AIAA-2001-4119

EXAMINING REUSE IN LASRS++-BASED PROJECTS

Michael M. Madden[*]

NASA Langley Research Center
MS 125B
Hampton, VA 23681

Abstract

NASA Langley Research Center (LaRC) developed the Langley Standard Real-Time Simulation in C++ (LaSRS++) to consolidate all software development for its simulation facilities under one common framework. A common framework promised a decrease in the total development effort for a new simulation by encouraging software reuse. To judge the success of LaSRS++ in this regard, reuse metrics were extracted from 11 aircraft models. Three methods that employ static analysis of the code were used to identify the reusable components. For the method that provides the best estimate, reuse levels fall between 66% and 95% indicating a high degree of reuse. Additional metrics provide insight into the extent of the foundation that LaSRS++ provides to new simulation projects.

When creating variants of an aircraft, LaRC developers use object-oriented design to manage the aircraft as a reusable resource. Variants modify the aircraft for a research project or embody an alternate configuration of the aircraft. The variants inherit from the aircraft model. The variants use polymorphism to extend or redefine aircraft behaviors to meet the research requirements or to match the alternate configuration. Reuse level metrics were extracted from 10 variants. Reuse levels of aircraft by variants were 60% - 99%.

[*] Senior Member, AIAA

Acronyms

AoR	Amount of Reuse Metric
DC	Dependency Chain Method
ERF	External Reuse Frequency Metric
ERL	External Reuse Level Metric
F	Frequency
GUI	Graphical User Interface
LaRC	Langley Research Center
LaSRS++	Langley Standard Real-Time Simulation in C++
LOC	Lines of Code
NOC	Number of Classes
OC	Object Chain Method
RO	Refined Object Chain Method
R_{sf}	Size-Frequency Reuse Metric

Introduction

The LaSRS++ project had three main goals:

1. Create one simulation framework from which developers could build both single-vehicle and multi-vehicle simulations for a variety of aircraft. This goal aimed to increase developer utilization. In the 1980's, each simulator at LaRC had its own code base. With one common framework, developers no longer had to undergo significant training to move from one simulator to the next.
2. Move vehicle models between simulators with minimal additional development. With separate code bases, vehicle models had to be rewritten and re-tested to move to another simulator.
3. Encourage software reuse. With separate code bases leading to segregated development teams, reinvention of features was not only common but sometimes necessary. Designed as an object-oriented framework, LaSRS++ provided a large number of generic components that developers could reuse when writing a new simulation. When

possible, new features are made into generic components and added to the framework for the benefit of future projects.

The first two goals defined project success. When the first two goals were demonstrated, LaRC adopted LaSRS++ as its standard simulation framework.[1,2] Software reuse could not be factored into the adoption decision because it could not be measured until several successful projects were created using LaSRS++.

This study examines 21 aircraft simulation projects that were built using LaSRS++. These projects fall into two categories: standalone aircraft models (11) or variants of the standalone aircraft (10). Some variants represent alternate configurations of an aircraft; for example, the F18C is modeled as a variant of the F18A. Other variants are created to support a specific research project. For example, B757-WXAP is a variant of the B757 for the Weather and Accident Prevention (WxAP) project.

To determine the success of LaSRS++ as a reusable framework, the standalone aircraft models (a.k.a. base aircraft) were examined. Static analysis techniques were applied to the code to produce a list of LaSRS++ components that the base aircraft reuse. This list was then applied to several reuse metrics to provide a measure of reuse for each aircraft.

As they do with the LaSRS++ framework, LaRC developers employ object-oriented design to manage the base aircraft as a reusable resource for the variants. To judge the success of employing this design choice, reuse metrics are also applied to the aircraft variants. This analysis focuses on the base aircraft as the pre-existing code base and ignores LaSRS++ code.

Categorizing 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:[3]

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.[4] Bieman also uses a single category for adapted code but calls it "leveraged."[5] Both studies use the term "verbatim" in place of "transported."[4,5] 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.[4] (Reuse from a pre-existing code base is "external" reuse.) Since this study attempts to measure the success of LaSRS++ (and base aircraft) as a reusable framework, internal reuse is ignored.

Under the LaSRS++ development process, developers are not allowed to copy and modify LaSRS++ code to produce a vehicle-only variant. If a vehicle requires data and/or behavior modifications to a LaSRS++ class, the developer must use inheritance and polymorphism to make the changes.[†] In other words, the vehicle-only specialization must inherit from the LaSRS++ class. Since no reused LaSRS++ code is modified, all LaSRS++ reuse would be transported (a.k.a. verbatim) under CSC's definitions. However, Bieman views inheritance and polymorphism as language-supported mechanisms for modifying reused code.[5] 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 LaSRS++ parent are also counted as "leveraged" reuse. All other reuse is verbatim. Thus, if new code exercises verbatim reuse of a LaSRS++ class, then all ancestors of the LaSRS++ class are also counted as verbatim reuse.

The same rules also apply to variants of base aircraft. The developer is not allowed to copy and modify the base aircraft code. The developer must use inheritance

[†] 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.[6]

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.

Methods

Identifying the Reused Components

The first step in measuring reuse of LaSRS++-based projects is to identify the reused LaSRS++ components. Getting an accurate list of reused components is problematic. The only accurate means is to instrument the code for call chain analysis and to create a set of runs that provide 100% code coverage of the vehicle source files and exercises all paths in the LaSRS++ code relevant to the vehicle. For software with the complexity of LaSRS++-based simulations, developing a complete set of runs is very time consuming and nearly impossible. A static analysis of the source is a more economical means of obtaining data. But static analysis provides, at best, an estimate of the reused components. For this study, two static analysis methods were selected to provide a lower and upper bound of reused components. A method provides an upper bound if it selects all components reused by the project and may possibly falsely identify extra components. A method provides a lower bound if it does not falsely select any components but, in the process, may overlook some reused components.

The dependency chain (DC) and the object chain (OC) were selected to provide these bounds. The dependency chain identifies all of the source files that are required to build a simulation containing the aircraft model. It provides the upper bound. The object chain method statically analyzes the code for the creation of objects. It counts only the classes and ancestor classes of identified objects. This method provides a lower bound when it ignores objects created within conditional statements in LaSRS++ code.[‡]

The dependency chain 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 the executable. The DC method falsely adds 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. The DC method identifies all reused files and may falsely identify some files as reused. Thus, reuse measures derived from the DC method represent an upper bound.

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 are examined for the list of LaSRS++ objects that they create. LaSRS++ files related to the object list are then analyzed for a list of objects that they will unconditionally create. The last step is repeated until the object chain is exhausted. 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 conditional statements in LaSRS++ code so that it will not falsely identify classes as reused. But, this may also cause the method to miss valid, reused classes.

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

1. An object (but not a pointer or reference) of the class is declared.
2. An object of the class is allocated using the operator `new`.
3. The class is a superclass of a previously instanced class.
4. A method of a previously instanced 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.

[‡] "if-else" or "switch" statements.

[§] This was a welcome side effect of the search pattern for object declarations.

The OC method finds most, but not all classes that the simulation will instance. It will fail to identify classes instanced only under the following conditions:

1. A template instances one of its type arguments. LaSRS++ has some classes that are instanced only by templates.
2. LaSRS++ code creates the object within a conditional statement. As stated above, this is by design.
3. A method of a previously instanced class uses pass-by-value for an argument of the class type. This is rare in LaSRS++ projects. Its rarity did not justify the effort to make a reliable search for it.

The OC method will not identify classes, which the project does not use; but it may overlook classes that the project does use. Thus, it serves as a good lower bound for reuse measures.

The true measure of reuse will lie between values computed from the OC and DC methods. The results in **Table 1** show that there is a large range between the two techniques. 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 measure 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 aircraft share a common set of 861 files. Thirty-six percent (309 files) are primitive classes that are not specific to the domain of simulation at LaRC. These classes are used to build the domain-specific math models (32 files), system interfaces (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 system interfaces (22%). An inspection of these files reveals a host of components that are conditionally built based on runtime options and cockpit selection. Overall, a minority of the commonly rejected files is conditionally built based on the presence of select components in the vehicle. Therefore, all aircraft models reuse most of the commonly rejected files. Adding them to the object chain's list will result in a list that provides a better estimate of reuse. 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. However, metrics derived from the refined list are more likely to be in the neighborhood of the true value than metrics derived from either the dependency chain or object chain lists.

Selecting the Metrics

This study was focused on answering one basic question. Does LaSRS++ succeed as a reusable framework? This question governed the selection of the metrics. Metrics that measure the amount of LaSRS++ reuse in a simulation are pertinent to the question. A survey of general reuse metrics is found in papers by Devenbu (et. al.) and by Frakes and Terry.[4,7] 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 most directly address the question. Bieman also proposes a set of reuse metrics specifically for object-oriented code.[5] But, these metrics provide insight into the nature of the reuse rather than the amount. Although Bieman's metrics are of interest, they do not directly answer the question. Future work may include Bieman's metrics.

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:

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

For this metric, LOC was measured as non-comment, non-blank source lines. However, the exact technique used to count LOC is not important when:[7,8,9]

a. 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.
b. The body of code is written in the same language.
c. The metric uses the ratio of LOC counts.

Table 1 Reuse Metrics of Base Aircraft Projects¶

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

All three criteria apply to LaSRS++-based code and the AoR metric.

¶ DC = Dependency Chain. OC = Object Chain. RC = Refined Object Chain.

[#] Number in parenthesis is the LOC of auto-generated code that contains table data. This code was excluded from the LOC count of the model.

Some auto-generated code was excluded from the LOC count. 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. This code was excluded from LOC counts used in metrics. For completeness,

Table 1 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, this auto-generated code is sometimes mixed with hand-written code. Thus, it is included in the LOC count. Since vehicle code uses far more table lookups than LaSRS++ components, including this code tends to depress the AoR metrics.

Frakes and Terry proposed the ERL metric.[4] 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++-based 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, the metric is the ratio of the number of classes (NOC) from LaSRS++ to the total NOC.

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

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.[7] 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.

A common criticism of the AoR and ERL metrics is that they give credit only once for reuse of a component.[7] 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.[10] Still, metrics that account for the frequency of reused code, provide insight into the depth of infrastructure offered by the 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}).[4,5] 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 of the class that the program creates. 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):

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

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) for its behavior. It can also act as an indicator of reduced 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:

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

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 pre-tested) code.

As with the other metrics, the static analysis cannot provide the true value of ERF or R_{sf}. But the OC method can 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 OC method has two limitations, it captures only a subset of the reused classes and it undercounts their frequency. The manner in which the OC method identifies objects guarantees an undercount of LaSRS++ objects. For example, the OC method counts only one object when an array of objects is declared or allocated. When the OC method counts a derived class instance, it

[**] 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.

does not also count an instance of the ancestors. Undercounting is not a problem for the vehicle-only classes. Since the amount of vehicle-only code is much smaller, instance counts can be verified using inspection. Since the DC method does not identify objects, it cannot be used to compute an upper bound. Thus, the maximum possible value is below one for ERF and R_{sf}.

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 1** 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).[‡‡] The ranges of AoR are 49%- 93% OC, 66% - 96% RO, and 69%- 97% DC. The ranges of ERL are 0.64 – 0.92 OC, 0.82 – 0.96 RO, and 0.84 – 0.97 DC. Only the AoR results for B757 and F18E under the OC method show less than a two-thirds reuse level. As stated earlier, the OC greatly undercounts the number of reused classes. If these results were excluded, then all base aircraft demonstrate a greater than two-thirds level of reuse of LaSRS++. LaSRS++ provides a source code repository from which a simulation takes the majority of its code.

As stated earlier, the author has greater confidence in the RO numbers than the OC or DC numbers. The results for the RO and DC methods are very close, within less than 2% for all aircraft. The DC method is a much simpler analysis than the OC or RO methods. Thus, for future LaSRS++-based vehicles, the DC method can quickly provide an estimate of reuse level that is fairly good though over-inflated.

The level of reuse provides part of the evidence that simulations use LaSRS++ as a framework. It shows that components designed as reusable are being reused. But it does not demonstrate whether LaSRS++ provides a foundation for building simulations; i.e., it provides a core set of components that are relevant to all aircraft simulations at LaRC. The file lists were examined for the quantity of files that were common to all aircraft. For each reuse identification method, **Table 2** shows a list of the LaSRS++ files reused by all base aircraft. For each of the file attributes (number, classes, and LOC), **Table 3** shows the percentage ranges of the common reused files to the total reused files and to the total files (reused and new).

Table 2 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

Table 3 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%

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. 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,

[††] 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.

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.

The study uses ERF and R_{sf} to quantify how much of a simulation's logical structure is built upon LaSRS++. **Table 1** shows the results, 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 design of LaSRS++-based 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 a simple 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.[§§] Simulation testing can concentrate on the new code, which 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 majority of errors that occur during testing should originate from new code. The developer can productively identify defects by first concentrating on the new code and ignoring LaSRS++ code.

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.[¶¶] 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 per 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. The ERF and R_{sf} metrics corroborate that LaSRS++ provides a large structural foundation for simulations, not just a code repository.

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 by the vehicles. Of the reused LaSRS++ classes, the vehicles reuse 6 - 11% through leverage (i.e. inheritance). The vehicles reuse 93 – 96% verbatim. The two ranges do not combine to 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.

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.

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

[¶¶] per aircraft instance.

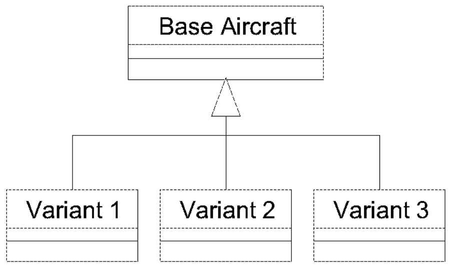

Figure 1 Design of Aircraft Variants

Research variants of an aircraft adhere to these requirements by relying on inheritance to manage the aircraft as a reusable resource. Figure 1 illustrates the standard design for 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 alternate configurations of the aircraft. For example, the F18TV is thrust vectoring variant of the F18A.

Reuse level metrics were collected for ten aircraft variants. Of the variants, only the last 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. All classes in the base aircraft are relevant to each instance of the vehicle. Thus, the DC method provides accurate file lists for vehicle code and was the only identification method used.

Reuse level metrics measure how well the design of variants promotes reuse of the base aircraft. The results are shown in Table 4. 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, the variants rely on the base aircraft for the majority of its code.

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 base aircraft class, the class representing the vehicle. 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 that has very a low percentage of leveraged reuse (<5%) and one that has a moderate percentage of 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 systems of the base aircraft. A mixture of research and configuration variants falls into

Table 4 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%

this category. While maintaining beneficial levels of reuse, the design of variants accommodates both variants that primarily add behavior to base aircraft and variants that moderately redefine base aircraft behavior.

Conclusions

To study the effectiveness of LaSRS++ as a reusable framework, this study computed reuse metrics for eleven vehicle models produced using LaSRS++. Three static code analysis methods were used to identify the reused LaSRS++ components: object chain (OC), refined object chain (RO), and dependency chain (DC). The OC and DC methods provide a lower and upper bound for the estimated reuse metrics respectively. But, the OC method severely undercounts the reused files. The RO method, which attempts to account for files the OC method overlooks, was considered by the author to be the best estimator of reused components. Since the DC results are close to the RO results, the DC method, which is the easiest to compute, can provide a good, quick estimate of reuse for future LaSRS++ vehicles.

Reuse level and reuse frequency metrics were measured at the LOC and class level. The reuse level metrics show that LaSRS++ succeeds as a repository of reusable components. According to the RO results, LaSRS++ makes up more than two-thirds of the total LOC and more than 82% of the total classes. Furthermore, the component lists revealed a core set of LaSRS++ components that are common across all aircraft. This core set makes up at least 80% of the reused components and at least 56% of the total program size. Thus, LaSRS++ provides a code foundation for building simulations at LaRC. The reuse frequency metrics demonstrate that this foundation extends beyond program size. LaSRS++ provides a solid structural foundation. The majority of a simulation's logical structure comes from LaSRS++. More than 94% of all objects in a simulation are created from LaSRS++ classes. Thus, design cycles involve the definition of less than 6% of a simulation's objects. More than 89% of the program logic resides within LaSRS++ code. Testing can concentrate on as little as 11% of the program logic. Of the classes that a vehicle reuses, more than 93% are reused verbatim and less than 11% are leveraged through inheritance. These percentages corroborate that aircraft models are a small extension to a larger infrastructure supplied by LaSRS++.

When creating variants of a base aircraft, LaRC developers use object-oriented design to manage the base aircraft as a reusable resource. Variants inherit from the base aircraft. Reuse level metrics were extracted from ten existing variants. The metrics show that 60% - 99% of a variant is composed of base aircraft code. The amount of leveraged reuse among variants demonstrates that the variant design works equally well for variants that simply add new behaviors and for variants that redefine base aircraft behaviors. The variant design successfully enables the tailoring of existing aircraft models.

Bibliography

[1] R. Leslie, D. Geyer, K. Cunningham, M. Madden, P. Kenney, and P. Glaab. *LaSRS++: An Object-Oriented Framework for Real-Time Simulation of Aircraft.* AIAA-98-4529, Modeling and Simulation Technology Conference, Boston, MA, August 1998.

[2] P. Kenney, et. al. *Using Abstraction to Isolate Hardware in an Object-Oriented Simulation.* AIAA Modeling & Simulation Technologies Conference, Boston, August 1998, AIAA-98-4533.

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

[4] W. Frakes and C. Terry. *Software Reuse and Reusability Metrics and Models.* Technical Report, TR-95-07, Virginia Polytechnic Institute and State University, 1995.

[5] J. Bieman. *Deriving Measures of Software Reuse in Object Oriented Systems.* Technical Report, CS-91-112, Colorado State University, 1991.

[6] G. Booch. Object-Oriented Analysis and Design with Applications. Benjamin/Cummings Publishing Company, Inc. NewYork, 1994. ISBN 0-8053-5340-2.

[7] P. Devanbu, S. Karstu, W. Melo, and W. Thomas. *Analytical and Empirical Evaluation of Software Reuse Metrics.* Proceedings of the 18th International Confer-

ence on Software Engineering, Berlin, Germany, p 189-199, 1996.

[8] J. Rossenberg. *Some Misconceptions About Lines of Code.* Proceedings of the Fourth International Software Metrics Symposium, Albuquerque, NM, p 137-142, 1997.

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

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