusable functionality. If the code to be reused includes 150 complex modules, only two of which interface externally, only the two modules would be the subjects of in-depth review.

Note that all previous and new review and analyses evidence are appropriate for the new certification effort.

Full Horizontal Versus Partial Horizontal Reuse

Reuse can include all artifacts within a specific life-cycle step or a subset. Full horizontal reuse includes all artifacts within a specific life-cycle step as illustrated in Figure 3. For example, reusing all source code would be an example of full horizontal reuse: All functionality is appropriate for the new application. Partial horizontal reuse would entail the extraction of a subset of functionality.

Typically, most design-for-reuse artifact libraries currently used today are horizontal code-component libraries. Often, an overall design is created based on the requirements at hand and an understanding of what is available in the code-component library. In fact, code libraries provided by language vendors adhere to this model: The user is to create an application-specific design based on the requirements and the language capabilities to support varying designs. With aviation software reuse, however, the common designs are critical in creating families of applications, especially with respect to alignment to hardware architectures.

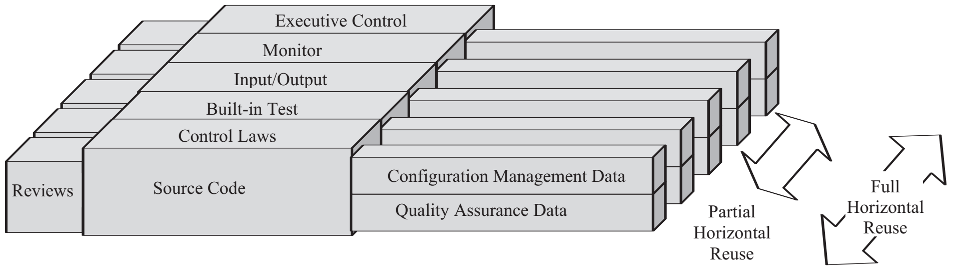
Note that partial horizontal reuse might not be undertaken with the goal of enhancing the systems features; partial horizontal reuse may be used to remove extraneous functions to create simplified applications.

Finally, as with vertical reuse, the verification of the reuse component interface to the target hardware and software is key. Careful analysis and planning for these interfaces must be performed.

Designed for Reuse Versus Not Designed For Reuse

Many organizations approach reuse from the purist *design-for-reuse* point of view.

Figure 3: *Partial Versus Full Horizontal Reuse*



These organizations create reusable artifacts with the express purpose of populating reuse repositories for use in future applications. These reusable artifacts typically require more time to create because they must be functionally robust to accommodate all expectations for future usage.

Once created, these repositories inevitably suffer from functionality creep: The *ultimate* functionality provided often does not account for some future usage scenario. As a result, the reusable elements are either duplicated and modified resulting in two similar elements to sustain or the element is extended with care for backwards compatibility. Of course, design for reuse can be very valuable and often provides the greatest cost savings. Clearly, well-conceptualized artifacts are easier to sustain and extend than constrained artifacts (designed as intended for initial use). But just as often, the costs associated with the creation of the repository are underestimated, as is the volatility of the functionality desired.

On the other hand, organizations often employ *scavenge* reuse, that is, harvesting existing artifacts that were not specifically designed for reuse. Depending upon the initial quality of the artifact, as well as the amount of horizontal and vertical reuse appropriate, scavenging existing software artifacts is often the best solution. On the other hand, if the initial software suffers from quality issues or the *fit* within the target application is not clean, starting from scratch may be the appropriate strategy.

Not Modified for Reuse Versus Modified for Reuse

Software artifacts that need not be modified for reuse typically offer the fewest certification hurdles. The cost-benefit is highest with scenarios in which the previous data can be used *as is* and only the applicability and interfaces verified. Minimal changes to artifacts, therefore, result in minimal additional certification effort. Often, only a regression analysis and a minimal regression suite are required to accommodate changed artifacts.

Changes to artifacts should be well considered to minimize the impact. Requirements and architectural changes in the software in which the reusable artifact is to be incorporated should often be tailored around reusable artifacts to minimize the overall project cost and schedule. Careful analysis of the requirements, design, and architecture of both the configuration to be reused, as well as the configuration into which reusable artifacts are to be incorporated, can provide critical input into the cost/benefit analysis and reuse strategy selection.

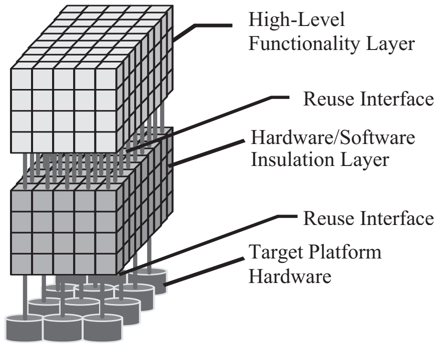
Partitioned Versus Non-Partitioned Reuse

Partitioned [4] software provides natural divisions for horizontal reuse. Designs that can partition software into volatile and non-volatile elements and minimize the amount of interfaces to be verified can result in significant cost and schedule savings. Since higher levels of criticality drive higher costs and longer schedules, minimizing the amount of software with critical functionality is desirable.

Partitioning a software system to separate higher- and lower-criticality levels can minimize the more costly critical severity verification activities. If the critical software partition is further designed with reuse in mind, the benefits can be twofold. For example, if engine-control software is partitioned into critical built-in test and engine-control functionality versus noncritical built-in test and monitoring communications, reuse could be performed on each partition independently.

If the noncritical built-in test and monitoring is most volatile, the more expensive engine-control and critical built-in test partition need not be completely re-addressed each time the more volatile areas change. This reused software can also build service history, lowering certification risk, and increasing confidence in the overall application.

Figure 4: *Layered Architectural Approach*


High-Level Functionality Layer
Reuse Interface
Hardware/Software Insulation Layer
Reuse Interface
Target Platform Hardware

Reuse Scenarios

Common Functionality – Different Target Platform

Avionics manufacturers often mitigate the effects of changing platforms with a layered architectural approach (see Figure 4). This architecture provides for a hardware-interface layer to insulate high-level application software from the effects of changing platforms: Software accommodation of hardware changes is limited to this interface layer. For different applications with different functionality using the same hardware, this insulation layer remains constant and can be reused. For applications with different hardware but with the same functionality, this interface layer cannot be reused, but the application software that *sits* on top of the insulating layer can be reused.

In so far as an application is certified for the functionality provided and that functionality depends on both the insulation layer and the application layer, the interface between the insulation layer and the application layer must be verified when either the high-level functionality layer or the insulation layer changes.

Common Functionality – Different Tools

Different target platforms, especially those not in the same family, often require changes in the toolset used in development and verification of the software system. Manufacturers often resist changing toolsets because of the additional impacts on tool qualification and new-tool learning curves. These hidden costs are often neglected in the planning of product changes.

Changes in design methodology typically have greater impact on qualification than do simple changes in compiler versions or changes in language. Isolation of change impact and the extent/scope of the regression analysis are typically more extensive with design methodology changes. Changing both the design methodology and the language compounds reuse issues.

With different languages, versions of the same language (and associated toolset changes), or even when different sets of compiler options are to be used that would result in different object code, previous DO-178B verification activities are typically invalidated and must be re-performed. When a different processor is used, the development toolset and the resulting object code will necessarily change, even for the same source code. In addition, hardware/software integration verification must be repeated. Hardware/software integration tests and hardware/software compatibility reviews must be updated as appropriate and performed again.

Common Functionality – Different Development Standards

When previous certification treatment is insufficient for the current application, whether the software to be reused is COTS, software developed to other guidelines (for example, military guidelines), software certified to DO-178 or DO-178A, or software developed to DO-178B but to a lower software criticality level, special certification considerations can be invoked.

In all cases, the objectives of DO-178B must be satisfied completely. The system safety assessment for the new application will provide guidance in the level to which this certification effort must proceed. Typically, reuse of significant portions of existing artifacts (code and supporting documentation) can be leveraged. Reverse engineering may be used to regenerate software life-cycle data that is inadequate or missing. As with all types of reuse, the Plan for Software Aspects of Certification should detail the strategy to ensure early buy-in by the certification authorities.

Common Functionality – Refactoring

For the purposes of this discussion, refactoring concerns the update of the software design and implementation without necessarily changing the functionality, tools, or target environment. Because of the significant costs of certification, refactoring is not typically performed; the rationale that "if it's not broken, don't fix it" provides the most cost/benefit. However, sometimes refactoring is necessary to provide a more robust, sustainable, and/or extensible application – and may be the appropriate, long-term, strategic approach.

Different Functionality – Common Platform/Tools

If functional alignment is not exact, change must be accommodated as discussed previously. This situation is probably the most common reuse scenario and includes both modifications of a single application for defect resolution and functional enhancements, and feature tailoring for different versions of the same application.

First and foremost in accommodating functional changes is the isolation of the change area within the software architecture. Localized changes facilitate regression verification, especially if specifically designed to do so. In addition, certification authorities, when reviewing modifications to certifiable configurations, must understand the change impact. Clearly defined localized changes are easier to analyze, easier to document, and easier to communicate to the certification authority.

Localizing changes is facilitated with highly cohesive and loosely coupled reusable artifacts. Loosely coupled elements have low levels of interdependency with their environments. Highly cohesive elements have a high level of uniformity in the element's functional goal. A cohesive element serves one functional purpose.

Aviation and Obsolescence

Electronic component manufacturers base their offerings on the ability of their product lines to generate income. Income is generated by selling quantities of components at specific prices. These manufacturers maximize their profits by providing high quantities at low cost rather than low quantities at higher cost. Unfortunately, the aviation market is not a significant consumer of electronic components. Instead, cell phones and personal computers drive the market offerings. As such, the fast, competitive pace of technology evolution has increasingly affected aviation manufacturers. Careful planning of component provisioning and migration are key to a successful product strategy.

Obsolescence issues make reuse key. In recent years, the obsolescence of electronic components has driven avionics manufacturers to redesign for new processors, memory and communications chips, and other electronic components.

Design for reuse often becomes design for obsolescence as manufacturers strive to reduce the cost of fielding their products. Manufacturing companies are often faced with a dilemma: Can a sufficient supply of components be purchased as a *last time* buy for all future projected use, can secondary suppliers that create the component in lower quantities at exorbitant prices supply the components, or should a redesign be performed with the associated recertification issues? Redesign often provides the ability to incorporate new functionality and defray future obsolescence issues, but the cost is often prohibitive. Reuse of all or parts of the software configuration can be key to finding the least-risk, go-forward manufacturing approach.

Different Functionality – Different Platform/Tools (Portability)

Changing functionality, as well as the target platform/toolsets, often occurs as new families of products are developed. A modular approach can be adopted to address the changes.

In contrast to the layered architectural approach, dividing architecture by specific functionality is also advantageous. As requirements change over time or as different members of the same application family require different functionality, specific functional divisions can be advantageous. The granularity of the architectural divisions must be carefully considered to isolate areas that can change independently (non-cohesive elements). In the extreme, hardware and software become reusable components – a *plug-and-play* strategy can be adopted. The best-known example of this reuse scenario is, of course, the personal computer with the wide array of associated peripheral devices. As with horizontal layering, verification of the interface is key to the incorporation of reusable elements.

Same Application – Different Aircraft Installation

DO-178B provides for "airborne systems or equipment, containing software that has been previously certified at a certain software level and under a specific certification basis" [3], being used in new aircraft installations.

If the system safety assessment performed for the new system does not indicate a new software level, the software configuration may be used *as is*. All software artifacts may be provided without change or further work for the new application certification. The development of a new Plan for Software Aspects of Certification and Accomplishment Summary may be the only tasks to be performed.

If any changes are to be performed to accommodate the new aircraft installation, the software must be treated as indicated above for previously developed software.

Benefits of Reuse

Technical and commercial trade journals widely tout reuse's promised benefits. Lower development costs, reduced programmatic risks, and shortened schedules flow from successful reuse and result in enhanced corporate competitive advantage. The value of reuse increases as the time, cost, and expertise invested in product development are continually leveraged across an ever-wider range of products.

A less publicized benefit of reuse is safety. For example, the concept of building a service history illustrates one of the safety benefits of reuse. Reused software, when properly analyzed and integrated, undergoes greater scrutiny and *time on the wing* over time. Another reuse safety benefit derives from properly partitioned software, which minimizes the amount of interaction between non-critical and critical software, thereby reducing the number of possible error sources for critical functionality.

A final benefit of reuse derives from

defect reporting over multiple applications. A well-coordinated reuse strategy will track defects common to reused components. When a defect is found on an application in a reuse component, other applications that use the same reuse component can be examined for defect impact and updated as appropriate.

Industry Case Study: Primus Epic

Honeywell's Primus Epic illustrates many of the reuse concepts discussed above. To address industry demands for system scalability, system reliability and maintainability, and reduced acquisition and application costs, Primus Epic, Honeywell's next generation integrated avionics systems, incorporates a highly flexible and cost-efficient framework.

The Primus Epic Product Line Architecture (PLA), packages integrated modular units and line-replaceable units into a single aircraft-wide Virtual Backplane Network. This architecture allows data generated by each system component to be available to all other system components.

Variation in the PLA is supported by the Module Avionics Unit (MAU): a cabinet containing field replaceable modules. These building blocks provide input/output, processing, and database storage functions.

The building blocks housed in the MAU communicate using the Avionics Standard Communications Bus using a Network Interface Controller module [5].

Partitioning was used with great effect to separate hardware components, separate hardware from software components, and separate software components within the Primus Epic system. Honeywell, for example, received a Federal Aviation Administration Technical Standard Order (TSO) approval for the modular avionics cabinet as an item of hardware. The various avionics functions such as the flight management system contained in the software are obtained with a separate TSO. The certification impacts of subsequent development or changes to a particular established configuration are considerably reduced.

Honeywell's Digital Engine Operating System (DEOS), which forms the Primus Epic software platform, provides standard services and interfaces for hosted applications. This operating system supports RTCA DO-178B partitioning, which minimizes the certification costs of the hosted software application by allowing different certification levels for applications with different criticality considerations. Different software functions hosted on DEOS can be certified with varying amounts of rigor, depending upon their particular effects on safety. Since software development and certification costs have increased astronomically in relation to the hardware costs, this capability further supports cost, schedule, and risk savings [6].

Honeywell's success with their Primus Epic system illustrates well the variability possible with a solid, common PLA. Primus Epic serves as the foundation for business jet, regional aircraft, and helicopter cockpits, including Dassault's Enhanced Avionics System (EASy) cockpit to be used in all new Falcon Jet models. The PLA accommodates extensive variation, including changes in aircraft configurations, changes in aircraft integrating components, radically different look and feel for both the displays (from two to six displays), and a variety of user input devices (from traditional controllers to new cursor control devices and voice-command mechanisms). Moreover, the specific functionality supported can range from movable navigation maps and real-time video to engine instrument and crew advisory systems and primary flight and navigation systems.

Primus Epic was designed as an integration platform; consider the EASy flight deck, which is based upon Primus Epic. Dassault was the primary system architect and worked in cooperation with Honeywell to create EASy. Honeywell opened up their previously proprietary communications bus specification to enable the creation or modification of a variety of custom and off-the-shelf components. Dassault was able to select among compliant avionics vendors to populate the cockpit, create new and effective configurations, and maintain their competitive advantage [7].

Conclusion

Cost and schedule can be saved, and safety can be enhanced with reuse for DO-178B certifiable software. A thorough understanding of the key reuse factors, a clear purpose and goals, a solid analysis, and careful planning are necessary to maximize the benefits of reuse. Manufacturers must analyze the many types of reuse and select among them. Reuse is a complex endeavor and the benefits are only available to those who approach it with care.◆

References

1. Ezran, M., M. Morisio, and C. Tully. Practical Software Reuse. 1st ed. London: Springer-Verlag, 15 Feb. 2002.
2. Dr. Carma McClure. "Model-Driven Software Reuse Practing Reuse Information Engineering Style," 1995 Extended Intelligence, Inc.
3. Radio Technical Commission for Aeronautics, Inc. RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification. Section 4.4.3 Structural Coverage Analysis Resolution, (d) Deactivated Code. Washington, D.C.: RTCA, 1 Dec. 1992 <www.rtca.org>.
4. Radio Technical Commission for Aeronautics, Inc. RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification. Section 2.3.1 Partitioning. Washington, D.C.: RTCA, 1 Dec. 1992 <www.rtca.org>.
5. Honeywell. "Primus Epic: Topology." Feb. 1998 <www.myflite.com/myflite/products/ias/epic/14T0p040gy.jsp>.
6. Hughes, David. "This Is Not Deja Vu." Aviation Week & Space Technology 4 June 2003.
7. Taverna, Michael A. "Making Flight Easy." Aviation Week & Space Technology 2 June 2003.

About the Author

Hoyt Lougee is the engineering manager, Aviation Division, at Foliage Software Systems. Foliage delivers software architecture, custom software development, and technology strategy consulting. Lougee's responsibilities include program management and software process improvement. Previously with AlliedSignal/Honeywell, Lougee has more than 13 years of experience with both military (DoD-STD-2167a) and commercial (RTCA DO-178B) aviation software development and certification efforts. Lougee has authored a number of white papers and presented at the 2002 Digital Avionics Systems Conference.

Foliage Software Systems
168 Middlesex TPKE
Burlington, MA 01803
Phone: (781) 993-5500
Fax: (781) 993-5501
E-mail: hlougee@foliage.com