

Estimating the Effective Size of Auto-Generated Code in a Large Software Project

Pam McDonald
THAAD Project Office

Dan Strickland
Dynetics, INC.
daniel.strickland@dynetics.com

Charles Wildman
THAAD Project Office

**DISTRIBUTION A: Approved for public release;
distribution unlimited.**

Estimating the Effective Size of Auto-Generated Code in a Large Software Project

The movie *Gone With the Wind* grossed over 198.6 million dollars domestically at the box office. The film classic has since been eclipsed by the likes of *Titanic* (\$600.8 million), *Star Wars* (\$461 million), and *E.T. the Extra-Terrestrial* (\$495 million) in terms of sheer revenue. However, *Gone With the Wind* was released in 1939. Adjusting the gross for inflation to present-year dollars, *Gone With the Wind* would have made over 1.1 billion dollars. The true success of the film is hidden by an outside factor. Likewise, in software development, a size estimate that is artificially inflated or deflated due to automatic code generation can hide the true effort of a program. Given the ability of code generation tools and languages to seamlessly generate large amounts of software lines of code, parametric models based on SLOC as an input can produce inaccurate estimates of effort. This paper identifies the issues associated with auto-generated code and outlines a method used by a large subject project to address those issues.

In *Software Cost Estimation With COCOMO II*, Dr. Barry Boehm writes, “Code generated with source code generators is handled by counting separate operator directives as source lines of code. It is admittedly difficult to count “directives” in a highly visual programming system. As this approach becomes better understood, we hope to provide more specific counting rules” [BOE99]. Large auto-generated code counts can skew effort, productivity, and defect density estimates [MAT97, POU97, ZIE95]. Clearly, counting auto-generated code is difficult, dangerous, and not completely defined. This paper separates the language of the code into *generating* and *resultant* languages. The generating code automatically generates a much larger amount of the resultant code. Auto-generation of code allows software developers the ability to produce massive amounts of code using less effort. However, parametric estimation models that use code size as a primary input (COCOMO II, SEER-SEM, etc.) have no existing vehicle to handle **both** generating and resultant languages [BOE99, GAL00]. Using the generating code size as the solitary input runs the risk of underestimating the development effort. While the design and coding phases of development are clearly performed by the generating language, the effort of testing is potentially performed by the resultant language. Likewise, if the resultant code size is used for the model input, the result risks overestimating the true effort and subsequent metrics based on

size. The solution in this paper amalgamates the inputs of the generating and resultant estimates to form a synthetic estimate that more accurately reflects the development effort of auto-generated code.

Determining the nature of the testing environment is the first step in estimating the effort of auto-generated code. In some software programs, extensive testing on the resultant code is both unnecessary and impractical given time-to-market constraints. In this instance, test cases are run in respect to the generating code. For estimation purposes in this case, the generating code size is used and the resultant code size is unimportant (as testing activities take place solely on the generating code). Estimates in this type of scenario can exclude any other inputs from this paper. However, many large software projects require testing to take place on the resultant code as opposed to the generating code. Such was the case with the subject project addressed in this paper. In this case, the total effort must be divided into testing and non-testing portions to produce a *synthetic* estimate. The exact division used in the subject project is covered later in this paper.

The second step in estimating auto-generated code effort is capturing the existing inputs. Inputs may include generating language, resultant language, generating code size estimate, and resultant code size estimate. Most programs will not have estimates for all four inputs, but those that are available should be captured and noted. Examples of generating languages include PowerBuilder, IDL, and Visual Basic. Some examples of resultant languages are Ada95, C++, and FORTRAN. The generating and resultant size estimates can be in any unit transferable to Source Lines of Code (SLOC), but this paper assumes either SLOC or Function Points. Generating and resultant size estimates in SLOC are distinct as they are based on two different languages of differing complexities. A single Function Point estimate is applicable to both generating and resultant size estimates as it is language independent.

The third step in estimating the effort of auto-generated code is identifying the testing and non-testing portions of the software lifecycle phases and noting their impact to the overall development effort. The non-testing phases use the generating language for development while the testing phases use the resultant language. In the sample project, the default phases and effort percentages of a large project in COCOMO II were used to identify this division [BOE97, BOE99, SOF01]. It is pretty simple to tag the Requirements Analysis (6.5% of the effort), Product Design (15.9% effort), and Detailed Design (22.4% effort) phases as non-testing. Likewise, the Integration Testing (26.2% effort)

phase falls squarely in the testing effort. It is the Code and Unit Testing phase (29.0% effort) that is not as cut and dried. At least one portion, the coding, is non-testing in nature and performed using the generating language. Yet another portion, the unit testing, implies testing performed on the resultant language. Deciding what portion of the Code and Unit Testing effort is performed by which language involves going one level deeper into the listed activities of the phase.

In COCOMO II, each development phase is further subdivided into activities and their respective percentages of effort in that phase. The default activities are Requirements, Product Design, Programming, Test Plans, Verification and Validation (V & V), Project Office Support, Configuration Management/Quality Assurance (CM/QA), and Manuals. The percentage effort associated with these activities in the Code and Unit Testing Phase is listed in Table 1 below:

Activity	Effort Percentage
Requirements	4%
Product Design	8%
Programming	56.5%
Test Plans	5.5%
V & V	8.5%
Project Office	6%
CM/QA	6.5%
Manuals	5%

Table 1 – Activity Distribution of Effort During Code and Unit Testing on a Large Project

The Requirements, Product Design, and Programming activities are non-testing in nature, while the Test Plans and V & V activities are testing activities in the phase. The Project Office, CM/QA, and Manuals activities are neither completely testing nor non-testing activities. In this case, the percentage effort is equally split for testing and non-testing. The final percentages inside of the Code and Unit Testing phase are 77.25% non-testing activities and 22.75% testing activities. When these percentages are applied to

the 29% effort for the entire Code and Unit Testing phase, the result is 22.4% non-testing and 6.6% testing. The final percentages for each type of effort are listed in Table 2 below:

Phase	Non-Testing Effort	Testing Effort
Requirements Analysis	6.5%	0%
Product Design	15.9%	0%
Detailed Design	22.4%	0%
Code and Unit Testing	22.4%	6.6%
Integration Testing	0%	26.2%
Totals	67.2%	32.8%

Table 2 – Non-Testing and Testing Effort by Phase in a Large Project

The percentages in Table 2 can be used to develop a formula for Synthetic SLOC with 67.2% effort in the non-testing, generating language and 32.8% effort in the testing, resultant language. Here, Synthetic SLOC is defined as the normalized effort of generating and resultant languages in a single size estimate. In COCOMO II, Synthetic SLOC would be used as the size input. It is noted that this case used the weighting percentages associated with a large project (between 128 and 5,000 KSLOC). Different sizes produce different weightings and could be computed in the same manor as above, but a decent “rule of thumb” is 66.7% effort in generating language and 33.3% effort in resultant language.

The final step in estimating the effort of auto-generated code is to identify the formula to be used for Synthetic SLOC based on the previously gathered inputs. In many of the cases listed, the Programming Languages Table (PLT) developed by Capers Jones is used to convert SLOC estimates to Function Point (FP) equivalent estimates based on language and vice versa [JON96]. The following are the cases and formulas used to determine Synthetic SLOC:

Case 1: Generating Language Known, Resultant Language Known, Generating Estimate in SLOC Known

This case uses the PLT values for the respective languages with units in SLOC / FP. The formula is as follows:

$$\text{Synthetic SLOC} = (66.7\%) * (\text{Generating SLOC}) + (33.3\%) * (\text{Resultant Language PLT Value} / \text{Generating Language PLT Value}) * (\text{Generating SLOC})$$

Case 2: Generating Language Known, Resultant Language Known, Resultant Estimate in SLOC Known

This case uses the PLT values for the respective languages with units in SLOC / FP. The formula is as follows:

$$\text{Synthetic SLOC} = (66.7\%) * (\text{Generating Language PLT Value} / \text{Resultant Language PLT Value}) * (\text{Resultant SLOC}) + (33.3\%) * (\text{Resultant SLOC})$$

Case 3: Generating Language Known, Resultant Language Known, Estimate in Function Points Known

This case uses the PLT values for the respective languages with units in SLOC / FP. The formula is as follows:

$$\text{Synthetic SLOC} = (66.7\%) * (\text{Function Point Estimate} * \text{Generating Language PLT Value}) + (33.3\%) * (\text{Function Point Estimate} * \text{Resultant Language PLT Value})$$

Case 4: Generating Estimate in SLOC Known, Resultant Estimate in SLOC Known

This case does not use the PLT values and results could be verified by using another case listed above if the inputs are available. The formula is as follows:

$$\text{Synthetic SLOC} = (66.7\%) * (\text{Generating SLOC}) + (33.3\%) * (\text{Resultant SLOC})$$

The cases listed above don't consider the effect of adapted/reuse code as defined by COCOMO II. If pre-existing code is adapted for use in the new development, the normal rules for

counting adapted code should be used in conjunction with the rules for Synthetic SLOC listed above. Also, any new code that is not auto-generated should be added to the Synthetic SLOC and adapted code to estimate size. Adapted code is still subject to the percent modification described in the COCOMO II manual based on re-design, re-coding, and re-testing efforts. These percentages need to take into account the auto-generation of the new code. If the adapted code is in the generating language, more attention must be paid to re-design and re-coding effort. If the adapted code is in the resultant language, more effort should be placed in re-testing and integrating. In the subject project, the adapted code was in the resultant language and the re-test percentage was correctly adjusted. The adapted code was added to the new, non-generated SLOC estimate and the Synthetic SLOC estimate and used as the Equivalent Source Lines of Code (ESLOC) input for the cost model.

The COCOMO II maintenance model uses Total SLOC (TSLOC) of the delivered software as the primary size input. In the case of auto-generated code, the delivered size is at least as large as the resultant code size. If TSLOC from a project with auto-generated code is used in a COCOMO II maintenance model, the estimated maintenance effort could be artificially inflated. Annual Change Traffic Percentage is another input to the maintenance model. Changes and corrections that comprise Annual Change Traffic could be made to either the generating code or the resultant code. Changes performed in the generating language require regeneration and re-testing of resultant code. Changes performed in the resultant language can be lengthy and time-consuming considering the size of the code relative to the effort. Auto-generated code introduces new wrinkles to the existing maintenance formula. Further research is needed in this area.

In of the subject project, there were two separate, identified examples of auto-generated code. The first example contained three software items that used auto-generated code. The testing was to be performed on the resultant code. The generating language was identified as IDL and the resultant language was Ada95. The generating and resultant sizes were estimated by the contractor. The PLT value is irrelevant when there are two size estimates. Using Case 4 from above, the Synthetic SLOC formula becomes:

$$\text{Synthetic SLOC} = 66.7\% * (\text{Generating SLOC Estimate}) + 33.3\% * (\text{Resultant SLOC Estimate})$$

Synthetic SLOC was added to the Adapted SLOC estimate to form a new ESLOC estimate used in the COCOMO II model. The second example contained one software item using auto-generated code. Again, the testing was to be performed on the resultant code. The generating language was identified as an in-house Fourth Generation Language (4GL) and the resultant language was Java. The resultant size was estimated by the contractor, but no estimate was made on generating size. The PLT value for Default 4GL is 20 SLOC/FP and the PLT value for Java is 53 SLOC/FP. Using Case 2 from above, the Synthetic SLOC formula becomes:

$$\text{Synthetic SLOC} = (66.7\%) * (20 / 53) * (\text{Resultant SLOC}) + (33.3\%) * (\text{Resultant SLOC})$$

The development in this example was completely new code, with no adapted code. The Synthetic SLOC output was used as the size input in the COCOMO II model

This paper introduces the difficulties with estimating effort in a program that uses auto-generated code. Using the concepts of generating and resultant languages, this paper presents a method for developing an amalgamated size estimate based on the default COCOMO II weightings for testing and non-testing activities. Further research is needed in the area of maintenance and how Synthetic SLOC might apply to the maintenance model, but this model is in use in two areas in a large software project. As more auto-generated code is identified within the sample project, more areas will adopt this sizing method. The method described within serves as a useful extension for sizing in any parametric cost estimation model for projects using auto-generated code.

References

- [BOE81] Boehm, Barry, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ. 1981.
- [BOE97] Boehm, B., et. al., *COCOMO II.1999.0 Model Definition Manual*, ftp://ftp.usc.edu/pub/soft_engineering/COCOMOII/cocomo99.0/modelman.pdf, University of Southern California, Los Angeles, CA. 1997.
- [BOE99] Boehm, B., et. al., *Software Cost Estimation With COCOMO II*, Prentice Hall PTR, Upper Saddle River, NJ. 1999.
- [GAL00] Galorath Incorporated, *SEER-SEM User's Manual*, Galorath Incorporated, El Segundo, CA. 2000.
- [JON96] Jones, Capers, "Programming Languages Table," <http://www.spr.com/products/programming.htm>, Software Productivity Research, Inc. Burlington, MA. 1996.
- [MAT97] Mathis, Randy, "Metric-Based Scheduling and Management," *CrossTalk*, July, 1997.
- [POU97] Poulin, Jeffrey S., "Reuse Metrics Deserve a Warning Label: the Pitfalls of Measuring Software Reuse," *CrossTalk*, July, 1997.
- [SOF01] Softstar Systems, *Costar 6.03*, <http://www.softstarsystems.com>, Softstar Systems, Amherst, NH. 2001.
- [ZIE95] Ziegler, Stephen F., "Comparing Development Costs of C and Ada," <http://www.rational.com/products/whitepapers/337.jsp>, Rational Software, Cupertino, CA. 1995.