Quantifying the Costs of COTS Volatility

Richard D. Stutzke

Science Applications International Corp.
6725 Odyssey Drive
Huntsville, AL 35806-3301

(205) 971-6224 (office)
(205) 971-6550 (facsimile)

stutzke@sed.redstone.army.mil

09 October 1997

Presented at the Twelfth International Forum on COCOMO and Software Cost Modeling (Los Angeles, 9-10 October 1997).

ABSTRACT

Ever increasing numbers of software development products are being built using Commercial Off The Shelf (COTS) components to reduce costs, risks and cycle time. Integrating COTS components into a product involves designing and writing "glue code", and then testing to ensure that the new component operates correctly and compatibly with other parts of the product (custom code and other COTS components). The glue code must often be revised and retested each time a vendor releases a new version of the COTS component. The project thus incurs rework costs, in addition to the original license costs.

This paper discusses the primary parameters that affect the costs associated with analyzing, modifying and retesting parts of the overall product each time a new version of a COTS component is released by a vendor. We describe a simple model for estimating COTS volatility costs. The model parameters include the total number of modules in the system, the number of expected releases, and the scope and complexity of the interactions affected by a new version of a component. (Our model does not address the increase of the modification costs with time, although we suggest two approaches for handling this.)

Although a validated and calibrated model still remains to be defined, the present work does identify guidelines for using COTS components that will reduce the costs associated with COTS volatility. These guidelines will help managers and engineers choose designs and COTS components in ways that reduce project cost and risk.
1.0 INTRODUCTION

Ever increasing numbers of software development products are being built using Commercial Off The Shelf (COTS) components. Project teams use COTS components to reduce costs, risks and cycle time, and sometimes also to improve product quality. COTS products offer known functionality for a stated license price.

The developer incurs additional costs when using COTS components as part of a product because the developer must integrate the components into the product's architecture. Such integration involves designing and writing "glue code", and then testing to ensure that the new component operates correctly and compatibly with other parts of the product (which may include custom code as well as other COTS components).

Using COTS components as part of a product also engenders certain potential risks. First, Garlan et al [Garlan, 1995] have described the danger of "architectural mismatch" between components, leading to time consuming and expensive modifications to achieve compatibility between the components of the system. Second, also noted by Garlan, the components may contain functions that are not needed or that duplicate functions in other components of the system, leading to "bloatware", software which consumes excessive amounts of computer processing cycles and/or memory. Third, the vendor of the COTS components defines the features (functions) and interfaces of the COTS product in response to competitive pressures and market acceptance. Of particular concern in this paper are those cases where the vendor changes features and/or interfaces. (In the worst case, the vendor may eliminate a function needed by the developer.) The developer becomes dependent on the vendor, who controls the component's requirements.

Another risk arises because the vendor may issue (potentially incompatible) releases of a COTS component during the course of a development project. If the developer must use this new version1 and if the new version is incompatible with the existing software product, then the developer must modify the glue code to restore compatibility of interfaces and functions. (The developer may have to revise other portions of the software product as well.) This rework can be expensive since a COTS component can have many interfaces, both to the custom application code and to other COTS components. (One example is a data base management system which has interfaces to application code and to the underlying operating system.) The number of such interfaces increases with the number of COTS components used and the particular product architecture used. The effort associated with updating each such interface depends on the "strength" of the coupling between the particular COTS component and the other components of the system. This strength depends on the architectures used for the components and for the

1 Sometimes the developer must deliver a product which is current with respect to the versions of the COTS components it contains, and so the team must incorporate the new releases at some time during a project, even if the old versions were operating compatibly.
product itself, the particular modifications made to the COTS component as part of the new
release, and the detailed way that the glue code is written for the component.

Every release of a new version of a COTS component triggers analysis of that component's
interfaces to other components of the system being developed, and may result in modification of
some or all of these interfaces. Each time the glue code is revised and retested, the project incurs
rework costs, in addition to the original license costs. The rework costs increase with the
duration of the project and the number of interfaces between the components.

Developers and their managers need a way to quantify the costs associated with COTS
volatility. A simple cost estimation model could support quantitative risk assessment, and could
also provide a cost value that could be added to the total project cost, thereby providing a more
accurate project cost estimate.

2.0 FORMULATING THE MODEL

The primary parameters that affect the costs associated with COTS volatility are:

- The number of components
- The "coupling" of the components (scope, difficulty)
- The costs of analyzing, modifying and retesting the system
- The expected release frequency
- The sequencing of releases relative to the project's activities.

The number of components, N, in the system includes custom and COTS components. The
coupling depends on the number of links, L, between a particular component, denoted by i, and
other components (custom or COTS). Coupling also depends on the "scope" of the interface.
This relates to directly observable characteristics such as the number of arguments passed in
procedure calls. It also relates to more subtle types of coupling such as assumptions about units
of measure, coordinate systems, and communications protocols. The costs of analyzing,
modifying and retesting arise as follows. First, every module linked to a given COTS component
needs to be analyzed each time that component is updated to determine if an incompatibility has
been created. Of the L, linked modules, some subset, M, must be modified. The costs of
analyzing, modifying and retesting the system can vary widely. (This is discussed more below.)
The expected release frequency depends on the vendor's marketing plan and ability to meet
planned release dates. For example, Microsoft alternates major and minor releases, issuing a new
version every 12 months, and tries to meet the planned release dates within a few days. (See
[Cusumano, 1995]). Obviously, more releases must be considered for longer project durations.
Lastly, the sequencing of releases relative to the project's activities may affect the cost. For
example, changes to a COTS component very early in a project have less cost impact than
changes made after the Code Complete milestone. [Boehm, 1989] illustrates that costs increase
by a factor of 100 for defects discovered during the operation of a deployed system compared to
defects discovered during the requirements analysis stage. Similar cost increases are plausible
for interfaces to COTS components.
2.1 Costs for One Interface

If we let E denote effort, and assume that every module linked to a specific COTS component requires the same effort for analysis (A), modification (M), and testing (T), then:

\[ E = E_A + E_M + E_T \]

where, assuming the same effort per module,

\[ E_A = L \times E_o \]
\[ E_M = M \times E_o \]
\[ E_T = E_o \times [M \times (M-1)/2 + M \times (N-M)] \]

where the last term is based on the model of Gerlich and Denskat [Gerlich, 1994]. The three constants, \( E_o \), \( E_m \) and \( E_o \), represent the average effort per module analyzed, modified, and tested. This equation has five parameters (excluding N).

2.2 Parameters of Interest

The estimated effort for every release of every COTS component could be summed to obtain the total estimated cost due to component volatility. If the total number of component releases is \( N_{\text{release}} \), then the model has \( N_{\text{release}} \times 5 \) parameters. This is too many for practical use so we need a way to aggregate the parameters.

There seem to be three parameters of interest. The first is the effort to analyze the links to the COTS component of interest. Analysis describes how much analysis and design effort we must do before modifying the modules. This effort also reflects our understanding of the modification and the system being modified. (If we understand the change, we can just go in and change the code. If we must decipher complex logic and/or design new algorithms, then more effort is required.) We propose a three-valued rating scheme for A:

- None (e.g., change error message, prompt, change, value of a constant),
- Easy (expect no serious problems in defining and making the modifications),
- Hard (involves complex functions and/or interfaces to other modules or systems, performance constraints or tradeoffs).

This parameter, \( A \), can possibly be related to COCOMO 2.0's Software Understanding (SU) cost driver. (See [Boehm, 1994].)

Second, the effort to modify the glue code depends on how many "items" must be added, changed or deleted. By "item" we mean lines of code, function points, or parameter values in a class template. We propose a three-valued rating scheme for the amount of modification, M:

- Few items (<= 5)
- Many items
- All items
where "All" includes a total rewrite or adding a totally new module. This parameter, M, can possibly be related to COCOMO 2.0's Adaptation and Assimilation (AA) and Adaptation Adjustment Factor (AAF). AAF may not be relevant, however, since we seldom have the source code for licensed COTS components and so their size in Source Lines Of Code is unknown.

Lastly, the number of modules affected by changes in a particular component are denoted by M (or possibly by L). One way we could possibly aggregate this measure is to define a scope parameter, S, also having three values:

- One - a single module
- Several - a few modules (less than 10)
- All - every module (includes most and many, i.e., > 10 modules)

We will use the S parameter later but chose to define it here.

Using these parameters requires us to define a number of coefficients for a cost model. If the effort depends on A and M, for example, there are 9 coefficients needed. We needed a way to aggregate the values into, say, 3 to 4 levels of difficulty, D. We can reduce the number of coefficients needed by constructing a 3x3 matrix having rows for Analysis and columns for Modification:

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Modify</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>VL</td>
</tr>
<tr>
<td>Easy</td>
<td>LO</td>
</tr>
<tr>
<td>Hard</td>
<td>ME</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As shown, we assign five-valued Likert scale to the cells, starting with Very Low (VL) in the upper left corner and proceeding to Very High (VH) in the lower right corner. This maps the nine value pairs to five values of difficulty, D. (If we choose to replace VL with HI and replace VH with HI, then only three values are produced.) This mapping is based on assumed "isoeffort" contours lying at 45 degree angles across cells of the matrix.

2.3 Initial Model

We are now able to formulate a simple model based on the following assumptions. First, all releases of a particular COTS component have a single characteristic difficulty, $D_i$, and affect the same number of links, $L_i$. This is reasonable since it is difficult to know if vendors employ some strategy such as Microsoft's where releases alternate in scope and impact. The easiest assumption is thus uniformity of release content. Second, the average efforts for analyzing, modifying and testing are the same for every release of a component. Third, the effort per component for testing ($E_T$) scales linearly with the number of links, and so $E_T = E_T \times L$. These assumptions mean that the effort for every release of a given COTS component requires the same effort, namely $E_i(D_i) \times L_i$, where $E_i(D_i)$ is the sum of $E_a$, $E_m$, and $E_t$. 

The total effort due to COTS volatility for all releases of all components is given by:

\[
E_{\text{Total}} = \sum_{i=1}^{N_{\text{comp}}} N_{R}(i) E_{i}(D_{i}) L_{i}
\]

EQ (1)

This model has three parameters \(N_{R}(i), E_{i}(D),\) and \(L_{i}\) per component or \(3* N_{\text{Release}}\). This is still too many parameters for a practical cost estimating relation.

2.4 Simplified Model

We would like to group the terms of the summation in Equation (1) by, say, difficulty but this is not possible since the \(L_{i}\) values are different for each \(i\). To obtain a simpler model, we introduce the scope parameter, \(S\), and some additional assumptions. First, all components having the same difficulty require the same effort. This means that \(E_{i}(D_{i})\) is replaced by \(E(D_{i})\) in the summation. Second, we will replace \(L_{i}\) by some function of \(S_{i}\), denoted by \(f(S_{i})\). This allows us to aggregate the releases based on their values of \(D\) and \(S\). These assumptions give:

\[
E_{\text{Total}} = \sum_{D=VL}^{\text{VH}} E(D) \sum_{S=\text{One}}^{\text{All}} N_{\text{REL}}(D, S) f(S)
\]

EQ (2)

This model has only 15 parameters, the set \(N_{\text{REL}}(D, S)\) where \(D = VL..\text{VH}\) and \(S = \text{One}..\text{All}\).

To make this model work, we must define suitable functions for \(E(D)\) and \(f(S)\).

2.5 The Function \(E(D)\)

\(E(D)\) represents the average effort to analyze, modify and retest each component linked to a COTS component, based on the difficulty \(D\). We hypothesize that this effort can be estimated similarly to the way that COCOMO handles adapted code: the amount of code times an adjustment factor. We assume that the estimator will specify a value for the average effort to write the glue code for a link in the planned development environment. This value could be expressed in person-hours, or could be estimated as some number of source lines times a production coefficient.

We adjust this average effort by multiplying it by a Difficulty Adjustment Factor, DAF, to account for the level of difficulty. The effort for Very Low (VL) difficulty should be zero so DAF(VL) = 0. Selby reported the increasing cost of modifying existing code [Selby, 1988]. Stutzke [Stutzke, 1996] extended Selby's model and proposed a model tied to COCOMO 2.0's SU and AA parameters. Stutzke's model estimated the cost for modifying 100% of the code in an existing module costs 1.58 times the original development cost. Assuming that the value of 1.6 corresponds to the Very High (VH) difficulty links, and equally spacing the intervening values gives the function DAF(D):
Note that for this scale, the center value, ME (Medium), is not equal to 1.0 and does not represent a Nominal (NM) value as is the convention for COCOMO's cost drivers.

2.6 The Function f(S)

The scope, S, reflects the number of modules linked to the COTS component of interest. We need a way to map the infinite range of values for Lᵢ into, say, three values of S. It seems plausible to assume that the maximum value of Lᵢ will not exceed 20 modules or so. (We are really assuming that 20 is a reasonable limit to the number of links to any module in a well-partitioned design.) We will just compute an average value of L for each S value and assume that the effort scales linearly with L. These assumptions give:

<table>
<thead>
<tr>
<th>S</th>
<th>Range</th>
<th>&lt;L&gt;</th>
<th>Rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>0-1</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>Several</td>
<td>2-9</td>
<td>5.5</td>
<td>6</td>
</tr>
<tr>
<td>All</td>
<td>10-20</td>
<td>15.0</td>
<td>15</td>
</tr>
</tbody>
</table>

3.0 IMPLEMENTATION DETAILS

The model needs to be validated before it can be used. Also, there are some possible refinements we could make. These topics are discussed in this section.

3.1 Validating and Calibrating the Model

We need to collect data on several modifications to validate the many assumptions of this model, and to calibrate the various coefficients. Ideally, we should collect data for several modifications triggered by multiple releases of some set of COTS components. For each modification, we should record the total effort to analyze, modify, and test the code, the number of components analyzed, Lᵢ, and the number of components modified, Mᵢ. For each COTS component we need to record the estimated values of the difficulty D and scope S. We must also record the total number of COTS components, N_{Comp}, and modules in the software product, N. (N includes those components counted in N_{Comp}).

This data will enable us to compute the average effort per modification, the average number of links analyzed, and the average number of links modified. It will also enable us to assess the validity of the assumption of equal effort per modification and equal number of links.
affected per modification. (The data will not allow us to determine if the analysis and the modification effort values for each link are approximately equal. Similarly, it provides no information on the total amount of glue code per link, and the amount of this code that is added, changed, and deleted.)

3.2 Using the Model

Assuming that the model can be validated and calibrated, the steps to use the model are as follows. First, for each COTS component, estimate the values for A and M and then determine the value of D using the 3x3 matrix. Then, estimate the value of S (or alternatively, L_s). Then, estimate the number of releases for that component expected during the duration of the project. (This is the development time, TDEV, estimated by COCOMO.) Second, for the project, estimate the effort needed to construct an average link, E_{avg}, considering the development environment, languages used, etc. Third, sum the values for the expected releases of individual COTS components grouped by D and S to get $N_{rel}(D,S)$. Fourth, compute the summation in Equation (2) to get the total effort.

3.3 Possible Refinements

Earlier we mentioned that the sequencing of releases relative to the project’s activities may affect the cost of the rework. For example, changes to a COTS component very early in a project have less cost impact than changes made after the Code Complete milestone. The reason is that there is less “product” to be modified early. Later, much design documentation and code exists and these must be modified. [Boehm, 1989, page 31] illustrates that costs increase by a factor of 100 for defects discovered during the operation of the deployed system compared to defects discovered during the requirements analysis stage. Similar cost increases are plausible for COTS components.

One possible model is to assume that the rework cost increases geometrically by the fixed amount $X$ for each release. If there are N releases, and the cost for the first modification is $E(0)$, then the total cost for N modifications is:

$$E_i = E(0) \times (X^N - 1)$$

Another possible approach is to base the rework costs on the number of modules completing detailed design and coding. This number increases rapidly from Product Design Review (PDR) until the Code Complete milestone, as illustrated in Figure 1, and reflects the amount of product that must be reworked. The abscissa represents the percentage of the total project duration. The milestone times shown are approximate. The ordinate shows the percentage of total modules coded. This model will give a constant rework cost after Code Complete. This is unrealistic since integration and testing will have to be redone if modules are changed during this period. Nevertheless, the simplicity of the linear model shown in Figure 1 may outweigh neglecting the increase in rework costs after Code Complete.
If we decide to include such effects in the model, this will cause disaggregation of the terms in Equation (2). The reason is that each release occurs at a different time (since the releases are not synchronized) and so the effort, E(D) must be adjusted for the effects of time. If this happens, there is no benefit to introducing f(S) since every release must be handled separately. This means that we would want to reintroduce L in place of f(S), thus causing the number of parameters to grow excessively large. It might be possible to construct functions representing continuous distributions for difficulty, scope, rework cost versus elapsed time, and either the number of changes received versus elapsed time or the time interval between the arrival of successive changes. These distributions could be integrated to give a result having only a few parameters. The estimator would only have to choose values for these parameters. (The estimator might also be allowed to choose the shape of the distributions.) This remains a possible area for future work.

Another possible approach is to define a "volatility parameter" for each COTS component. This would be similar to COCOMO 2.0's platform machine volatility (PVOL) or personnel continuity (PCON) cost drivers [Boehm 1994]. This parameter would replace the number of expected releases with a Lickert rating scale reflecting the average update rate. The Lickert rating would be mapped to an update rate. Multiplying this rate by the project's duration (TDEV) would give the expected number of releases, which would then be used in the model described in Section 2. This approach is less direct than just estimating the number of releases. It does, however, decouple the volatility of the COTS component from the project duration and so may be a more desirable formulation than just specifying the number of releases directly.
4.0 IMPLICATIONS FOR MANAGEMENT

Even though the model is crude and incomplete, it does provide insight into factors that affect the costs associated with COTS volatility. These insights lead to some useful guidelines for project managers:

- Choose COTS components that are stable
- Choose COTS components that are compatible
- Design the product to reduce coupling
- Shorten the development cycle and/or exposure time
- Commit to specific versions
- Handle modifications in batches

The following paragraphs explain each of these guidelines.

A stable COTS component is complete and has proven capability. Completeness means that the component has a full set of features and so is not likely to undergo extensive additions. Proven means that the component is mature, and has been used by many users in the field. This hopefully ensures that serious bugs have been eliminated. (Cusamano and Selby report, however, that the half-life of code for Microsoft products is only 18 months. [Cusamano, 1995]. This means that maturity is only apparent, not actual for these products. Caveat emptor!) This guideline decreases the number of expected releases, \( N_{R(i)} \), and so the total volatility cost.

Compatible COTS components are ones that are architecturally matched and so (one hopes) fewer modifications will be needed, and fewer bugs will be introduced due to incompatibilities. This decreases the effort needed to analyze, modify and test modules each time a component is released.

Design the product to reduce coupling in two ways. First, choose a "clean" partitioning of the architectural elements to reduce interfaces between lower level elements belonging to different higher level elements. This reduces the number of links, \( L_i \). Second, isolate volatile components via bindings. The same binding can be used by all modules needing to link to the component. This decreases the amount of unique glue code that must be written, reducing the effort needed to modify a link. It may also reduce the number of links, \( L_i \).

Shorten the development cycle and/or the "exposure time". This decreases the number of releases \( N_{R(i)} \) that must be handled. One way to do this is to incorporate no new releases after some project milestone, say, Code Complete or, even better, after Product Design Review (PDR). The next two paragraphs describe related activities.

Freeze the baseline by identifying a specific version of each COTS component to be delivered. This should NOT be stated as "The most current version available at the time of
Instead, the proposal or project plan should state that version X of COTS component Y will be delivered. (This is particularly important for firm fixed price contracts!)

Handle modifications in batches to reduce the effort to analyze, modify, and test. The concept here is to consolidate a set of releases using a “pathfinder team” to analyze, check and understand the changes, and to verify their compatibility (or, if incompatible, to identify ways to reestablish it). The pathfinder team works independently of the main team and defines procedures and tools to modify the links. Then the consolidated set of modifications is made to the actual system by the main team at a time carefully chosen to reduce the impact to the project. The pathfinder team does the analysis, and does some testing to validate the necessary modifications. The pathfinder team also provides special tools to reduce the effort needed to modify the actual links. The project invests in “pathfinding” for each set of releases to reduce the costs of analyzing, modifying and testing the individual links. (A different cost model is needed for this situation of course.)

A related idea is to reduce the number of releases to be handled by skipping releases. Unfortunately, this may not save as much effort as one might expect. You cannot usually jump from version 5 to version 7, skipping version 6 because the modification package for version 7 assumes that you are starting with version 6. You must apply both sets of modifications in turn. This approach may still, however, reduce the effort needed for analysis, modification and testing.
References


