A Software Product Line Life Cycle Cost Estimation Model

Barry Boehm, A. Winsor Brown, Ray Madachy, Ye Yang
Center for Software Engineering, University of Southern California
{boehm, awbrown, yangy}@sunset.usc.edu, madachy@usc.edu

Abstract

Most software product line cost estimation models are calibrated only to local product line data rather than to a broad range of product lines. They also underestimate the return on investment for product lines by focusing only on development vs. life-cycle savings, and by applying writing-for-reuse surcharges to the entire product rather that to the portions of the product being reused. This paper offers some insights based on the exploratory development and collaborative refinement of a software product line life cycle economics model, the Constructive Product Line Investment Model (COPLIMO) that addresses these shortfalls. COPLIMO consists of two components: a product line development cost model and an annualized post-development life cycle extension. It focuses on modeling the portions of the software that involve product-specific newly-built software, fully reused black-box product line components, and product line components that are reused with adaptation. This model is an extension built upon USC-CSE's well-calibrated, multi-parameter Constructive Cost Model (COCOMO) II, tailored down to cover the essentials of strategic software product line decision issues and available supporting data from industries.

1. Introduction

Most software product line cost estimation and return-on-investment models confine themselves to software development costs and savings. However, if life cycle costs are considered, product lines can have a considerably larger payoff, as there is a smaller code base to undergo corrective, adaptive, and perfective maintenance.

This paper presents an initial product line life cycle cost estimation model based on COCOMO II [1]. It illustrates the use of the model with a representative product line set of parameters based on experience in aircraft and spacecraft product line domains. We believe that these parameters are likely to be representative of other embedded software domains, and have prepared this initial model as a starting point for testing this hypothesis. We have also developed an Excel version of the model to facilitate experimentation.

Section 2 presents the elements of a COCOMO II-based product line investment model called COPLIMO. Section 3 presents the COPLIMO life cycle model; defines its input parameters, algorithms, and output estimates; and discusses an example of its use. Section 4 concludes the paper and discusses future work.

2. COCOMO II-Based Constructive Product Line Investment Model Elements

Following the COCOMO II family nomenclature, we have called this model the Constructive Product Line Investment Model, or COPLIMO. Software product line investment models need to address two main sources of cost investment or savings [2]:

The Relative Cost of Writing for Reuse (RCWR): the added cost of writing software to be most cost-effectively reused across a product line family of applications, relative to the cost of writing the software to be used in a single application.

The Relative Cost of Reuse (RCR): the cost of reusing the software in a new product line family application relative to developing newly-built software for the application.

2.1 Relative Cost of Writing for Reuse (RCWR) Estimation

COPLIMO models RCWR via one specialized COCOMO II multiplicative cost driver, Development for Reuse (RUSE), and constraints on two other COCOMO II cost drivers, Required Reliability (RELY) and Degree of Documentation (DOCU). The RUSE factor addresses the added costs of product line domain engineering, product line architecture developments, and generalizing the code to cover a range of applications. Table 1 shows its rating scale and corresponding effort multipliers (effort estimates in person-months can be translated into cost estimates via an organization-dependent cost-per-person-month factor). For example, Table 1 indicates that the effort to develop software reuse across a product line is 15% higher than the effort to reuse software within a single project, exclusive of additional documentation and reliability investments.

The productivity range of 1.24/0.95 = 1.31 for the RUSE factor is considerably lower than the RCWR factor reported in the literature [2]. This is partly because the multiple regression analysis used to determine this and other COCOMO II productivity ranges normalizes out the effects of other contributing variables. For software product line reuse, these are primarily the COCOMO II degree of Documentation (DOCU) and required software reliability (RELY) variables.

Table 1 RUSE Cost Driver Ratings and Multipliers

RUSE Descriptors		None	Across project	Across program	Across product line	Across multiple product lines
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	N/A	0.95	1.00	1.07	1.15	1.24

The COCOMO II DOCU factor is used to capture the added costs of documenting reusable software for family-wide use. The RELY factor is used to capture the added costs of verifying and validating that the reusable software will work successfully across the product line family. The COPLIMO RCWR model requires that the DOCU rating be at least Nominal for Nominal and High RUSE ratings, and at least High for Very High and Extra High RUSE ratings; and that the RELY rating be no more than one level below the RUSE rating. Tables 2 and 3 show the rating scales and effort multipliers for the DOCU and RELY factors. Investing in High or Very High DOCU levels for product line software may not be excessive, as it can reduce the relative cost of reuse via more understandable software.

Table 2 DOCU Cost Driver Ratings and Multipliers

DOCU Descriptors:	Many life cycle needs unco-vered	Some life cycle needs unco-vered.	Right-sized to life cycle needs	Exce-ssive for life cycle needs	Very excessive for life cycle needs	
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	0.81	0.91	1.00	1.11	1.23	N/A

Table 3 RELY Cost Driver Ratings and Multipliers

RELY Descriptors:	Slight incon-venie-nce	Low, easily recove-rable losses	Moderate, easily recover-able losses	High fina-ncial loss	Risk to hum-an life	
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	0.82	0.92	1.00	1.10	1.26	N/A

2.2 Relative Cost of Reuse (RCR)

The COPLIMO and COCOMO II RCR model has two main options. Each determines an equivalent amount of newly developed software to include in the total equivalent size of the software application.

The Reused, or Black Box (unmodified code) RCR model has only single Assessment and Assimilation (AA) factor. It accounts for the effort required to assess the candidate reusable components and choose the most appropriate one, plus the effort required to assimilate the component's code and documentation into the new application. Table 4 shows the rating scale for the AA factor and the corresponding RCR factor, in terms of the percentage of the reused component's size to be included in the application's total size. Thus, for example, a 10,000 SLOC component with an AA rating of 2 (Basic module search and documentation) would add 10,000*(2/100)=200 SLOC to the equivalent size of the application to be developed.

Table 4 Rating Scale for Assessment and Assimilation Increment (AA)

AA Increment	Level of AA Effort
0	None
2	Basic module search and documentation
4	Some module Test and Evaluation (T&E), documentation
6	Considerable module T&E, documentation
8	Extensive module T&E, documentation

The COPLIMO and COCOMO II Adapted, or White Box (modified code) RCR model includes the AA factor and several other factors within a nonlinear estimation model, Equation 1. This involves estimating the amount of software to be adapted and three degree-of-modification factors: the percentage of design modified (DM), the percentage of code modified (CM), and the percentage of integration effort required for integrating the adapted or reused software (IM). These three factors use the same linear model as used in COCOMO 81, but COCOMO II adds some nonlinear increments to the relation of Adapted KSLOC to Equivalent KSLOC to reflect the non-linear tendencies of the model reflected in data from [8] and elsewhere. These are explained next. Eqn. (1):

$$\text{EquivalentKSLOC} = \text{AdaptedKSLOC} \times \text{AAM}$$

$$\text{AAF} = (0.4 \times \text{DM}) + (0.3 \times \text{CM}) + (0.3 \times \text{IM})$$

$$\text{AAM} = \begin{cases} \dfrac{[\text{AA} + \text{AAF}(1 + (0.02 \times \text{SU} \times \text{UNFM}))]}{100}, \text{for AAF} \le 50 \\ \dfrac{[\text{AA} + \text{AAF} + (\text{SU} \times \text{UNFM})]}{100}, \text{for AAF} > 50 \end{cases}$$

The Software Understanding increment (SU) is obtained from Table 5. SU is expressed quantitatively as a percentage. If the software is rated very high on structure, applications clarity, and self-descriptiveness, the software understanding and interface-checking penalty is 10%. If the software is rated very low on these factors, the penalty is 50%. This is where the investments in more thorough documentation discussed under the DOCU effort multiplier can pay off via reduced SU penalties during reuse. SU is determined by taking the subjective average of the three categories.

Table 5 Rating Scale for Software Understanding Increment (SU)

	Very Low	Low	Nominal	High	Very High
Structure	Very low cohesion, high coupling, spaghetti code.	Moderately low cohesion, high coupling.	Reasonably well-structured; some weak areas.	High cohesion, low coupling.	Strong modularity, information hiding in data / control structures.
Application Clarity	No match between program and application world-views.	Some correlation between program and application.	Moderate correlation between program and application.	Good correlation between program and application.	Clear match between program and application world-views.
Self-Descriptiveness	Obscure code; documentation missing, obscure or obsolete	Some code commentary and headers; some useful documentation.	Moderate level of code commentary, headers, documentation.	Good code commentary and headers; useful documentation; some weak areas.	Self-descriptive code; documentation up-to-date, well-organized, with design rationale.
Increment to ESLOC	50	40	30	20	10

The amount of effort required to modify existing software is a function not only of the amount of modification (AAF) and understandability of the existing software (SU), but also of the programmers' relative unfamiliarity with the software (UNFM). The UNFM factor is applied multiplicatively to the software understanding effort increment. If the programmer works with the software every day, the 0.0 multiplier for UNFM will add no software understanding increment. If the programmer has never seen the software before, the 1.0 multiplier will add the full software understanding effort increment. The rating scale for UNFM is in Table 6.

Table 6 Rating Scale for Programmer Unfamiliarity (UNFM)

UNFM Increment	Level of Unfamiliarity
0.0	Completely familiar
0.2	Mostly familiar
0.4	Somewhat familiar
0.6	Considerably familiar
0.8	Mostly unfamiliar
1.0	Completely unfamiliar

Equation 1 is used to determine an equivalent number of new lines of code. The calculation of equivalent SLOC is based on the product size being adapted and a modifier that accounts for the effort involved in fitting adapted code into an existing product, called the Adaptation Adjustment Modifier (AAM).

AAM uses the factors discussed above, Software Understanding (SU), Programmer Unfamiliarity (UNFM), and Assessment and Assimilation (AA) with a factor called the Adaptation Adjustment Factor (AAF). AAF contains the quantities DM, CM, and IM where:

- DM (Percent Design Modified) is the percentage of the adapted software's design which is modified in order to adapt it to the new objectives and environment. (This is necessarily a subjective quantity.)
- CM (Percent Code Modified) is the percentage of the adapted software's code which is modified in order to adapt it to the new objectives and environment.
- IM (Percent of Integration Required for Adapted Software) is the percentage of effort required to integrate the adapted software into an overall product and to test the resulting product as compared to the normal amount of integration and test effort for software of comparable size.

If there is no DM or CM (the component is being used unmodified) then there is no need for SU. If the code is being modified then SU applies.

The range of AAM is shown in Figure 1. Under the worst case, it can take twice the effort to modify a reused module than developing it as new (the value of AAM can exceed 100). The best case follows a one for one correspondence between adapting an existing product and developing it from scratch.

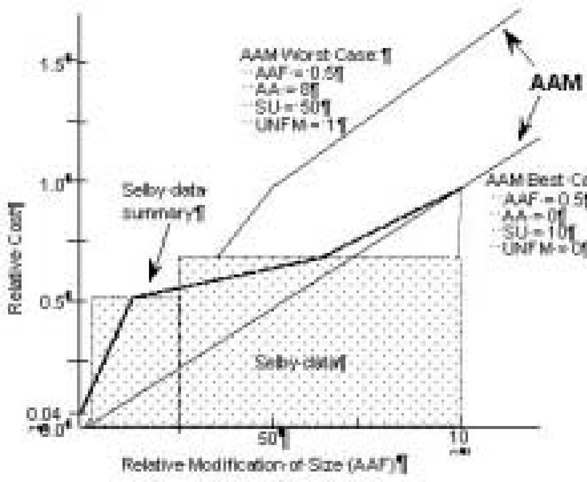


Figure 1 Nonlinear Reuse

2.3 Guidelines for Quantifying Adapted Software

This section provides guidelines to estimate adapted software factors for different categories of code using COCOMO II. The New category refers to software developed from scratch. Adapted code is pre-existing code that has some changes to it, while reused code has no changes to the pre-existing source (used as-is). COTS is off-the-shelf software that is generally treated the same as reused code when there are no changes to it. (Although the COCOMO II COCOTS model shows that there can be significant differences [9].) One difference is that there may be some new glue code associated with it that also needs to be counted (this may happen with reused software, but here the option of modifying the source code may make adapting the software more attractive).

Table 7 Adapted Software Parameter Constraints and Guidelines

Code Category	**Reuse Parameters**					
	DM	**CM**	**IM**	**AA**	**SU**	**UNFM**
New: all original software			N/A			
Adapted: changes to pre-existing software	0%~100%; normally>0%	0+%~100%; usually >DM and must be > 0%	0%~100+%; IM usually moderate and can be>100%	0%~8%	0%~50%	0~1
Reused: unchanged existing software	0%	0%	0%~100% rarely 0%, but could be very small	0%~8%	N/A	

Since there is no source modified in reused and COTS, DM=0, CM=0, and SU and UNFM don't apply. AA and IM can have non-zero values in this case. Reuse doesn't mean free integration and test. However in the reuse approach, with well-architected product-lines, the integration and test is minimal.

For adapted software, CM > 0, DM is usually > 0, and all other reuse factors can have non-zero values. IM is expected to be at least moderate for adapted software, but can be higher than 100% for adaptation into more complex applications. Table 7 shows the valid ranges of reuse factors with additional notes for the different categories.

3. The COPLIMO Product Line Life Cycle Cost Model

The COPLIMO product line life cycle cost model has two components: a product line development cost model and an annualized post-development extension. The model makes several simplifying assumptions about the uniformity and stability of the portions of the software that involve product-specific newly-built software, fully reused black-box product line components, and product line components that are reused with adaptation. These simplifications can be replaced by relatively straightforward extensions, which would eventually be needed for detailed product line planning and control usage. However, the simpler model will usually be sufficient for early exploratory tradeoff analysis of product line options.

3.1 Overview

The product line development cost (actually effort) estimation model assumes that each of the products in the product line contains software of roughly the same size. This is represented in the model as a PSIZE parameter measured in thousands of source lines of code (KSLOC), using the COCOMO II SLOC counting conventions [1]. It also assumes for simplicity that each of the products in the product line has roughly the same fractions of product-specific programmed software, product-line adapted software, and product-line reused software, as defined next. Again these simplifying assumptions can be relatively easily generalized.

The product-specific fraction of the software product's size, PFRAC, is the portion of the software that is unique to the particular product in the product line. This can cover special mission-specific or payload-specific features, or specialized market discriminators. For product lines of similar aircraft or spacecraft, a typical value of PFRAC is 0.4.

The adapted-software fraction of the software, AFRAC, is the portion of the product line software that must be modified to work well. This can cover adaptations for international localization, device-specific specializations, or specialized user interfaces. For product lines of similar aircraft or spacecraft, a typical value of AFRAC is 0.3.

The reused-software fraction of the software, RFRAC, is the portion of the product line software that can be reused as a black box without modification. This often covers the infrastructure software and core applications support functions such as mathematical or real-time control functions. For product lines of similar aircraft or spacecraft, a typical value of RFRAC is 0.3.

3.1.1 Relative Cost of Writing for Reuse (RCWR): Development

Again for simplicity, the model assumes that the COCOMO II reuse, cost driver, and scale factor parameters have equal values for each portion of new software, adapted software, and reused software in the product line. And again, these simplifying assumptions have straightforward generalizations.

The baseline for the relative cost of writing for reuse is the cost of not developing for reuse. With our simplifying assumptions of common sizes and cost driver parameters for each of the N similar products, the effort in person-month (PM) involved in not developing N similar products for and with reuse is

PMNR (N) = N · A· $(PSIZE)^B$ Π (EM), Eqn. (2)

where PSIZE is the general software product size defined above, A and B are the COCOMO II calibration coefficient and scale factor, and Π (EM) is the product of the effort multipliers for the COCOMO II cost drivers.

Including considerations of reuse, the effort for writing the first instance of the product line software for reuse is

PMR (1) = PMNR (1) * [PFRAC + RCWR*(AFRAC+RFRAC)], Eqn. (3)

Where RCWR is the product of the increase in effort multipliers resulting from the higher ratings in the COCOMO II RUSE, DOCU, and RELY cost drivers required for product line reuse.

As a simple example, suppose that each products size is 100 KSLOC, that the coefficient A as the standard COCOMO II. 2000 value of 2.94, and that for simplicity the scale factor B and the product of effort multipliers Π (EM) are both 1.0. Then the effort to develop a product without reusability or reuse is

PMNR (1) = 1· 2.94· (100) (1.0) = 294 PM

In comparison, suppose that the product line to be developed has the aircraft-spacecraft distributions of reuse categories: PFRAC=0.4, AFRAC=0.3, and RFRAC-0.3. If the product line has considerable variation, we may rate its develop-for-reuse factor RUSE as halfway between Very High and Extra High, with an effort multiplier of 1.195. In order to minimize the Software Understanding penalty, we may elect a Very High DOCU rating, with an effort multiplier of 1.23. To minimize rework, we may elect a Very High RELY rating with an effort multiplier of 1.26. The resulting RCWR factor is then (1.195) · (1.23) · (1.26) = 1.85, and the overall cost of the first product line instance is

PMR (1) = 294 [0.4 + 1.85(0.3+0.3)] = 444 PM

3.1.2 Relative Cost for Reuse: Development

To determine the relative cost or effort for reuse, we need to determine the equivalent sizes of the adapted and reused software.

The equations for equivalent size of reused and adapted software were provided in section 2.2 in terms of several additional parameters. For this example, we use the following parameter values:

Product-specific software (PFRAC):	0.4
Black-box plug-and-play reuse (RFRAC):	0.3
Reuse with modifications (AFRAC):	0.3
Assessment and assimilation factor (AA):	2
Software understanding increment (SU):	10
Unfamiliarity factor (UNFM):	0.5
% design modified (DM):	15%
% code modified (CM):	30%
% integration redone (IM):	40%

With respect to each 100 KSLOC product line reuse instance, the equivalent size, EKSLOC, caused by reuse works out to:

$$\text{EKSLOC}_P = 0.4 \times 100 KSLOC$$
$$= 40 KSLOC$$
$$\text{EKSLOC}_R = 0.3 \times 100 KSLOC \times \left[\frac{2}{10}\right] = 0.6 KSLOC$$

$$\text{EKSLOC}_A = 0.3 \times 100\ \text{KSLOC} \times$$
$$\left[\frac{2 + (0.4 \times 15 + 0.3 \times 30 + 0.3 \times 40) \times (1 + (0.02 \times 10 \times 0.5)}{100}\right]$$
$$= 30 KSLOC[2 + (27) \times (1.1)]/100 = 10.1 KSLOC$$
$$\text{EKSLOC} = EKSLOC_P + EKSLOC_R + EKSLOC_A$$
$$= (40 + 0.6 + 10.1) KSLOC$$
$$= 50.7 KSLOC$$

For each of the N-1 products benefiting from product line reuse, the resulting product development effort is:

2.94 (50.7) = 149PM

Table 8 and Figure 2 show the comparison between the development of N similar products without and with product line reuse. For this example, product line reuse almost breaks even at two products, and shows significant effort savings thereafter. The return on investment (ROI) is acceptably positive with a product line of N=3 similar products, and grows significantly at higher values of N.

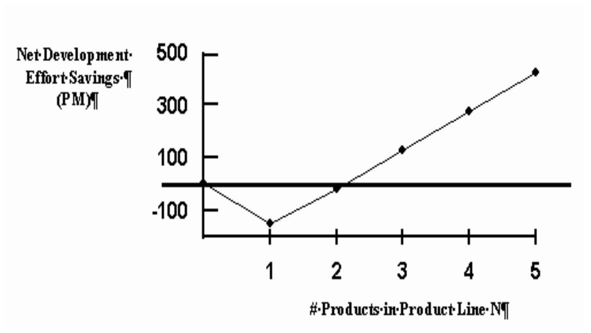


Figure 2 Net Development Effort Savings

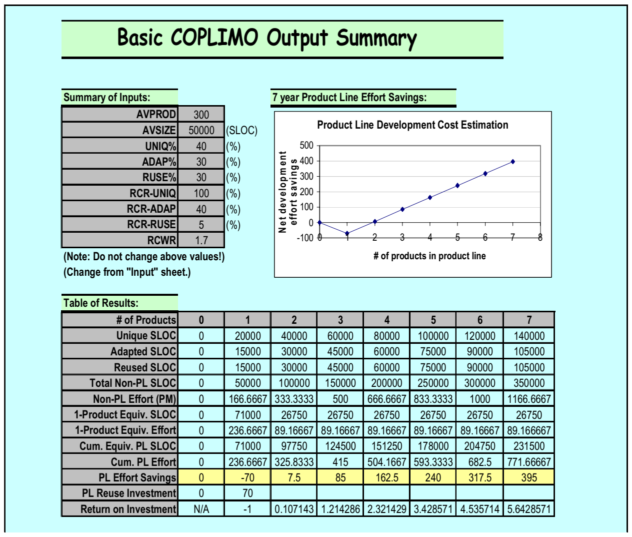
Table 8. Relative Development Effort Without and With Product Line Reuse

# Products N / Effort (PM)	1	2	3	4	5
No Reuse	294	588	882	1176	1470
Product Line	444	593	742	891	1040
Product Line Savings	-150	-5	140	285	430
ROI=PLS(N)/|PLS(1)	-1.0	-.03	+.93	+1.9	+2.9

Although this product line development reuse model is relatively simple with respect to the potential extensions that can be modeled with COCOMO II (variable product sizes, scale factors, effort multipliers, component sizes), its breakdown into fractions of new, adapted, and reused software provide finer-grain distinctions than do most software reuse models. Many such models overestimate a product line's relative cost of writing for reuse by applying the RCWR factor to the entire software product, when actually there is no need to apply it to the fraction of non-reused product specific software. In the example above, with 40% non-reused software, this reduced the estimated initial-project reuse investment from 294 (1.85) = 544PM to 294 [.4 + .6(1.85)] = 444PM. Thus, the effective RCWR for the project was 444/294 = 1.51 rather than 1.85. For this particular set of reuse parameters, this brought the product-line breakeven point closer to 2 projects rather than the usual 3 projects. As we will see next, life cycle considerations make product line reuse even more cost effective.

3.1.3 Example of COPLIMO Spreadsheet Model

Figure 3 shows an example of the COPLIMO spreadsheet model. It shows an example of sensitivity analysis: that if the RCWR factor is reduced from 1.85 to 1.7, the return on investment for 2 products goes from slightly negative in Figure 2 to slightly positive in Figure 3.



Basic COPLIMO Output Summary

Summary of Inputs:

AVPROD	300	
AVSIZE	50000	(SLOC)
UNIQ%	40	(%)
ADAP%	30	(%)
RUSE%	30	(%)
RCR-UNIQ	100	(%)
RCR-ADAP	40	(%)
RCR-RUSE	5	(%)
RCWR	1.7	

(Note: Do not change above values!)
(Change from "Input" sheet.)

7 year Product Line Effort Savings:

Product Line Development Cost Estimation
Net development effort savings
500
400
300
200
100
0
-100
0
1
2
3
4
5
6
7
8
# of products in product line

Table of Results:

# of Products	0	1	2	3	4	5	6	7
Unique SLOC	0	20000	40000	60000	80000	100000	120000	140000
Adapted SLOC	0	15000	30000	45000	60000	75000	90000	105000
Reused SLOC	0	15000	30000	45000	60000	75000	90000	105000
Total Non-PL SLOC	0	50000	100000	150000	200000	250000	300000	350000
Non-PL Effort (PM)	0	166.6667	333.3333	500	666.6667	833.3333	1000	1166.6667
1-Product Equiv. SLOC	0	71000	26750	26750	26750	26750	26750	26750
1-Product Equiv. Effort	0	236.6667	89.16667	89.16667	89.16667	89.16667	89.16667	89.166667
Cum. Equiv. PL SLOC	0	71000	97750	124500	151250	178000	204750	231500
Cum. PL Effort	0	236.6667	325.8333	415	504.1667	593.3333	682.5	771.66667
PL Effort Savings	0	-70	7.5	85	162.5	240	317.5	395
PL Reuse Investment	0	70						
Return on Investment	N/A	-1	0.107143	1.214286	2.321429	3.428571	4.535714	5.6428571

Figure 3. Output Interface of COPLIMO

As shown in Figure 3, the Basic COPLIMO Outputs screen has three displays:

- The table in the upper left is a summary of the Inputs.
- The table at the bottom is a summary of the Outputs whose derivation is described in Section 3.
- The graph in the upper right is a plot of Product Line Effort Savings as a function of the number of products K in the product line. Based on the decreased value of the RCWR parameter, the graph shows that starting with the second product in the product line, there is positive net savings.

3.2 The Basic Product Line Annualized Life Cycle Cost Model

The basic product line annualized life cycle model assumes that the fractions of new product-specific, adapted, and reused software stay relatively constant across the life cycle. It also assumes that their cost driver ratings*, reuse parameters, and annual change traffic

* Actually, the COCOMO II effort multipliers for maintaining high or very-high reliability are lower than their development counterparts by 12% and 16% but this factor is not included in the example for simplicity and conservatism.

(ACT) stay relatively constant across the life cycle as well. ACT is the relative fraction of a product's software that is modified per year.

The constant-ACT assumption implies that a single ACT factor applies to the annual degree of change of each product's reuse categories: new product-specific, adapted, and reused software. Again, this and the other simplifying assumptions have straightforward COCOMO II generalizations.

3.2.1 Life Cycle Effort Without Reuse

As with the development-phase comparisons with and without software product line reuse covered in Section 2.3.1, the life cycle effort comparison begins with the life cycle effort estimation of a set of N similar products with no reuse, across a post-development life cycle duration of L years. In general, for a constant set of factors ACT, SU, and UNFM across L years, the estimated life cycle effort PM(N,L)NR is obtained via the equivalent size of annually maintained software, AMSIZE, as the sum of the development with no reuse, PM(N)NR, and L times the annual post-development effort:

$$AMSIZE = PSIZE \cdot ACT(1 + \frac{SU}{100} \cdot UNFM)$$

$$PM(N,L)_{NR} = PM(N)_{NR} + L \cdot N[A \cdot (AMSIZE)^{B} \cdot \Pi(EM)]$$

From our previous example of develop of N similar products without reuse, we had PSIZE = 100 KSLOC, A = 2.94, B = 1.0, Π(EM) = 1.0, and PM(N)NR =N · 100 · 2.94 = 294 · N. For life cycle maintenance, we use the previous value of UNFM = 0.5, but increase the SU penalty to 20 because no extra effort was devoted to documentation. This makes the equivalent size of each product's annual maintenance increment

$$AMSIZE \quad = 100(.20)[1 + \frac{20}{100} \cdot 0.5] = 22\, EKSLOC \quad .$$

Then with a maintenance length L = 5 years, we have

$$PM(N,5) = 294 \cdot N + 5 \cdot N \cdot [2.94 \cdot (22)] = 617 \cdot N.$$

3.2.2 Life Cycle Effort With Reuse

The life cycle effort with reuse across the three categories of product-specific programmed software, adapted product line software, and reused product line software yields three categories of annual maintenance and AMSIZE:

$$AMSIZE_P = PSIZE \cdot PFRAC \cdot ACT \cdot (1 + \frac{SU}{100} \cdot UNFM)$$

$$AMSIZE_R = PSIZE \cdot RFRAC \cdot ACT \cdot (1 + \frac{SU}{100} \cdot UNFM)$$

$$AMSIZE_A = PSIZE \cdot AFRAC \cdot ACT \cdot (1 + \frac{SU}{100} \cdot UNFM) \cdot [1 + AAF(N-1)].$$

For a product line of N similar products, there are N instances of maintaining the AMSIZEP amounts of product-specific software. There is only one instance of maintaining the AMSIZER amount of fully-reused software, and one instance of maintaining the AMSIZEA amount of base adapted software and its N-1 variants, whose amount of specialized-adaptation software is calculated from its Adaptation Adjustment Factor AAF defined in Equation 1.

For the life cycle effort with reuse example, we use the same overall parameter values used for example without reuse in the previous section: PSIZE=100 KSLOC, A=2.94, B=1.0, Π(EM)=1.0, ACT=0.2, and UNFM=0.5. For the software understanding effort, we use 10 for the reused and adapted software developed with an extra documentation investment, and 20 for the product-specific software developed with nominal documentation. The resulting AMSIZE-quantities for annualized maintenance effort are:

$$AMSIZE_P = (100KSLOC) \cdot (0.4) \cdot (0.2) \cdot (1 + \frac{20}{100} \cdot 0.5) = 8.8KSLOC$$

$$AMSIZE_R = (100KSLOC) \cdot (0.3) \cdot (0.2) \cdot (1 + \frac{10}{100} \cdot 0.5) = 6.3KSLOC$$

$$AMSIZE_A = (100KSLOC) \cdot (0.3) \cdot (0.2) \cdot (1 + \frac{10}{100} \cdot 0.5) \cdot [1 + .27(N-1)]$$
$$= 6.3 + 1.7(N-1)KSLOC$$

Table 9 summarizes the components and total life cycle sizes, efforts for development with and without reuse across development and life cycle maintenance for a maintenance period of L=5 years, as a function of the number of similar products developed.

Table 9 Effects of Product Line Reuse on Life Cycle Effort

# Products in Product Line	1	2	3	4	5
AMSIZE-P (KSLOC)	8.1	16.2	24.2	32.3	40.4
AMSIZE-R (KSLOC)	6.1	6.1	6.1	6.1	6.1
AMSIZE-A (KSLOC)	6.1	7.7	9.3	11.0	12.6
Total Equiv. KSLOC	20.2	29.9	39.6	49.3	59.1
Effort (AM) (x2.94)	59.4	88.0	116.5	145.1	173.7
5-year Life Cycle PM	296.9	439.8	582.6	725.4	868.3
PM (N,5) R (+444)	740.9	883.7	1026.5	1169.4	1312.2
PM (N,5) NR	590.9	1181.9	1772.8	2363.8	2954.7
Product Line Savings(PM)	-149.9	298.2	746.3	1194.4	1642.5
ROI=PLS(N) / |PLS(1)|	-1.00	1.99	4.98	7.97	10.96
Devel. ROI (Table 8)	-1.00	-0.01	0.98	1.97	2.96

The two lines PM(N,5)R and PM (N,5) NR show the significant difference in 5-year life cycle cost growth enabled by the reduction in amount of software to maintain with product line reuse as compared to non-reuse. The comparative returns on investment between a life-cycle model and a development-only model shown in the bottom two lines show the much higher ROI obtained when considering life cycle effects. Figure 4 shows this graphically in terms of relative effort savings. Even after present-value discounting of future cash flows, the life cycle savings remain highly significant. Recent work by Cohen [10] shows similar life-cycle results.

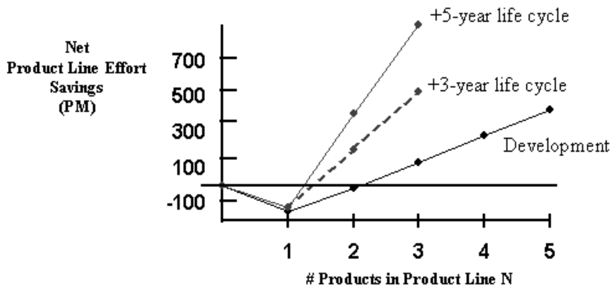


Figure 4 Net Life Cycle Effort Savings

3.2.3 Discussion of Results

Most quantitative models of software product line reuse significantly underestimate the potential savings and return on investment due to reuse [2, 3, 4, 7, and 11]. They generally overestimate the relative cost of writing for reuse (RCWR) by applying it to the entire product, rather than just to the parts that are likely to be reused. As we saw at the end of Section 3.1.2, selective development for reuse reduced the effective project-level RCWR from 1.85 to 1.51.

The models also generally underestimate reuse savings by addressing just development vs. life cycle costs. As seen in Table 9 and corroborated by the empirical data in [10], the fact that product line reuse involves considerably fewer lines of code to maintain has a powerful effect on lifecycle savings and ROI. Even with relatively conservative assumptions, consideration of life cycle savings shows a high ROI even for a product line of just two similar products.

This does not mean that just any attempt at product line reuse will generate large savings. A good deal of domain engineering needs to be done well to identify the portions of the product line most likely to be product-specific, fully reusable, or reusable with adaptation. And a good deal of product line architecting needs to be done well to effectively encapsulate the sources of product line variation. Particularly in immature or rapidly evolving domains, the costs and risks of domain engineering and product line architecting may be much higher, and the half-life of the asset investments much lower. Rapidly evolving domains also erode some of the assumptions in the COPLIMO model about stability of product-specific and reuse fractions.

Besides the technical issues, management and cultural challenges may inhibit successful software reuse. Attempting to develop and capitalize on product line assets across organizations disinclined to cooperate usually will not succeed. Cultural difficulties with “not invented here” attitudes and risk aversion must be successfully addressed. A critical success factor is an empowered product line manager with the ability both to make product line asset investments and to incentivize organizations to use the resulting assets. Some good treatments of these factors are in [3, 5, 6, and 11].

On the other hand, being too risk-averse in a dynamic competitive environment is not prudent. From a value-based software engineering standpoint [12, 13], there are other significant product line benefits besides life cycle cost savings that should be considered, such as rapid time to market and rapid adaptability to market changes. The COPLIMO model and its menu of extensions provide a strong framework for performing such broader-gauge cost-benefit extensions.

4. Conclusions and Future work

Most quantitative models of software product line reuse significantly underestimate the potential savings and return on investment due to reuse. Furthermore, those models also generally underestimate reuse savings by addressing just development vs. life cycle costs. However, the case is that product line reuse may involve considerably fewer lines of code to maintain has a powerful effect on lifecycle savings and ROI.

The proposed COPLIMO and its menu of extensions provide a strong framework for performing such broader-gauge cost-benefit extensions. It facilitates the determination of the effects of various product line domain factors on the resulting product line returns on investment. Particularly important factors are the fraction of product-specific software (PFRAC), adapted software (AFRAC), and black-box reused software (RFRAC). These factors are easy to estimate optimistically, and COPLIMO enables product-line investors both to perform sensitivity analysis on their values, and to recalibrate their return on investment once actual values of PFRAC, AFRAC, and RFRAC become available from actual projects.

Future work primarily involves experimental COPLIMO extensions to enable more sensitivity analysis of its current assumptions, such as equal size and return of product line products, constant annual values of reuse parameters, integration with other COCOMO II family models such as COCOTS; and experimentation with new sources of empirical product line life cycle data such as [10].

References

[1] B. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II*, Prentice Hall, 2000.
[2] J. Poulin, *Measuring Software Reuse: Principles, Practices, and Economics Models*, Addison-Wesley, 1997.
[3] D. Reifer, Practical Software Reuse, Wiley, 1997.
[4] W. Lim, Managing Software Reuse, Prentice-Hall, 1998.
[5] D. Weiss, C.T.R. Lai, Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley, 1999.
[6] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.
[7] J.E. Gaffney, R.D. Cruickhank, "A General Economics Model of Software Reuse", Proceedings, of the 14th International Conference on Software Engineering, NY: ACM, New York, 1992, pp 327-337.
[8] R. W. Selby, "Empirically Analyzing Software Reuse in a Production Environment", Software Reuse: Emerging Technology, W. Tracz (Ed.), IEEE Computer Society Press, 1988, pp. 176-189.
[9] C. Abts, B. W. Boehm, E. Bailey-Clark, "COCOTS: A Software COTS-based System (CBS) Cost Model", proceeding of the European Software Control and Metrics (ESCOM) Conference, London, 2001, pp. 1-8.
[10] S. Cohen, "Predicting When Product Line Investment Pays", Technical Note, CMU/SEI-2003-TN-017, July 2003.
[11] M. G. Jacobson, P. Jonsson, Software Reuse, Addison-Wesley, 1997.
[12] S. R. Faulk, R. R. Harmon, and D. M. Raffo, "Value-Based Software Engineering (VBSE): A Value-Driven Approach to Product-Line Engineering", Proceedings of the First International Conference on Software Product-Line Engineering, MA: Kluwer Academic Publishers, Boston, 2000, pp205-223.
[13] B. Boehm, "Value-Based Software Engineering", ACM Software Engineering Notes, Vol. 28, No. 2, 2003, pp1-12.