JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 1, August 2004

Measuring Reuse in a Simulation Framework

Michael M. Madden[*]
NASA Langley Research Center, Hampton, Virginia 23681

In the early 1990s, NASA Langley Research Center (LaRC) embarked on a systematic reuse program for source code development in its simulation facilities. The program produced an object-oriented framework for real-time, human-in-the-loop vehicle simulation. The Langley Standard Real-Time Simulation in C++ (LaSRS++) was used as a case study to evaluate three static methods for identifying reusable code and to use their results to evaluate LaSRS++ as a reusable asset through metrics. Since static analysis cannot accurately measure reuse in large frameworks, two of the methods, the object chain and dependency chain methods, were designed to represent the lower and upper bounds of potentially reused components. With a large gap between lower and upper bounds, the refined object chain method was created to better estimate the likely set of reused assets. Reuse metrics were then extracted from simulation models of 11 aircraft. The three identification methods calculated average reuse rates of 74%, 84%, and 86%. The study also examined the ability to extend the adaptable architecture of LaSRS++ to the aircraft models using object-oriented techniques. Developers could then treat the aircraft models as reusable assets that are specialized using the same process for specializing LaSRS++. The models are specialized for individual experiments or to create a new simulation of a different aircraft type in the same aircraft family. These specializations are called variants. The reuse measurement techniques were applied to 10 variants. Reuse levels of aircraft models by variants were 60% - 99%, comparable to reuse levels of LaSRS++ by the aircraft models.

I. Introduction

INCREASING features and complexity drive the evolution of software. In addition to new features, customers also expect improving quality from software. A software organization quickly finds that it cannot increase its staff or costs to meet the growing demand for change. The software organization must pursue other methods to increase productivity, improve software quality, and reduce cost. Software reuse is one method that can achieve these goals.[1-3] Software reuse is most effective when it is pursued systematically, not ad hoc.[4,5] Systematic reuse programs emphasize the planning and development of reusable software assets for future use. Managing a systematic reuse program requires insight into its progress. Measurement provides that insight. Reuse measures constitute more than a measure of success on the program's focus, reuse. They are also necessary to analyze how reuse affects organizational performance and to weigh the impact of management decisions on the reuse program. Reuse measures provide the basis for the examining the effect reuse has on productivity, quality, cost, and time-to-market. Reuse measures allow managers to monitor the return on investment (ROI) of reusable assets and their continued relevance. Developing an asset for reuse costs 111% to 480% of the cost for developing a one-use asset.[3] The increased cost is justified by amortizing the cost over the number of products that reuse the asset. Reuse measures can provide the number of asset uses that are necessary to compute the ROI for a reusable asset. Dropping reuse levels can be a sign that the organizational knowledge of reusable assets is declining (which can occur with turnover) or that some reusable assets no longer have application in new products. Reuse measures also gauge how improvement programs affect reuse, especially if the focus is on increasing reuse. Retaining reuse measures becomes important for estimating cost on new projects. Software reuse is not free; the rule of thumb is that reusing

Presented as Paper 2001-4119 at the AIAA Modeling and Simulation Technologies Conference, Montreal, Canada, 6-9 August 2001; received 10 December 2003; revision received 26 March 2004, accepted for publication 30 June 2004. This material is a work of the U.S. Government and is not subject to copyright protection in the United States. Copies of this paper may be made for personal or internal use, on condition that the copier pay the $10.00 per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923; include the code 1542-9423/04 $10.00 in correspondence with the CCC.
[*]Aerospace Technologist, Systems Development Branch, Airborne Systems Competency; MS 125B; Michael.m.madden@nasa.gov. Senior Member AIAA.

software costs approximately 20% of writing equivalent new software.[6] A history of reuse measures allows project managers to estimate how much reuse they can expect on a project as part of estimating its cost.

Although the definition of software reuse has broadened to include artifacts other than code, measures based on code reuse remain the best method for accessing the benefits of reuse.[6] A survey of code reuse metrics is found in papers by Prem Devanbu et al. and by William Frakes and Carol Terry.[7,8] James Bieman also proposes a set of reuse metrics specifically for object-oriented code.[9] These metrics provide definitions of reuse and measurement equations, but they do not address how to identify the reusable assets in a finished product. Identification is straightforward in the simple case where self-contained functions from a library are reused without modification. A simple call-graph analysis would suffice. As the reusable assets become more complex, sophistication of the identification techniques must increase. Yet, the identification techniques must be capable of automation if the reuse measures are to be employed regularly in management decisions and project planning. Therefore, accurate identification of reusable assets may become impractical for some systems; and estimation methods must be employed.

Though many reports of reuse success exist, few reports detail their identification technique such that another organization can repeat the technique. Victor Basili provides detailed reuse identification in his 1996 paper on reuse in object-oriented systems.[1] Since Basili sought accuracy in his study, the identification technique relied on the developers to identify the origin of each component before they placed it the configuration management system. Though not automated, this technique could work in practice if it was infused into the development process and the organization found value in expending the manual effort to identify reuse. Though effective for his experiment, Basili's identification technique has a limitation in its applicability. It assumes all reuse can be traced through explicit selection of components by the developer. In frameworks, which are large-scale reusable assets composed of collaborating components, not all of the reused components may be identified by tracing through the components with which the new code interacts. Moreover, the framework could implicitly include components, which the completed product does not reuse. The developer may not be able to physically separate these unused portions of the framework from the final product.

This paper describes techniques that were developed to identify reused assets from the Langley Standard Real-Time Simulation in C++ (LaSRS++), an object-oriented framework for real-time, human-in-the-loop vehicle simulation.[10] NASA Langley Research Center (LaRC) developed LaSRS++ as part of a systematic reuse program to manage increasing complexity of its simulation experiments. The Flight Simulation and Software Branch (FSSB) currently maintains LaSRS++ and is responsible for developing the simulation products that run in LaRC's simulator facilities. This study examines 21 aircraft simulation products that were built using LaSRS++. These products fall into two categories: standalone aircraft models (11) or variants of the standalone aircraft (10). A variant represents either a different aircraft type in the same family of aircraft, or it represents a specialization for a specific research project. For example, the F18C is modeled as a variant of the F18A. The B757-WXAP product is a variant of the B757 for the Weather and Accident Prevention (WxAP) project.

The standalone aircraft models (a.k.a. base aircraft) are direct users of LaSRS++. Identification methods and reuse metrics were applied to the base aircraft to assess reuse of the LaSRS++ framework. Since inheritance and polymorphism are major mechanisms of reuse in LaSRS++, the base aircraft reflect the adaptable architecture of LaSRS++.[*] Developers take advantage of this architecture to utilize the base aircraft as reusable assets for the creation of variants. To judge the success of this architecture flow-down, the reuse measurement techniques were also applied to aircraft variants. Analysis of reuse in variants focuses on the base aircraft as the pre-existing code base and ignores LaSRS++ code. The reuse results of variants assess how object-oriented features can act as catalysts that transfer adaptable designs into derived products, enhancing the potential for reuse of the derived product.

II. Taxonomy of LaSRS++

Understanding how LaSRS++ approaches reuse aids formulation of the identification techniques and reuse metrics. It also provides a context for interpreting the results. Charles Krueger developed a taxonomy of software reuse that identified eight reuse approaches.[11] Under this taxonomy, LaSRS++ can be divided into two parts, each categorized by a different reuse approach. LaSRS++ provides both a software architecture and a source code component library.

[*]Inheritance groups classes, which share common attributes and behaviors, into hierarchies. Polymorphism allows a derived class to redefine the behavior of a base class interface.[28]

Product development using LaSRS++ emphasizes specialization of the software architecture, and the software architecture represents the majority of the LaSRS++ code. Krueger describes software architecture as:

> large-grain software frameworks and subsystems that capture the global structure of a software system design. This large-scale global structure represents a significant design and implementation effort that is reused as a whole.[11]

A software architecture is specific to an application domain. Its abstractions constitute a domain language for the architecture. LaSRS++ covers the domain of vehicle simulation. Developers create simulation products from LaSRS++ using component refinements. LaSRS++ relies on object-oriented techniques of inheritance and polymorphism as the primary refinement mechanisms. Krueger defines specialization via component refinement as vertical. Thus, LaSRS++ is a vertical software architecture.

Krueger's definition of software architecture is broad. It focuses on the domain-based abstractions as the primary characteristic. The software architecture must also have a realization mechanism that produces an executable product, but this realization can take many forms. Sparks et al. use the term "object-oriented framework" to refer to a software architecture whose realization is "a set of classes designed to work together to solve a problem or supply some capability".[12] (This paper will use the abbreviated term, "framework".) Sparks also defines two styles of frameworks, "called" and "calling". Products reuse called frameworks by calling the framework when the framework's service is needed. On the other hand, calling frameworks control program execution and call the new code. LaSRS++ is a calling framework. LaSRS++ provides a main() program and an executive package for program control. Developers reuse the LaSRS++ framework by specializing LaSRS++ classes with product-specific derivations and by registering key derived classes to the framework.[13] Sparks states that calling frameworks achieve higher rates of reuse because "the bulk of the framework is reused from the single decision to reuse that framework".[12] However, a given product may not reuse the whole framework. In fact, the LaSRS++ framework does contain "optional" components that the developer must explicitly select and instantiate to be reused; such instantiation can occur through inheritance or composition. For example, LaSRS++ provides weapons modeling as an option. Military aircraft models will reuse the weapons classes, but civilian aircraft models will not.

LaSRS++ also provides a source code component library as defined by Krueger. The library contains classes and support functions that the developer can reuse when developing new code. The LaSRS++ component library contains 560 components. Explicit reuse of these components requires developer selection and integration. Some components are also designed for specialization via inheritance or template parameters. The library components were used to build the framework portion of LaSRS++; therefore, many library components are implicitly reused even if the developer does not explicitly select them.

Krueger defines two characteristics that a source code component library must possess to make use of the library more effective than building components from scratch. The library must provide "a level of abstraction that emphasizes what the components do as opposed to the source code that describe how it is done" and "techniques to locate components efficiently".[11] Basically, the developer must be able to find the component and understand how it works faster then they could build it. LaSRS++ provides the necessary abstraction level via its application of object-oriented design. LaSRS++ restricts implementation details to the private portion of the class. The public portion contains only the methods and data definitions required to use the component. Thus, developers can quickly learn what the component does by reading only its public interface. This learning is facilitated by a common, human-readable style for header files and detailed comments for each public method and data type. The common code style and comment requirements are defined in the LaSRS++ Programmer's Guide, and they are enforced using code inspections.[14] LaRC also uses the DOC++ document generator to format component information for web browsers.[*]

LaRC uses simple techniques to aid developers in locating components. The components are carefully named using English words and common abbreviations to indicate the services they provide. The components are stored in directory structures where directory names indicate the theme of the components that the directory contains. For example, the Quaternion component is stored in a directory called Math. The directory structure is indexed using the GNU Find Utilities.[†] Thus, a developer who knows the component name or who can guess part of the name can find the component using the locate command. Components can also be located using the DOC++-generated documentation; but the structure of this web document mirrors the directory structure of the source files.

These techniques would not normally be efficient for selecting reusable components in a library this large.[15] However, developers quickly become proficient in the use of the LaSRS++ library through interaction with the

[*]Data available online at http://docpp.sourceforge.net (cited July 2004).

[†]Data available online at http://www.gnu.org/software/findutils/findutils.html (cited July 2004).

framework portion of LaSRS++ and through peer-reviews of software designs. Because the LaSRS++ framework is built using the component library, many component elements appear in the public and protected methods of the framework. The developers must understand these components to specialize the framework for their product. FSSB's development process requires peer reviews of software designs. These design reviews provide an opportunity for experienced LaSRS++ developers to point out selections from the component library that would improve reuse in the design.

The Software Engineering Institute (SEI) has identified a reuse practice that can encompass both the framework and library identities of LaSRS++. The Software Product Line Practice also describes characteristics of LaSRS++ that the reuse approaches of Krueger do not address.

> A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.[*]

Two features of a software product line distinguish it from the general reuse approaches, the scope of its reusability and a prescribed process for creating the product from the reusable assets. The ideal software architecture is general enough to be reused in any product within the architecture's application domain. However, the ideal software product line strives only to be reusable within a family of products. Typically, software product lines are developed by an organization for its own use, though collaborations and acquisition of product lines are also possible. LaSRS++ lies between both ideals. LaSRS++ was designed with a product line mentality. However, because LaSRS++ was developed for use in a research environment, a broad scope of possible products can be envisioned. Some of these possibilities extend beyond FSSB's current mission and require designs with more general applicability. Perceived effort became the balancing criteria when deciding between a design that satisfies FSSB's current mission and a design that would allow FSSB to take advantage of opportunities to expand its mission. The decision to abstract the world model is one example. FSSB's mission is to provide flight simulation products to aeronautics researchers. This mission only requires modeling of the Earth. There was no need to abstract the world model; the only envisioned use would be the modeling of extraterrestrial bodies. Modeling extraterrestrial bodies was considered possible because FSSB is a NASA organization but improbable because other organizations were responsible for space simulations. The additional work was perceived to be small so the design was pursued. The decision later paid off when LaRC decided to propose the ARES Mars aircraft for NASA's Mars Scout Opportunity (a 2007 Mars mission). LaSRS++ was used to perform feasibility and performance studies of design concepts for ARES. Because of the early design decision to abstract world models, inserting a model of Mars into LaSRS++ could be done quickly.[16]

> SEI provides a more detailed definition of the prescribed product creation found in software product lines: Each product is formed by taking applicable components from the base of common assets, tailoring them as necessary through preplanned variation mechanisms such as parameterization or inheritance, adding any new components that may be necessary, and assembling the collection according to the rules of a common, product-line-wide architecture.[*]

At a high level, this definition describes how simulation products are built using LaSRS++. Every aircraft model in LaSRS++ must follow the same basic architecture. The aircraft model must descend from the Aircraft class. The dynamics of the aircraft model must be broken into subsystems. These subsystem models must descend from SimulationModel. How the aircraft model interacts with its subsystem models is also prescribed. A detailed treatment of aircraft model construction in LaSRS++ that exemplifies prescribed product creation is found in Ref. 13.

As a "software product line", the scope of LaSRS++ encompasses the functions common to all LaRC simulations. The LaSRS++ architecture divides this scope into eight packages.

1) Executive - This package contains the high-level classes responsible for system construction, operation, and destruction.

2) Real-Time Services - This package manages the real-time clock and synchronous data transfers with simulator hardware.

[*]Software Engineering Institute, "A Framework for Software Product Line Practice," available online at http://www.sei.cmu.edu/plp/framework.html, Version 4.1, Feb. 2004 (cited July 2004).

3) Models - This package contains classes that model the physical world or provide support functions for the models. Examples include world models (shape and dynamics, atmosphere, gravity, and navigation database), equations of motion for dynamic objects, abstractions of aircraft subsystems (e.g. engine, aerodynamics, landing gear, weapons), data recording, and linear model generation.

4) Hardware Communication - These classes handle communication with external simulator hardware. Hardware communication classes manage the mechanism for transporting data between the simulation and the simulator hardware (e.g. Ethernet, SCRAMNet+). They also perform necessary data translations (e.g. 32-bit ARINC-429 words for avionics).

5) Hardware Mediators - The LaSRS++ architecture separates the model package from the details of the simulation hardware and vice-versa by using a mediator pattern.[17,18] Hardware Mediators actively control the transfer of data between the model and hardware communication packages. On input, the mediators move data from the hardware communication objects into the model objects; and, on output, the mediators move the data from the model objects to the hardware communication objects.

6) Toolbox - The Toolbox package contains primitive classes that are used as building blocks for other LaSRS++ packages. These primitives include mathematical constructs (e.g. quaternion), abstraction of OS services (e.g. threads), screen and file I/O, common data protocols, and abstractions of transport mechanisms. The majority of these components are not specific to the simulation domain.

7) Graphical User Interface Toolbox - This package contains primitive objects designed to simplify the building of graphical user interfaces for LaSRS++ engineers. These primitives are used to build the LaSRS++ Graphical User Interface (GUI) and are used by projects to build extensions to the LaSRS++ GUI. These primitives are higher-level abstractions built on top of a third party GUI library. Currently, LaSRS++ uses the GTK-- library. Like the non-GUI toolbox package, the components in this package are not specific to the simulation domain.

8) Graphical User Interface - The GUI package defines the architecture and object collaborations of the LaSRS++ GUI. Within that architecture, the package defines mechanisms for projects to extend and specialize the GUI. The components in this package are specialized graphical elements for viewing and manipulating the simulation, which is composed from the first six packages.

The two "toolbox" packages (6 and 7) exemplify the parts of LaSRS++ that developers interact with as reuse libraries. The other six packages form the calling framework. Table 1 shows the size decomposition of LaSRS++ among the eight packages as a percentage of the total size of LaSRS++. (The percentages add to 101% due to rounding errors.)

Table 1 Package Composition of LaSRS++

Package	% SLOC
Executive	1%
Real-Time Services	4%
Models	29%
Hardware Communication	10%
Hardware Mediators	10%
Toolbox	25%
Graphical User Interface Toolbox	8%
Graphical User Interface	14%

Although LaSRS++ is best described as a software product line, this paper will refer to LaSRS++ as a framework. The term framework is more widely recognized and prior literature on LaSRS++ uses this term exclusively.

III. Categorizing Code Reuse

Before identifying reused code, its characteristics must be defined. Computer Sciences Corporation (CSC) defined four categories of reuse in their "standards and practices" manual:[19]

1) Transported. The code is reused as-is from a pre-existing code base.

2) Adapted. The code is taken from a pre-existing code base and less than 25% of it is modified.

3) Converted. The code is taken from a pre-existing code base and 25%-50% of it is modified.

4) New. The code is written from scratch; or the code is taken from a pre-existing code base and more than 50% of it is modified.

Frakes and Terry combine the adapted and converted categories together into a single "adaptive" category.[8] Bieman also uses a single category for adapted code but calls it "leveraged."[9] Both Bieman and Frakes and Terry use the term "verbatim" in place of "transported."[8,9] This paper will use the simpler three-category system and employ the terms: "verbatim", "leveraged", and "new". Some studies also give credit for reuse if a program uses new code more than once. Frakes and Terry call this "internal" reuse.[8] (Reuse from a pre-existing code base is "external" reuse.) Since this study attempts to measure the effectiveness of reusable assets (i.e. LaSRS++ and base aircraft), internal reuse is ignored.

Under the LaSRS++ development process, developers are not permitted to copy and tailor LaSRS++ code for a product, i.e. create a product-specific version of the class. If a product requires data and/or behavior modifications to a LaSRS++ class, the developer must use inheritance and polymorphism to make the changes. The product-specific specialization must inherit from the LaSRS++ class. Since the process does not recognize tailored LaSRS++ code as reuse, the identification methods were designed to ignore it. Thus, all identified LaSRS++ components would appear to be transported reuse (a.k.a. verbatim) under CSC's definitions. However, Bieman views inheritance and polymorphism as language-supported mechanisms for modifying reused code.[9] Thus, Bieman categorizes inheritance as "leveraged" reuse. This paper will apply Bieman's definition from the perspective of the new code only. When a new class inherits from a LaSRS++ class, then the reuse is categorized as "leveraged". All ancestors of the "leveraged" LaSRS++ parent are also counted as "leveraged" reuse. All other reuse is verbatim. For example, if new code exercises verbatim reuse of a LaSRS++ class, then all ancestors of that LaSRS++ class are also counted as verbatim reuse.

The same rules also apply to variants of base aircraft. The development process discourages copy and tailor of the base aircraft code. The developer must use inheritance and polymorphism to add the variant's unique data and behaviors to base aircraft classes. Inheriting from the base aircraft constitutes leveraged reuse of the base aircraft. All other reuse of the base aircraft is verbatim.

IV. Methods

A. Identifying the Reused Components

The first step in measuring reuse of LaSRS++ projects is to identify the reused components. Systematic reuse programs face a fundamental limitation in precisely identifying what code was reused. Components designed for reuse tend to provide a broad feature set to accommodate the multitude of situations in which the component may be used. But, only a subset of the features may actually be exercised when the component is reused in a single product. The developer cannot separate the required features from those that are not used and cannot count them separately.[20] This problem is magnified in frameworks where collaborations of classes can represent unused features. The design of the framework may not permit the developer to physically separate all unused collaborations from the final product.[20]

Taking a line of code (LOC) view of reuse aims to answer the question, "How much code did the reusable asset save the developer from writing?" The amount of code branching that is typical in large systems exponentially explodes the possible code paths that a product can take. Identifying the subset of paths that the product will take requires costly analysis and may be technically infeasible. Regular measurements for the purpose of assessing and managing the asset will not be possible. LOC-level analysis is also unnecessary. LOC-level identification really acts as an intermediate step to answer the more important underlying question, "How much did the reusable asset save in reduced effort and cost?" LOC-level identification is an attempt to make the reuse percentage match the percentage of saved cost. The underlying assumption is that reused code has the same cost characteristics as new code. As discussed in the introduction, this is not true. Reused code has a different development cost than new LOC, and reuse is not free. To determine the actual savings of reusable code, its cost is amortized across all products that use it. The reused code and its cost must be tracked separately. Since lines of code cannot be physically separated from their component, their cost is bound to the cost of the component. Thus, component-level identification is as good as LOC-level identification for quantifying cost savings when reused code costs are tracked separately.

Component-level identification is also more appropriate for assessing the utilization of the reusable asset by the development team. Developers generally don't retrieve and interface with individual lines of code from the asset. Instead, the developers work with higher-level abstractions. This paper defines a component to be the smallest unit of abstraction that can be reused as a whole. For object-oriented software, the component is a class. For non-object-oriented software, data structures, functions, and/or modules are components. Identifying reused components provides the finest grain input into reuse measures that quantify utilization of the reusable asset. Component-level identification provides a reasonable basis for assessing cost savings and utilization. Thus, this study uses component-level identification as an input into its reuse metrics.

Culling the unused components from larger reusable assets remains necessary to improve the accuracy of the reuse measurements. Static analysis of the product code is a fast and economical means of identifying components. But static analysis provides, at best, an estimate of the reused components. Static analysis cannot predict which code branches that a product will take within the reusable asset. Because code outside of branches will be executed for all uses of the asset, components that reside outside of code branches in the reusable asset will be reused when the asset is reused. Thus, components that have the potential to remain unused must reside in code branches; code branches represent the only optional paths through the code. Static analysis frequently does not have enough information to predict which code branches a given product will take within the reusable assets. To tackle the issue of code branches, two static analysis methods were developed to provide a lower and upper bound of reused components. A method provides an upper bound if it assumes all code branches are taken. It will select all components reused by the product and may possibly falsely identify extra components. A method provides a lower bound if it assumes that the reusable asset takes none of its code branches. It will not falsely select unused components but, in the process, may overlook reused components.

The dependency chain (DC) and the object chain (OC) methods were developed by the author to provide these bounds. The DC method identifies all of the source files that are required to build the product. This is equivalent to traversing all code branches; thus, it provides the upper bound. The OC method statically analyzes the code for the creation of objects. It identifies the classes and ancestor classes of the objects that it selects. This method provides a lower bound when it ignores objects created within code branches that appear within the reusable asset, i.e. LaSRS++.[*] However, it does select all objects created within new code whether or not the object is created within a code branch. The method assumes that the developer would not write code branches in new code that the product is not intended to take. The OC method views classes as required if they are instantiated in the new code and outside of code branches in the reused code. But, it views classes as optional features when the class is instantiated only within code branches of the reused asset. (For non-object oriented systems, a similar method would ignore function calls and data structure accesses that occurred within conditional statements that appear within the reusable asset.)

In calling frameworks that are object-oriented, abstraction may present a barrier for reuse identification from a purely bottom-up search, i.e. starting the search only from the new code. For example, the Models package in LaSRS++ is designed free of dependencies from the other framework packages. Other framework packages operate on the Models package, but the Models package is a passive server unaware of its clients. The Executive package runs the Models package. Likewise, the Hardware Mediators package transfers data between the Models package and the Hardware Drivers package. The GUI package transfers data between the Models package and the user. All of this occurs without the model package initiating contact with the other packages. Thus, the model package has no source code dependencies on the other packages. If new code directly reused only the Models package, a search starting with the new code would be confined to the Models package. Reused components outside of the Models package would not be visible in this search. Thus, the methods used in this study start their search using both the new code and the Executive package. As the program control, the Executive package depends on all other framework packages; it is a starting point for the top-down search.

The dependency chain (DC) was obtained from ClearCase®, the configuration management tool used for LaSRS++. When used for build management, ClearCase records the source files that the compiler reads when it creates each object file. [Compilers that generate makefile dependencies from each source file can provide a similar list.] The search begins with the records for the object code that was built from new code and the LaSRS++ Executive package. Header files for classes not previously examined are extracted from the lists. Then the object file records for those classes are examined. This process continues until the search produces no unexamined classes. For LaSRS++, the DC method can falsely add some classes because some LaSRS++ components are conditionally created based on checks of the vehicle's contents. For example, the file dependency list for the B757 includes all LaSRS++ components related to weapons because the LaSRS++ graphical user interface (GUI) conditionally creates a weapon system dialog if the vehicle has a weapon system. Conditional instantiation based on vehicle contents is the main cause of unused collaborations binding to a LaSRS++ product. The DC method identifies all reused files and falsely identifies some files as reused. Thus, reuse measures derived from the DC identification represent an upper bound for LaSRS++.

In the OC method, the source files are statically analyzed to produce a list of created objects. A Perl script was created to extract the object list from the source files. First, the vehicle-only files and select files from the Executive package are examined for the list of LaSRS++ objects that they create. Then, LaSRS++ files related to the object list

[*]In C++, "if-else" and "switch" statements are examples of code branches.

are analyzed for a list of objects that they will unconditionally create and that have not been previously examined. The last step is repeated until the object chain is exhausted. This describes, in principle, what the OC method does. To keep the script small and its execution time manageable, the implementation used in this study has some further restrictions and limitations

The implementation of the OC method identifies an object (i.e. a class instance) if any of the following conditions are encountered:

1) An object of the class is declared (but not a pointer or reference).

2) An object of the class is allocated using the operator new.

3) The class is a superclass of a previously identified class.

4) A method of a previously identified class returns an object of the class by value.*

5) A scope-qualified invocation of a class member is made and no other instance of the class is identified. This condition identifies class utilities, which contain only static members. The program creates a single global instance of the class utility. Thus, all invocations for a class utility are attributed to a single global object.

The script will fail to identify classes instanced only under the following conditions:

1) The class is instanced only as a type argument of a template instantiation.

2) A method of a previously identified class uses pass-by-value for an argument of the class type.

3) LaSRS++ code creates the object within an if-else or switch code block. These statements represent the code branches that the OC method ignores by design. Classes instanced anywhere within new code are identified, however.

The implementation finds most, but not all the LaSRS++ classes that the simulation will instance unconditionally. The first two situations represent simplifications that ignore object creation in circumstances that are rare in LaSRS++; these circumstances did not justify the added the effort to make a reliable search for them.

The implementation also makes two simplifying assumptions that could weaken the assertion that it provides a lower bound. The implementation assumes that the product uses all methods of an identified class and that developers do not use for and while statements to represent optional code paths. Both assumptions were made because strict adherence to the OC method definition would have resulted in search criteria that discard important object creation code. This becomes important later because the OC implementation is the only implementation capable of computing the frequency of component use that is input into some of the selected metrics.

Identification of a LaSRS++ class does not guarantee that all of its methods will be called. Under a strict OC method implementation, object creations should be detected only in those methods invoked outside of the LaSRS++ code branches. However, static analysis cannot resolve virtual method calls (i.e. polymorphic calls). Thus, the virtual method call must be viewed as a code branch where the code can branch to any one of the concrete implementations that reside in the product. To strictly maintain a lower bound, virtual methods must be eliminated from consideration. The problem with implementing this requirement is that many important LaSRS++ behaviors are contained within virtual methods; this includes portions of system construction at startup. Eliminating virtual methods would discard important object creation code. In the spirit of performing component-level identification, the implementation avoids this problem by assuming that the product uses the whole component minus the component's code branches.

The implementation assumes for and while statements are looping constructs that handle repetitive actions; developers will use if-else and switch statements to program optional behavior. Though a developer could program optional behavior conditions as a for or while loop, including for and while statements as code branches would have further eliminated many legitimate components that are instanced only in one-to-many relationships (i.e. these classes participate only as collections of objects). The author asserts that the number of unused components that this simplification identifies, if not zero, are two small in number to significantly move reuse measures based on the OC identification from the lower bound.

To assess the impact of the above two simplifications, a version of the OC script was modified to ignore for and while loops and to assume that only the class constructor was called. Without performing further analysis, the constructor is the only statically bound method of a class that the product is guaranteed to call. (This could still be an issue if a class has more than one constructor but the average number of constructors per LaSRS++ class is ~1.1.) This allows for a quick assessment of the maximum impact that the simplifications could have on results; but examining only constructors produces an overly restrictive script that strictly adheres to the OC method. This script identified 19% fewer LaSRS++ files on average. The effect on the metrics is discussed in the "Results" section. The differences, though significant, were not so large that they rule out the simplified OC implementation as a reasonable lower bound.

*This was a welcome side effect of the search pattern for object declarations.

The OC method eliminates implicit selection of components within the reused asset's code branches. It is the conditional creation of objects in LaSRS++ files that creates false identifications in the DC method. The OC method ignores object creation within if-else and switch statements that appear in LaSRS++ code so that it avoids falsely identifying classes as reused. But, this also may cause the method to miss valid, reused classes. The OC method will not count classes that are instanced under those conditions that will never be true for the given vehicle. However, it may overlook classes that the project does use such as those instanced under conditions that can become true for the vehicle or in which one of a set of conditions must be true for every vehicle. Those implicitly reused classes that the method does identify will be instantiated in the unconditional code that every product will execute. In essence, the OC method captures the skeleton of LaSRS++ whose use is required plus all LaSRS++ collaborations that the new code explicitly reuses. Within the limits of the simplifications described earlier, reuse measures derived from OC identification should represent a realistic lower bound for reuse measures.

The true collection of reused components should lie between sets identified by the OC and DC methods. The reused file counts in Table 2 show that there is a large gap between the two techniques when they are applied to aircraft models. In all cases, the OC method selects less than half of the LaSRS++ files selected by the DC method. To obtain a better idea of where the true collection may fall in this range, a third result is computed by refining the results for the OC method.

The OC list of reused files is refined by inspecting the difference between the reused class lists from the OC and DC methods. The difference lists for all 11 aircraft share a common set of 861 files. This set of 861 files is called the "commonly rejected" files. Thirty-six percent (309 files) are from the toolbox packages and were divided as follows: building blocks for math models (32 files), external communication (66 files), or GUI elements (209 files). The remaining domain specific files are broken down as follows: math models (146), system interfaces (122), and GUI elements (286). In total, 57% of the rejected files are GUI components. This is not surprising since many GUI elements are conditionally created in response to user inputs. The remainder is nearly split evenly between math models (21%) and external interfaces (22%). An inspection of the math models and interface files reveals a host of components that are conditionally built based on runtime options and cockpit selection. A visual inspection reveals that less than one-fifth of the commonly rejected files represent parts of potentially unused collaborations that are conditionally created based on vehicle contents; the other 80% are generic components or files that all aircraft should reuse.

Adding the commonly rejected files to the object chain's list should result in a list that provides a better identification of reused LaSRS++ components. This refinement method is called the refined object chain (RO) method. The refined number represents neither a new upper bound nor a new lower bound. A combination of the object chain list and the common rejected file list may contain files that a project does not use in addition to possibly overlooking files that the project does use. The effect of the RO method is to weed out optional components requiring explicit instantiation. These components find their way back onto the DC list through other components that utilize their interface and are conditionally built by LaSRS++ based on detection of their instantiations. (In this sense, refined dependency chain would have been a better name.) Returning to the weapon system example from the DC discussion, the RO method will not identify the WeaponSystem class as reused because it is explicitly instantiated by the military aircraft and appears only on their OC lists. (And, thus will not appear as one of the commonly rejected files.) But, the RO method will still identify the WeaponSystemDialog because it is conditionally created and, therefore, will always be rejected by the OC method and appear on the commonly rejected files list unless an aircraft specializes WeaponSystemDialog. The specialization (new code) will cause the WeaponsSystemDialog to appear on the OC list for that aircraft; therefore, it will no longer appear on the commonly rejected files list. This is also how legitimate files also fail to appear on the RO list. If an aircraft specializes the AeroDialog, which is conditionally created but reused by all aircraft, it will no longer appear on the RO list. Through the RO method can be improved by manually removing the potentially unused components from the list of commonly rejected files, LaRC is interested in automated methods that do not require human calibration. The RO method defined here can be fully automated. However, changes in its relative distance between the DC and OC methods must be monitored for signs that the characteristics of the commonly reject files have changed. The RO method is used as an indicator of whether the true value of the reuse metrics lies closer to those computed from the DC method results or those computed by the OC method.

B. Selecting the Metrics

This study focused on reuse metrics that aid monitoring and management of reusable assets. These metrics aid assessments of reuse benefits (cost savings, increased productivity, quality improvements, and shortened project durations) and of reusable asset utilization. The metrics best suited for this purpose measure the amount of reuse of the asset from different perspectives. A survey of general reuse metrics is found in papers by Devanbu et al. and by

Frakes and Terry.[7,8] Of the currently proposed metrics, the "amount of reuse" (AoR) metric, the External Reuse Level (ERL), the External Reuse Frequency (ERF), and the "size-frequency reuse" (R_{sf}) metric represent measures of the amount of reuse. In addition, the percentage of verbatim vs. leveraged reuse will also be examined. Bieman also proposes a set of reuse metrics specifically for object-oriented code.[9] But, these metrics provide insight into the nature of the reuse rather than the amount. Although Bieman's metrics are of interest, they're application in managing a reusable asset is unclear. Future work may include Bieman's metrics. The selected metrics are generally applicable to any reusable asset that provides code. Thus, these metrics enable comparison and knowledge sharing among organizations.

The AoR metric is the simplest and most widely used. It is the ratio of lines of code (LOC) reused from LaSRS++ to the total LOC of the simulation:

$$AoR = \frac{LOC_{framework}}{LOC_{total}} \tag{1}$$

For this metric, LOC was measured as non-comment, non-blank source lines. However, the technique used to count LOC is not important when:[8,21,22]

1) The body of code being analyzed is written according to a published style guide, or the body of code is written within the same subject domain.

2) The body of code is written in the same language.

3) The metric uses the ratio of LOC counts.

The first two criteria apply to LaSRS++ code. FSSB's developers are required to follow a published style guide whose purpose is to make all new and reused code look uniform.[14] The code in this study also falls within the single domain of aircraft simulation. The code in this study was written in one language, C++. The AoR metric meets the third criteria since it is a ratio of LOC counts. To verify that the counting technique is not important, CodeCount, an LOC counter from the University of Southern California, was run on new code and LaSRS++ code.[*] CodeCount provides both "physical" LOC (non-comment, non-blank lines) and "logical" LOC (which are representative of the number of statements). Using the results from both LaSRS++ and new code, a correlation factor between logical and physical was computed. This correlation coefficient was high, $r^2 = 0.996$, with a significance of less than 0.5%. This confirms a linear relationship between the two different counting methods for both the new code and the LaSRS++ code.

Mili et al. warn that reusable assets tend to be constructed in a manner that inflates their size relative to new code and, thus, inflates LOC-based reuse metrics.[20] This is a separate concern from the tendency of reused assets to contain more features than new code. Greater care in making the code readable and well documented can inflate size. Some reused assets generate the code, and generated code tends to be less tight than hand coding. No difference exists in how new code and LaSRS++ code are constructed; common construction methods virtually eliminate this form of inflation. The same team hand-codes both the products and the reusable assets.[†] The same style guide applies to both new and LaSRS++ code. The guide is enforced through code reviews. The code counting experiment from the previous paragraph also provides evidence that no relative inflation exists between LaSRS++ and new code. It confirms that both LaSRS++ and new code have the same average number of physical lines per statement (logical line). This finding can be applied to the files and LOC data from Table 2 to show that LaSRS++ classes also do not have more statements on average than aircraft code (122 LOC average for LaSRS++ classes vs. 225 LOC average for aircraft classes).

Frakes and Terry proposed the ERL metric.[8] This metric views the system as an aggregation of parts with different levels of abstraction. Classes, functions, and source files are different ways of decomposing the system. ERL recognizes that developers do not typically reuse individual lines of code but higher-level abstractions. In LaSRS++ projects, developers almost exclusively work with classes as the basic unit of reuse. Thus, the class was the level of abstraction chosen. In this context, ERL is the ratio of the number of classes (NOC) from LaSRS++ to the total NOC.

[*]Center for Software Engineering, CodeCount, data available online at http://sunset.usc.edu/research/CODECOUNT/, Univ. of Southern California, Dec. 2000 (cited July 2004).

[†]The "Results" section explains that a small amount of new and LaSRS++ code is auto-generated. This code was excluded from metrics were possible.

$$ERL = \frac{NOC_{framework}}{NOC_{total}} \tag{2}$$

The strength of the AoR metric is that it accounts for component size. The ERL metric does not. Since size and cost are closely related, AoR better correlates with cost savings than ERL.[8] AoR is more frequently used to derive measures that quantify reuse benefits. But, ERL provides better insight into how much of the system's decomposition into components is covered by reused components. It can act as an indicator of how well developers were able to identify and incorporate building blocks from the framework into a new program. In other words, ERL is a type of utilization measure.

A common criticism of the AoR and ERL metrics is that they give credit only once for reuse of a component.[8] However, Poulin defends the single-use credit. Since programmers should be expected to build a system from a decomposition of functions rather than as a stream of consciousness, the cost of implementing a component is saved only once.[6] Still, metrics that account for the frequency of reused code, provide other perspectives on asset utilization and insight into the depth of infrastructure offered by a framework. A system that relies greatly on reused code for much of its activity cost less to design and test than a system that relies mostly on new code.

Both the ERL and AoR metrics have frequency counterparts: the External Reuse Frequency (ERF) and the Size-Frequency Reuse (R_{sf}).[7,8] The frequency (F) of a reused item is the number of times that item is invoked in the code. For classes, it is the number of objects that the product instantiates from a class. ERF is the sum of the frequency of the reused classes divided by the sum of the frequency of all classes (i.e. new and reused):

$$ERF = \frac{\sum_{i=l}^{N_{framework}} F_i}{\sum_{i=l}^{N_{total}} F_i} \tag{3}$$

ERF can act as an indicator of reduced design time since it measures how much the system relies on pre-existing classes (i.e. designs artifacts) for its behavior. It can also act as an indicator of the reusable asset's ability to reduce maintenance. Components with a high frequency of use have been adapted to and tested in more situations. Thus, they tend to be more adaptable to future changes and more robust than components with a low frequency of use.

R_{sf} is similar to ERF except the frequency of each class is multiplied by its LOC count during the summation:

$$R_{sf} = \frac{\sum_{i=l}^{N_{framework}} (F_i * LOC_i)}{\sum_{i=l}^{N_{total}} (F_i * LOC_i)} \tag{4}$$

R_{sf} can act as an indicator of reduced testing time since it measures how much of the total program logic consists of pre-existing (and previously tested) code. Program logic is defined to be the stream of code that would result if all calls to class methods and functions were replaced with the method/function code. The test time reduction is not a result of fewer tests, because the product must still be tested against all of its requirements. Instead, the reduction manifests itself as reduced rework correcting defects because the reusable assets are less likely to fail.

As with the other metrics, static analysis cannot provide the true value of ERF or R_{sf}. But the OC method retains the ability to provide a lower bound. While the OC analysis processes source files for object creation, it can track the number of instances it encounters for a class. The previous section established that the OC method represents the lower bound of identified reused components. The OC method will also not inflate the frequency of identified components and may undercount the frequency. For example, since the OC method does not examine code branches in the reused code, it will not detect additional instances of identified reused classes in those code branches. Undercounting classes and their frequency is not a problem for the new classes. The OC method assumes that all new classes are used and it investigates all of the new code including the branches. New classes can only be instantiated in the new code; therefore, the OC implementation will not skip any instantiations of new code. The OC implementation provides accurate frequencies for new classes and potentially undercounted frequencies for instances of potentially under-identified reused code. Thus, the OC implementation retains a lower bound for frequency metrics. The DC method does not identify objects; thus, it cannot be used to compute an upper bound. Thus, the true value for ERF and R_{sf} lies between the OC values and one.

Table 2 Reuse Metrics of Base Aircraft Projects[a]

Aircraft	File Count		LOC		Class Count		AoR	ERL	ERF	R_{sf}
Name	Model	LaSRS++	Model[b]	LaSRS++	Model	LaSRS++			(OC)	(OC)
B757	309	1882 ^{DC}	102486	228082 ^{DC}	131	889 ^{DC}	69% ^{DC}	.87 ^{DC}	.94	.91
		913 ^{OC}	(1659)	116834 ^{OC}		403 ^{OC}	53% ^{OC}	.76 ^{OC}		
		1774 ^{RO}		216801 ^{RO}		835 ^{RO}	68% ^{RO}	.86 ^{RO}		
F18E	367	1805 ^{DC}	98127	215983 ^{DC}	165	851 ^{DC}	69% ^{DC}	.84 ^{DC}	.94	.89
		702 ^{OC}	(435003)	93421 ^{OC}		298 ^{OC}	49% ^{OC}	.64 ^{OC}		
		1563 ^{RO}		193388 ^{RO}		730 ^{RO}	66% ^{RO}	.82 ^{RO}		
F18E-RPV	190	1813 ^{DC}	45193	216160 ^{DC}	90	854 ^{DC}	83% ^{DC}	.91 ^{DC}	.96	.90
		696 ^{OC}	(12812)	92322 ^{OC}		294 ^{OC}	67% ^{OC}	.77 ^{OC}		
		1557 ^{RO}		192289 ^{RO}		726 ^{RO}	81% ^{RO}	.89 ^{RO}		
F18A	175	1795 ^{DC}	32434	213780 ^{DC}	81	848 ^{DC}	87% ^{DC}	.91 ^{DC}	.95	.95
		730 ^{OC}	(9658)	93559 ^{OC}		314 ^{OC}	74% ^{OC}	.80 ^{OC}		
		1591 ^{RO}		193526 ^{RO}		746 ^{RO}	86% ^{RO}	.90 ^{RO}		
F16C	97	1713 ^{DC}	19895	212794 ^{DC}	45	808 ^{DC}	91% ^{DC}	.95 ^{DC}	.98	.97
		660 ^{OC}	(27805)	92058 ^{OC}		280 ^{OC}	82% ^{OC}	.86 ^{OC}		
		1521 ^{RO}		192025 ^{RO}		712 ^{RO}	91% ^{RO}	.94 ^{RO}		
HL-20	75	1700 ^{DC}	16266	208841 ^{DC}	36	802 ^{DC}	93% ^{DC}	.96 ^{DC}	.99	.98
		721 ^{OC}	(2055)	97820 ^{OC}		311 ^{OC}	86% ^{OC}	.90 ^{OC}		
		1582 ^{RO}		197787 ^{RO}		743 ^{RO}	92% ^{RO}	.95 ^{RO}		
F16XL	84	1693 ^{DC}	11210	208632 ^{DC}	38	798 ^{DC}	95% ^{DC}	.96 ^{DC}	.98	.98
		610 ^{OC}	(2372)	84130 ^{OC}		255 ^{OC}	88% ^{OC}	.87 ^{OC}		
		1471 ^{RO}		184097 ^{RO}		687 ^{RO}	94% ^{RO}	.95 ^{RO}		
F16A	88	1711 ^{DC}	10800	214386 ^{DC}	41	805 ^{DC}	95% ^{DC}	.95 ^{DC}	.98	.98
		630 ^{OC}	(2325)	90303 ^{OC}		263 ^{OC}	89% ^{OC}	.87 ^{OC}		
		1491 ^{RO}		190270 ^{RO}		695 ^{RO}	95% ^{RO}	.94 ^{RO}		
F15A	89	1707 ^{DC}	10074	210427 ^{DC}	42	803 ^{DC}	95% ^{DC}	.95 ^{DC}	.98	.98
		650 ^{OC}	(2760)	90265 ^{OC}		273 ^{OC}	90% ^{OC}	.87 ^{OC}		
		1511 ^{RO}		190232 ^{RO}		705 ^{RO}	95% ^{RO}	.94 ^{RO}		
Generic Fighter	76	1701 ^{DC}	7954	209842 ^{DC}	34	802 ^{DC}	96% ^{DC}	.96 ^{DC}	.98	.99
		648 ^{OC}	(821)	89434 ^{OC}		274 ^{OC}	92% ^{OC}	.89 ^{OC}		
		1509 ^{RO}		189401 ^{RO}		706 ^{RO}	96% ^{RO}	.95 ^{RO}		
General Aviation	56	1698 ^{DC}	7015	208760 ^{DC}	28	801 ^{DC}	97% ^{DC}	.97 ^{DC}	.99	.99
		699 ^{OC}	(0)	94795 ^{OC}		300 ^{OC}	93% ^{OC}	.92 ^{OC}		
		1560 ^{RO}		194762 ^{RO}		732 ^{RO}	96% ^{RO}	.96 ^{RO}		

[a]DC = Dependency Chain. OC = Object Chain. RC = Refined Object Chain.
[b]Number in parentheses is the LOC of auto-generated code that contains table data. This code was excluded from the LOC count of the model.

V. Results

A. Base Aircraft Metrics

Eleven base aircraft were analyzed. These aircraft cover a wide range of configurations: transports (B757), fighters (F18E, F18E-RPV, F18A, F16C, F16XL, F16A, F15A, and Generic Fighter), advanced concept vehicles (HL-20[*]), and general aviation. Table 2 shows the reuse level metrics derived for these aircraft using the three analysis methods: dependency chain (DC), object chain (OC), and refined object chain (RO).[†] Some auto-generated

[*]A high lifting body evaluated for space crew transportation.
[†]LaSRS++ is a multi-vehicle capable framework. Metrics were computed for a simulation with one vehicle.

code was excluded from the LOC counts. Many projects use look-up tables. LaSRS++ supplies a utility that generates code from the raw table data. The utility supplies two options for loading the data: read from file or coded static arrays. Many projects load some or all of their table data via code because it provides faster startup time. Table data was excluded from LOC counts used in metrics. For completeness, Table 2 does provide the LOC for the table data in each vehicle. The total LOC of table data in the entire LaSRS++ framework is 2044 and is also excluded. The utility also auto-generates the proper object declarations and look-up code. However, the auto-generated look-up code is sometimes mixed with hand-written code. Thus, the look-up code is included in the LOC count. Since vehicle classes use more table lookups than LaSRS++ components, including the look-up code tends to depress the AoR metrics.

As discussed in "Identifying the Reused Components," the OC implementation makes some assumptions that weaken the lower bound assertion so that important object creation code will be retained to identify reused classes. A restrictive but strict implementation of the OC method was constructed to assess the maximum impact of these simplifications on the metrics. The results are shown in Table 3 for reference. The table shows the minimum, maximum, and average absolute change to each of the metrics among the eleven aircraft. The table shows that the average difference is significant but not a large departure from the result reported using the simplified OC implementation. Thus, the simplified OC implementation can reasonably be used as a lower bound and will be used to report the results.

Table 3 Metrics Differences with Strict OC Implementation

Metric	Minimum	Maximum	Average
AoR	-1.8%	-8.2%	-4.8%
ERL	-1.7%	-6.1%	-3.5%
ERF	-0.008	-0.043	-.032
R_{sf}	-0.006	-0.026	-.016

Figures 1 through 4 display the four metrics as column charts. Figure 1 displays the AoR results. The ranges of AoR are 49%- 93% OC, 66% - 96% RO, and 69%- 97% DC. The averages for AoR are 74% OC, 86% RO, and 87% DC. Figure 2 displays the ERL results. The ranges of ERL are 0.64 - 0.92 OC, 0.82 - 0.96 RO, and 0.84 - 0.97 DC. The averages for ERL are 83% OC, 91% RO, and 92% DC. Only the AoR results for B757 and F18E under the OC method show less than a two-thirds reuse level. If these results were excluded, then all base aircraft would demonstrate a greater than two-thirds level of reuse of LaSRS++ under all methods. The results show that LaSRS++ provides a source code repository, from which a simulation takes the majority of its code.

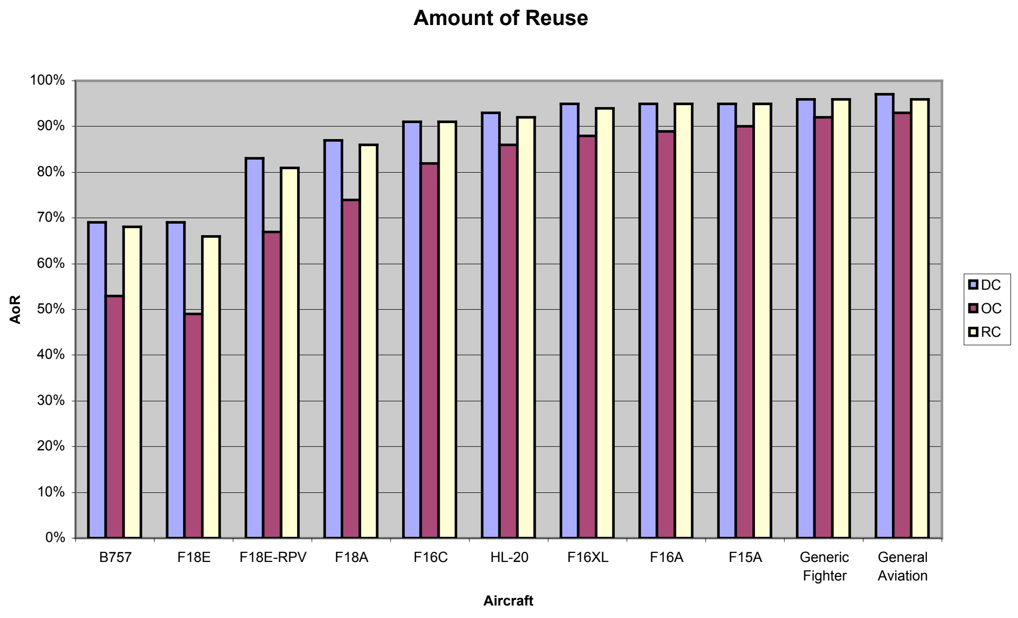

Fig. 1 Amount of reuse in LaSRS++ aircraft.

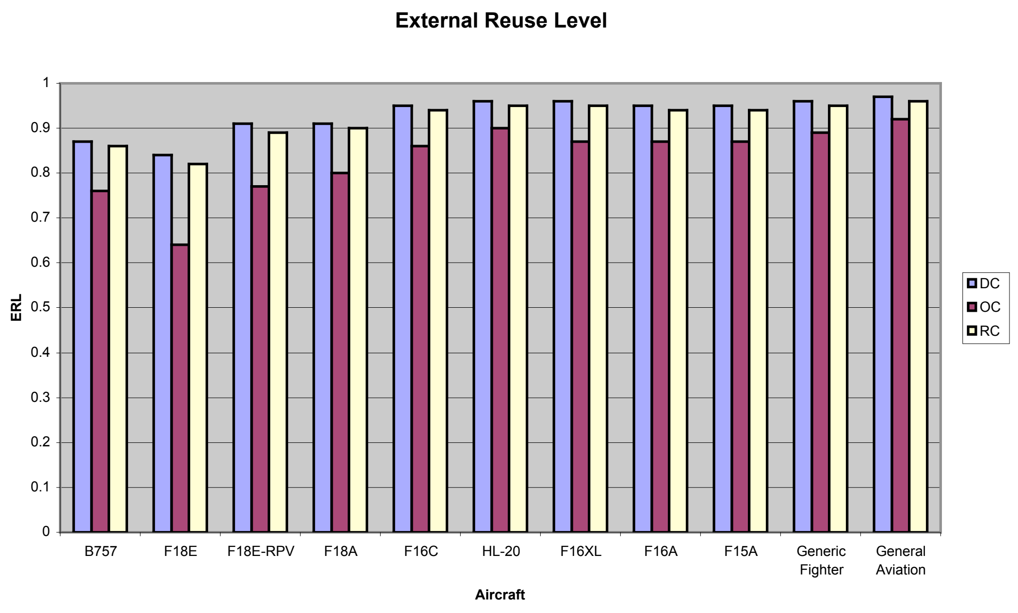

Fig. 2 External reuse level for LaSRS++ aircraft.

The RO method attempts to identify the likely neighborhood where the actual reuse value lies within the range created by the OC or DC methods. The results for the RO and DC methods are very close; the difference in average AoR is 1%. The DC method is a much simpler analysis than the OC or RO methods because it does not require parsing source code to identify reused components. For future LaSRS++ vehicles, the DC method can quickly provide an estimate of reuse level that is fairly good though over-inflated. But the other two methods must still be applied at defined intervals to watch for signals that the characteristics of LaSRS++ utilization have changed. Such changes could reduce the goodness of the DC estimate.

Level-of-reuse measures provide part of the evidence that products reuse an asset as a framework or software product line. They show that components designed as reusable are being reused. But they do not demonstrate whether the asset provides a foundation for building the product; i.e., the asset provides a core set of components that are relevant to all products. The LaSRS++ file lists were examined for the quantity of LaSRS++ files that were common to all aircraft. For each reuse identification method, Table 4 shows a list of the LaSRS++ files reused by all base aircraft. For each of the file attributes (number, classes, and LOC), Table 5 shows the percentage ranges of the common reused files to the total reused files and of common reused files to the total files (reused and new).

Table 4 Files Common to All Base Aircraft

Analysis Method	Files	Classes	LOC
Object Chain	542	225	77691
Refined Object Chain	1403	657	177658
Dependency Chain	1673	792	206954
Common OC Rejects	861	432	99967

There is a wide range of results between the identification methods with the RO and DC methods being closest in agreement. The OC method shows that the common components make up at least two-thirds and as much as 90% of the reused code in a LaSRS++ product. The RO method indicates that the common components likely make up more than 80% of the reused code. Percentages for the DC method are even higher. As a percentage of total program size, the OC method shows that LaSRS++ provides the developer with a minimum one-third of their simulation at the start. The RO and DC methods, indicate that LaSRS++ likely provides the developer with at least half of the simulation up front and may provide as much as 90% of the simulation. The length of the common file set reflects the wide range of features that LaSRS++ provides to simulations: real-time scheduling, equations of motion, environment modeling, hardware interfaces, GUI, etc. As the proportion of these common files to the total simulation demonstrates, LaSRS++ is a framework that gives developers a head start on producing simulations for LaRC's facilities.

Table 5 Common Component Percentage Ranges

Analysis Method	Common/Total Reused Files	Classes	LOC
OC	59 – 89%	56 – 88%	66 – 92%
RO	79 – 95%	79 – 96%	82 – 97%
DC	89 – 99%	89 – 99%	91 – 99%
Analysis Method	**Common/Total (Reused + New) Files**	**Classes**	**LOC**
OC	44 – 78%	42 – 77%	35 – 81%
RO	67 – 90%	68 – 91%	56 – 91%
DC	76 – 94%	78 – 95%	63 – 94%

The study uses ERF and R_{sf} to quantify how much of a product's logical structure is built upon the reusable asset. Figures 3 and 4 shows the results for the aircraft models, as estimated from the OC method. The results are the lower bound for the true value. The ERF shows that a simulation instances more than 94% of its objects from LaSRS++ classes. Only a small portion of objects is instanced from new code. The result implies that design cycles should be short, involving only a small number of objects. Experience shows that the design of LaSRS++ simulations concentrates on the unique aspects of the vehicle model. LaSRS++ handles all other concerns such as scheduling, equations of motion, environment modeling, GUI, and hardware communication. Even when other aspects of the simulation need tailoring for a simulation, the design usually involves derivation from an existing class hierarchy.

The R_{sf} shows that greater than 89% of the total program logic lies within LaSRS++ code. The LaSRS++ code has already undergone extensive testing. Thus, the developer can assume that LaSRS++ code has a low defect count.[*] Though the product must be tested against all of its requirements, simulation testing can concentrate on the new code, which likely has a higher defect rate. The developer can productively identify defects by first concentrating on the new code and before investigating the LaSRS++ code. Among the aircraft, new code makes up no more than 11% of the total system behavior. As with design, simulation testing tends to focus on the unique aspects of the vehicle model.

The high ERF and R_{sf} values reflect the structure of LaSRS++ simulations. The vehicle-only code (i.e. new code) is a small extension of a large existing infrastructure. New code tends to be an aggregation of LaSRS++ components that inherits from a LaSRS++ class hierarchy. Since new classes specialize at modeling aspects of a vehicle, a simulation typically instances a new class only once (per aircraft instance). As computed by the OC method, the average number of instances per new class ranges from 1.1 (General Aviation) to 2.7 (B757). Only the F18A, F18E, and B757 have an average of greater than two (2.3, 2.3, and 2.7 respectively). The next highest average is 1.5 for the F18E-RPV. On the other hand, the average number of instances for a LaSRS++ class ranges from 8.5 (General Aviation) to 19.0 (F18E). The high ERF and R_{sf} values result from a higher average frequency of use for LaSRS++ classes than for new classes. LaSRS++ classes are more often used than new classes as building blocks for the simulation. This is a desirable characteristic for a reusable asset. The ERF and R_{sf} metrics corroborate that LaSRS++ provides a structural foundation for simulations, not just a code repository.

[*]Software with the complexity of the LaSRS++ framework is rarely defect-free.

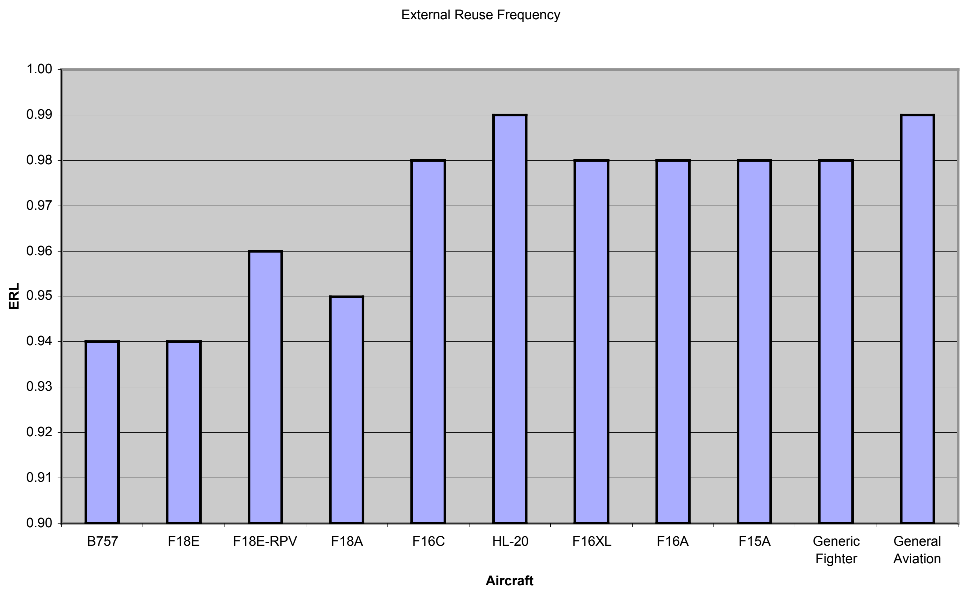

Fig. 3 External reuse frequency for LaSRS++ aircraft.

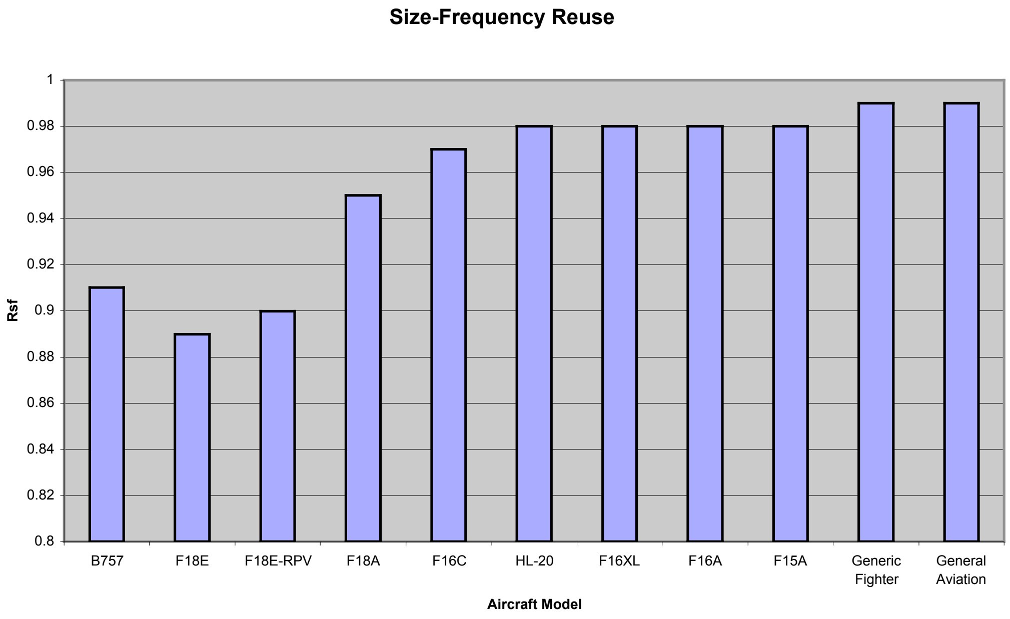

Fig. 4 Size frequency reuse for LaSRS++ aircraft.

Vehicle models can reuse LaSRS++ in one of two forms: leveraged (via inheritance) and verbatim. Since it examines the code for inheritance and object creation, only the OC method was capable of providing an estimated categorization of LaSRS++ reuse. Of the reused LaSRS++ classes, the vehicles reuse 6 - 11% through leverage (i.e. inheritance). The vehicles reuse 93 - 96% verbatim. The two ranges combine to greater than 100% because some classes are reused both verbatim and through leverage. Verbatim reuse is clearly the predominant form of reuse for vehicle models. The large percentage of verbatim reuse reinforces the depiction of vehicle models as a small extension to a large underlying infrastructure.

B. Aircraft Variant Metrics

A wide variety of research projects may use the base aircraft to conduct experiments. Research projects have four requirements for working with the base aircraft:

1) Research projects must be able to modify any aspect of the aircraft for the experiment.
2) Research projects must be isolated from each other. Changes from one project cannot unintentionally affect the results of another project.
3) Elevating project enhancements to the base aircraft should be simple.
4) Research projects must have access to the latest changes made to the base aircraft.

Research variants of an aircraft adhere to these requirements by relying on inheritance to manage the aircraft as a reusable resource. Figure 5 illustrates a high-level design view of variants. Each variant inherits from the base aircraft. All specialization is done within the derived class. Polymorphism is used to redefine any behaviors inherited from the base aircraft. Inheritance allows projects to modify the base aircraft while maintaining project isolation. If an update is made to the base aircraft, all variants immediately inherit it. Because modifications by the variant are made within the base aircraft structure that it inherits, elevating the changes into the base aircraft is usually straightforward. The same design is also used to create different aircraft types in the same aircraft family where substantial overlap exists in the how the aircraft are modeled. For example, the F18TV is modeled as a thrust-vectoring variant of the F18A. These are called configuration variants. Variants created for an experiment are called research variants. Inheritance provides an additional advantage that enables reuse of the base aircraft. Because the base aircraft specialize LaSRS++ through inheritance, the base aircraft reflect the adaptable architecture of LaSRS++. Specialization of a base aircraft by a variant uses a very similar procedure to specialization of LaSRS++ by the base aircraft. A detailed discussion of vehicle architecture in LaSRS++ is beyond the scope of this paper. The reader can find a more detailed discussion in Ref. 13.

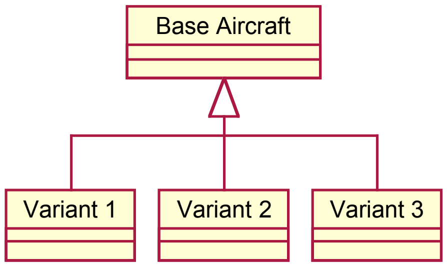

Fig. 5 Design of aircraft variants.

Reuse level metrics were collected for ten aircraft variants. Of the variants, only two (F18C and F18TV) are configuration variants; the other eight are research variants. The metrics focus on the reuse of the base aircraft by the variant; the metrics ignore LaSRS++ code. Unlike LaSRS++, the base aircraft is not a generic framework with classes that may be applicable to only a subset of its applications. All classes in the base aircraft are relevant to each instance of the vehicle. Thus, the DC method provides accurate class lists for vehicle code and was the only identification method used to measure reuse in variants.

Reuse level metrics measure how well the design of variants promotes reuse of the base aircraft. The results are shown in Table 2. The AoR values range from 60% to 99% with all but the F18C at 70% or greater. The ERL values range from .61 to .94. The research variants tend to introduce small modifications to the base aircraft. The AoR for the research variants is 73% or greater, with all but the F16A-FL at 90% or greater. The configuration variants introduce a larger body of modifications. Configuration variants usually introduce major changes to the aerodynamic model, engine model, control surface models, and/or control law. The AoR for the two configuration variants is 60% (F18C) and 70% (F18TV). Overall, a variant relies on the base aircraft for the majority of its code. The AoR and ERL results for aircraft variants are comparable to the AoR and ERL results for base aircraft reuse of LaSRS++. The similarity indicates that inheritance can effectively flow down architectures designed for reuse.

Table 6 Reuse Metrics of Aircraft Variants

Name	Parent	Classes	LOC	AoR	ERL	Leveraged
B757-ANOPP	B-757	9	1927	98%	.94	1.5%
B757-CTAS	B-757	10	1358	99%	.93	3.0%
B757-RIPS	B-757	9	533	99%	.94	1.5%
B757-VISTAS	B-757	12	6973	94%	.92	3.0%
B757-WXAP	B-757	33	10070	91%	.80	3.0%
F16A-FL	F16A	22	4016	73%	.65	39.0%
F18E-AWS	F18E	17	2267	98%	.91	7.9%
F18A-SRA	F18A	21	3611	90%	.79	18.5%
F18C	F18A	51	21813	60%	.61	9.9%
F18TV	F18A	41	14089	70%	.66	27.2%

Although variants are derived from the base aircraft, the variants do not necessarily reuse a significant portion of the base aircraft through leverage. The design of variants only requires leveraged reuse of one class, the class representing the base aircraft. For example, the only requirement for the B757-ANOPP vehicle is that the B757Anopp[*] class derives from the B757Base class. The percentage of the base aircraft classes reused through leverage varies greatly from 1.5% to 39%. The variants can be divided into two groups, one group that has very a low percentage of leveraged reuse (<5%) and one that has a moderate percentage of leveraged reuse (>5%).

A very low percentage of leveraged reuse indicates that the variant primarily extends the base aircraft. In other words, the majority of the variant's code adds new features to the base aircraft, and very little of the base aircraft behavior is redefined. All of the variants in this category are research variants. A moderate percentage of leveraged reuse signifies that the variant redefines behavior in one or more subsystems of the base aircraft. A mixture of research and configuration variants falls into this category. While maintaining beneficial levels of reuse, the design of variants in LaSRS++ accommodates both variants that primarily add behavior to base aircraft and variants that moderately redefine base aircraft behavior.

C. Comparison with Other Reports

Capers Jones explores the potential reuse ratios that can be achieved.[23] He cites a couple of unpublished studies. One study of commercial banking and insurance products in California noted that 75% of the functions occurred in more than one program. A second study observed that in similar commercial products, less than 30% of the code was concerned with the actual problem the product was written to solve. Jones then presents 85% as "the target which researchers into reusable code are attempting to standardize".[27] These percentages are based on examination of one-of-a-kind programs. So the reuse percentages would represent an ideal where the reused code was wholly used by the product. As discussed in "Identifying the Reused Components", AoR based on assets designed for reuse will be inflated because not all products will reuse the full capability of the assets. Accounting for this inflation, the average LaSRS++ AoR computed by all three methods (74% OC, 85% RO, and 86% DC) compares well to the ideal. LaSRS++ appears to capture the bulk of commonality that is expected between LaRC simulation programs, but room for improvement likely remains.

Comparison with actual results from other reuse programs may indicate whether better reuse performance could have been possible if different processes or people were used. An ideal comparison would be with other programs in the simulation domain that identified reuse using methods comparable with this study. But, no repository of reuse results exists from which we could expect to get data for matching programs.[24] The literature has a scattering of results from various application domains that differ in how they identify reuse. The differences are not large enough to prevent general comparisons. The results selected for comparison are all highly successful, systematic-reuse programs with similarities to the software product-line development approach. Each study examines reusable assets that were developed internally to support the mission of a single organization.

Lanergan and Grasso provide one of the earliest and most widely cited results from a systematic reuse program. They report an average AoR of 60% for business software developed by Raytheon Missile Systems Division in the late 1970s.[25] Lim reports on reuse programs that began in the 1980s at three HP organizations. One organization wrote software for manufacturing resource. The second developed firmware for plotters and printers. The third produced unidentified firmware. Selected case studies for quality, productivity, and time-to-market analysis from the

[*]Capitalization follows LaSRS++ style guidelines for class names.

first two organizations had AoR values of 31%, 38%, and 68% (Ref. 3). The third organization achieved AoR values of 71%, 81%, and 77% in their last three products.[3] SEI reports that Cummings, a manufacturer of diesel engines, was able to achieve an average AoR of 75% for their engine software using a software product-line approach.[26] SEI also reports that Motorola achieved 80% average AoR by applying the software product-line approach to their family of one-way pagers.[30] The Software Engineering Laboratory (SEL) at NASA Goddard ran a reuse program that comes closest to LaSRS++ in application domain. The SEL collected data on flight dynamics projects. For the years 1990 to 1993, the SEL observed a 79% average AoR.[2] In 1998, Robert Glass reported that the SEL achieved a 75% average AoR for object-oriented software and a 70% average AoR for non-object-oriented software.[27] Even the lower AoR average of 74% reported by the object-chain method for LaSRS++ falls into the range of these reports. LaSRS++ achieves real-world reuse performance equivalent to other high-success reuse programs.

Unfortunately, few reports provide reuse measures other than AoR. Devanbu et al. provide reuse-size frequency, reuse level, and reuse frequency measures for the code written for Basili's 1996 paper on reuse.[1,7] However, two major differences between Devanbu's analysis and the analysis of LaSRS++ prevents direct comparison. Devanbu's measures included "internal" and "external" reuse; LaSRS++ counts only "external" reuse with LaSRS++ as the sole external asset. Basili's experiment also modeled "opportunistic" reuse rather than systematic reuse; the AoR levels of Basili's study were much lower than those reported in this paper.

This study provides one of the first treatments of ERL and frequency statistics for a large, reusable framework. The AoR comparison lends some credence that the results of this study can be viewed as characteristics of a well-constructed framework.

VI. Conclusions

Organizations running systematic reuse programs need reuse measures to manage the reusable assets and make informed decisions. Reuse measures must be automated for regular use. The first step in measuring reuse among products is identifying the assets that are reused. Large-scale reusable assets like frameworks and software product lines can make automated identification difficult when their design does not allow physical separation of unused features. Weeding out unused features is desirable to reduce inflation of the reuse measures. Three static code analysis methods were developed at NASA Langley Research Center to identify the reused components from the LaSRS++ framework. The object chain (OC) and dependency chain (DC) were designed to provide a lower and upper bound for the set of identified components. A large gap exists between the two methods when they are applied to products reusing the Langley Standard Real-Time Simulation in C++ (LaSRS++). In response, the refined object chain method (RO) was developed as an indicator of the likely set of reusable assets. The identification rules for the three methods were designed so that they could be generally applied to other reusable assets and described in detail to enable comparison by other organizations.

The set of reused components identified by the three methods were used as inputs to reuse metrics. The metrics were selected that aid assessments of reuse benefits and asset utilization. The metrics measured reuse level and reuse frequency with both units of line-of-code (LOC) and class. The study computed reuse metrics for eleven aircraft models produced using LaSRS++. The averages among the three identification methods showed that LaSRS++ represents somewhere in the range of 74% - 86% of the total product LOC and more than 82% of the total classes. The results quantify the code foundation that LaSRS++ provides for building simulations at LaRC. The reuse frequency metrics measure the logical contribution of that foundation. More than 94% of all objects in a simulation are created from LaSRS++ classes. Thus, product design can focus on the definition of less than 6% of a simulation's objects. More than 89% of the program logic resides within LaSRS++ code. Thus, the majority of defects should surface in as little as 11% of the program logic. The results of the LOC-based reuse level compare well to theoretical targets of potential reuse and reports from other highly successful reuse programs. The comparison identifies LaSRS++ as a successful reusable framework. This study is among one of the first detailed treatments of the component-based (i.e. class-based) and frequency-based metrics on a large reusable framework. Thus, results using the other three metrics are held as an example of what is achievable and characteristic of a good systematic reuse program.

The reuse measurement techniques were also applied to variants of aircraft models. Variants are specializations of an aircraft model for an experiment or to model a different aircraft type within the same aircraft family. These measures were used to assess the effectiveness of inheritance to flow down the adaptable LaSRS++ architecture to the base aircraft so that the base aircraft can be treated as reusable assets for research products. Reuse level metrics were extracted from ten existing variants. The metrics show that 60% - 99% of a variant is composed of base aircraft code. These levels are comparable to what the base aircraft achieve in reusing LaSRS++ and demonstrate success in flowing the reusable architecture into the base aircraft. The amount of leveraged reuse among variants demonstrates that architecture flow-down works equally well for variants that simply add new behaviors and for variants that redefine base aircraft behaviors. This examination reveals another good characteristic for a reusable framework, the ability to extend its adaptable architecture into the derived products.

References

[1]Basili, V. R., Briand, L. C., and Melo, W. L., "How Reuse Influences Productivity in Object-Oriented Systems," *Communications of the ACM*, Vol. 39, No. 10, Oct. 1996, pp. 104-116.

[2]McGarry, F., Pajerski, R., Page, G. Waligora, S. Basili, V., and Zeikowitz, M., "Software Engineering Process Improvement in the NASA Software Engineering Laboratory," Technical Report, CMU/SEI-94-TR-22, Software Engineering Institute at Carnegie Mellon Univ., Dec. 1994.

[3]Lim, W. C., "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software*, Vol. 11, No.5, pp. 23-30, Oct. 1994.

[4]Isoda, S., "Experiences of a Software Reuse Project," *Journal of Systems and Software*, Vol. 30, Issue 3, Sept. 1995, pp. 171-186.

[5]Frakes, W. B., and Fox, C. J. "Modeling Reuse Across the Software Lifecycle," *Journal of Systems and Software*, Vol. 30, Issue 3, Sept. 1995, pp. 295-301.

[6]Poulin, J., Caruso, J., and Hancock, D., "The Business Case for Software Reuse," *IBM Systems Journal*, Vol. 32, No. 4, 1993 pp. 567-594, 1993.

[7]Devanbu, P., Karstu, S., Melo, W., and Thomas, W., "Analytical and Empirical Evaluation of Software Reuse Metrics," *Proceedings of the 18th International Conference on Software Engineering*, 1996, pp. 189-199.

[8]Frakes, W., and Terry, C., "Software Reuse and Reusability Metrics and Models," Technical Report, TR-95-07, Virginia Polytechnic Institute and State Univ., Blacksburg, VA, 1995.

[9]Bieman, J., "Deriving Measures of Software Reuse in Object Oriented Systems," Technical Report, CS-91-112, Colorado State Univ., Fort Collins, CO, July 1991.

[10]Leslie, R., Geyer, D., Cunningham, K., Madden, M., Kenney, P., and Glaab, P., "LaSRS++: An Object-Oriented Framework for Real-Time Simulation of Aircraft," AIAA-98-4529, Aug. 1998.

[11]Krueger, C., "Software Reuse," *ACM Computing Surveys*, Vol. 24, No. 2, June 1992, pp. 132-183.

[12]Sparks, S., Benner, K., and Faris, C., "Managing Object-Oriented Framework Reuse," *IEEE Computer*, Vol. 29, No. 9, Sept. 1996, pp. 52-61.

[13]Madden, M. M., and Neuhaus, J., "A Design for Composing and Extending Vehicle Models," AIAA-2003-5458, Aug. 2003.

[14]Unisys Corporation, *The LaSRS++ Programmer's Guide*, Internal Document, Contract NAS1-20454, April 2002.

[15]Prieto-Diaz, R., "Implementing Faceted Classification for Software Reuse," *Communications of the ACM*, Vol. 34, No. 5, May 1991, pp. 88-97.

[16]Kenney, P. S., "Simulating the ARES Aircraft in the Mars Environment," AIAA-2003-6579, Sept. 2003.

[17]Kenney, P. S., Leslie, R. A., Geyer, D. W., Madden, M. M., Glaab, P. C., and Cunningham, K., "Using Abstraction to Isolate Hardware in an Object-Oriented Simulation," AIAA-98-4533, Aug. 1998.

[18]Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[19]Leach, R., "Methods of Measuring Software Reuse for the Prediction of Maintenance Effort," *Software Maintenance: Research and Practice*, Vol. 8, 1996, pp. 309-320.

[20]Mili, H., Mili, F., and Mili, A., "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, June 1995, pp. 528-561.

[21]Rossenberg, J., "Some Misconceptions About Lines of Code," *Proceedings of the Fourth International Software Metrics Symposium*, 1997, pp. 137-142.

[22]Rothenberger, M., and Hershauer, J., "A Software Reuse Measure: Monitoring an Enterprise-Level Model Driven Development Process," *Information and Management*, Vol. 35, 1999, pp. 283-293.

[23]Jones, C., "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, Sept. 1984, pp. 488-494.

[24]Rothenberger, M. A., Dooley, K. J., Kulkarni, U. R., and Nada, N., "Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices," *IEEE Transactions on Software Engineering*, Vol. 29, No. 9, Sept. 2003, pp. 825-837.

[25]Lanergan, R. G., and Grasso, C. A., "Software Engineering with Reusable Designs and Code," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, Sept. 1984, pp. 498-501.

[26]Clements, P., "Software Product Lines: Concepts and Challenges," presentation at the International Colloquium of the SFB 501, Kaiserslautern, March 2003.

[27]Glass, R. L., "Reuse: What's Wrong with This Picture?" *IEEE Software*, Vol. 15, No. 2, March/April 1998, pp. 57-59.

[28]Booch, G., *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings Publishing Company, Inc. New York, 1994, pp. 27-80.