Measuring Metadata-based Aspect-oriented Code in Model-driven Engineering

Sagar Sunkle, Vinay Kulkarni, and Suman Roychoudhury
Tata Research Development and Design Center,
Tata Consultancy Services,
54B, Industrial Estate, Hadapsar
Pune, 411013 INDIA
Email: sagar.sunkle@tcs.com || vinay.vkulkarni@tcs.com || suman.roychoudhury@tcs.com

Abstract—Metrics measurement for cost estimation in model-driven engineering (MDE) is complex because of number of different artifacts that can potentially be generated. The complexity arises as auto-generated code, manually added code, and non-code artifacts must be sized separately for their contribution to overall effort. In this paper, we address measurement of a special kind of code artifacts called metadata-based aspect-oriented code. Our MDE toolset delivers large database-centric business-critical enterprise applications. We cater to special needs of enterprises by providing support for customization along three concerns, namely design strategies, architectural, and technology platforms \((d, a, t)\) in customer-specific applications. Code that is generated for these customizations is conditional in nature, in the sense that model-to-text transformation takes place differently based on choices across these concerns. In our recent efforts to apply Constructive Cost Model (COCOMO) II to our MDE practices, we discovered that while the measurement of the rest of code and non-code artifacts can be easily automated, product-line-like nature of code generation for specifics of \((d, a, t)\) requires special treatment. Our contribution is the use of feature models to capture variations in these dimensions and their mapping to code size estimates. Our initial implementation suggests that this approach scales well considering the size of our applications and takes a step forward in providing complete cost estimation for MDE applications using COCOMO II.

I. INTRODUCTION

Model-driven engineering (MDE) is presumed to be superior to code-centric development due to raised level of abstraction and automation and a slew of advantages that these bring to building software. From initial small beginnings \([1]\), our MDE approach evolved into one capable of developing large database-centric business-critical enterprise applications in which models are treated as primary artifacts over the entire software development life cycle (SDLC) \([2], [3]\). Over the years, we have delivered 60+ large business-critical applications worldwide on a variety of technology platforms \([4]\). Yet, as we approach new customers and domains, we are often faced with a question as to how can we prove that our flavor of MDE or MDE in general is more economically beneficial to them than the established code-centric development technologies. To prove this as well as to create an approach for estimating effort, duration, and cost for MDE-based projects, we are involved in identifying and measuring different kinds of artifacts generated over the entire SDLC in our MDE toolset. While we found that measurement of most categories of artifacts generated and used in MDE can be easily automated, measurement of code generated for a set of aspectual concerns is difficult to automate. When developing applications for various domains, we had discovered that for the same domain, applications differed in design strategies, architectural specifics, and technology platforms \([5]\). Application-specific choices of these concerns were found to be cross-cutting which led to scattering and tangling in code generators when realizing these concerns. We addressed this issue with building blocks abstraction that resulted in metadata-based aspectual code generation \([3]\). When measuring the code generated using this technique though, we faced a difficult situation. Since this code is application-specific and aspectual in nature, we would have to extend the code generators themselves to automate counting of source lines of code (SLOC) for the code of these concerns. At the same time, this code exhibited product-line-like nature \([6]\), which meant that a straight-forward counter statement insertion approach would not scale. We therefore had to follow a more structured approach that was scalable and exhaustive with respect to code generated for aspectual concerns.

In this paper, we show our approach of sizing metadata-based aspectual code in applications developed using our MDE toolset. Our specific contribution is the use of feature models to capture variations in design strategies, architectural specifics, and technology platforms, and mapping of features to code size values. Since code generated for specific choices of aspectual concerns is consistent across applications and across domains, we have to extend code generators only once for a set of choices. Once the size measurements are available and mapped to features in feature models representing the variations in the aspectual concerns, it is possible to directly arrive at total size of code that is generated for these concerns. This size is then input to the Constructive Cost Model (COCOMO) II equations for estimating effort, duration, and cost.

The rest of the paper is organized as follows. Section II motivates separate measurement of aspectual concerns. We then discuss various transformation languages used in our MDE toolset and then describe the metadata-based aspect-oriented code generation practices in detail. Section III describes how...
we automate the measurement of this kind of code using feature models. Section [V] discusses preliminary results and shows how the rest of code and non-code artifacts may be calculated. Section [V] concludes the paper.

II. BACKGROUND

A. Motivating Separate Measurement of Aspectual Code

To calculate effort, duration, and cost of application development, COCOMO II takes size of the application as input [7]. This size is calculated either directly in terms of SLOC or function points that are converted to SLOC. One obvious way to circumvent specialized counting of various MDE artifacts is to take the code-centric approach where SLOC is measured for the final source code of an application generated using MDE. We believe that such an approach falls short due to two reasons: 1) taking code-centric perspective disregards core advantages that MDE brings to application development such as increased level of abstraction and automation which affect development effort, and 2) it has been suggested that when measuring SLOC of code, distinction should be made between code that is manually written and auto-generated and that non-code artifacts should also be taken into consideration since their generation requires some effort. We address the first cause by mapping characteristics of our MDE toolset to COCOMO II scale factors and effort multipliers. Our underlying premise is that various general MDE characteristics such as increased abstraction and automation, as well as toolset-specific characteristics influence and alter effort and scaling in application development. A detailed explanation of our mapping approach is outside the scope of this paper.

To address the second cause, we identified categories of artifacts to be measured in an application generated using our MDE toolset. We found that various adjustments are suggested to physical SLOC of code that is auto-generated which takes less effort that manually writing code [5]. Similarly, adjustments have been suggested based on target programming language and counting of non-code artifacts like database scripts, configuration and deployment scripts, test cases, and so on, which also take effort to be created and must be accounted for [10].

With this line of thought, we concentrated a special category of code being generated from models in our MDE toolset. This code takes care of certain aspect-like functionality of enterprise applications by providing customization in select concerns. Unlike auto-generated skeletal code, which is created by default for every class in the application model, the aspectual code is generated only under special conditions and therefore differs from application to application. This means that for an accurate and comprehensive application-specific SLOC measurement, we have to provide a mechanism to automate the SLOC counting of such code. In the next section, we first discuss in brief various transformation languages used in our MDE toolset and then explain how aspectual code generation works in our MDE toolset. Later sections present our approach for measuring the aspectual code.

B. Model-aware Languages for Code Generation

To specify business logic of application, we use a high level model-aware language called Q++ [4] in our MDE toolset. Q++ treats UML class models as types. Its usage guarantees that business logic specifications are consistent with models. It is used in transforming models to a variety of SDLC artifacts such as code, tests, deployment descriptors, user documentation, and configuration scripts.

Code generators in our MDE toolset are written and customized with a language called OMGen [6]. OMGen is specially designed to aid traversal and un-parsing of the application model. It is possible in OMGen to navigate through the application user model, in terms of application metadata model. OMGen uses syntax that is more or less similar to mainstream languages like Java and C++. Functions can refer to external functions so that functions for a specific purpose are grouped in a single file.

For customizing certain concerns in MDE development, we use a combined aspect-oriented model-driven approach. In the next section, we explain in detail how OMGen is used in generating metadata-based aspect-oriented code. We also describe what these concerns are and the application-specific customizations that are usually required to be implemented.

C. Metadata-based Aspect-oriented Code Generation

Transformation languages in our MDE toolset described in previous section along with scalable infrastructure [4] collectivelly addressed functionality concerns of large business enterprise applications. However, when developing applications for banking, financial services, and insurance domains, we found that applications for the same domain differed in design strategies, architectural specific, and technology platforms [5]. We came across different design strategies such as audit, persistence, caching, and attribute value handling. We observed variations in architectural specifics such as client-server pattern of distributed architecture, synchronous-asynchronous remote method invocation and middleware variations such as OLTP monitor or EJB for synchronous invocation, and OLTP monitor or reliable queuing mechanism for asynchronous invocation. We found that technology platforms varied such as for example, developing applications for Unix using PowerBuilder, C++, Tuxedo, ODBC, Sybase, or for mainframes using JSP, C++, CICS, ProC, DB2 and for Windows using WinForms, C#, COM+, ODBC, SQLServer, etc.

To address the code scattering and tangling when supporting code generation and application development for these concerns, we developed aspect-oriented model-driven techniques which effectively separated out support and maintenance of these concerns and enabled us to provide manageable variations in them. The basic intuition about using aspects for concern-specific code generation is to capture recurring code patterns into models while solution architecture-specific code
is generated separately and weaved into the code generated from the models.

To achieve this, we utilized an abstraction called building blocks. A building block is a localized specification of a concern in terms of a concern-specific meta-model \[11\]. It can specify both data contribution (attributes) and method contribution (code fragments in method bodies). A hierarchy of building blocks is represented as a tree where each leaf building block specifies how to stamp out concern-specific model elements and each composite building block specifies how model elements constructed in member building blocks are woven together. Aspect-oriented model-driven code generation is realized by post-order traversal of the building block hierarchy in three steps, namely instantiation, transformation, and weaving. The instantiation step stamps out models and merges them. The transformation step transforms models into code snippets and generates weaving specifications. The weaving step composes the generated code snippets by processing weaving specifications.

The building block abstraction is implemented using metadata, particularly using model tags. A tag is essentially a \(<\text{name, value}>\) pair with its name identifying a building block. From the modeling perspective, a tag can provide a bridge from model elements to first-class modeling constructs such as objects, properties, and associations. Figure [1] shows how code generation by processing building blocks in OMGen is coupled with business logic specification, i.e., behavioral code (method bodies) written in model-aware Q++ language \[3\]. The dashed boundaries around tag definitions and business logic specification indicate that these are separated out and code generation due to various tags does not affect manually added method bodies in Q++.

We targeted the building blocks based code generation at handling application-specific variations in design strategies, architecture, and technology platform dimensions. For each of the design strategies (D), architectural specifics (A), and technology platforms (T), we could obtain a \(\langle d_1 \ldots n, a_1 \ldots n, t_1 \ldots n \rangle \) giving our code generators a product-line-like nature as depicted in Figure [2]. We thus came up with a code generator product line where a specific code generator could be specified in terms of several transformational units encoding model-to-text transformation rule specified using model templates. Configuring and extending a code generator entailed selecting from a set of variation points in these rules and adding new template(s) respectively.

III. MEASURING CODE FOR ASPECTUAL CONCERNS

Variations in each of the \(\langle d, a, t \rangle\) concerns are implemented using tags which are referred to in the OMGen scripts for code generation. For instance, depending on whether a class has been modeled to be audited by setting AuditRequired tag to Yes, bodies for the methods imparting audit functionality such as getimage, setPreImage, and setPostImage are generated. These methods enable storing history of changes made to the classes. Along with the methods for audit, a weaving specification is generated indicating how to weave audit functionality into a given method as shown in Listing [1]

```
Listing I: Audit strategy weaving specification

1. Bracket
2. Order::method_name()
3. Before getImage()
4. setPreImage() After getImage()
5. setPostImage()
```

It is possible that if both persistence and audit strategies are enabled for a class, then code may be generated for interaction of these concerns. This happens for instance, for the modify method generated for persistence strategy, which is generated along with create, delete, get, and exists methods. When a class is modeled as persistent and required to be audited, weaving specification for modify method will be generated and based on it, corresponding code. It is evident that SLOC of source statements for individual \(\langle d, a, t \rangle\) as well as their combination should be taken into consideration. In the following we propose feature model based approach for the same.
Asynchronous Windows Caching Persistence SLOC Count Auditing

(47)\texttt{AuditRequired}\) shown in Listing 1. The target is Java code for audit of application classes. The tag \texttt{AuditRequired}\) is what represents the building block for audit strategy. Source statement following every colon \(:\) is the code to be generated. The dollar sign \$\) substitutes the value of the property specified.

Listing 2 only shows one place (in a single .osc file, osc for OMGen scripts) in the code generator source. The lines shown in slanted and bold \textbf{font}\) in Listing 2 are the source statements generated for audit purposes based on various conditions. It can be observed that source code generated based on the inclusion of audit strategy is spread across a number of conditions evaluated against values of properties of a class, method, and attributes.

The tag \texttt{AuditRequired}\) is referred to in 77 different places in 18 .osc files for code generation in the current stable version of generic code generator. In this version, there are altogether 71 .osc files taking care of skeletal model-to-text transformation, business logic, GUI, and data layer code generation. Out of 18 .osc files that the tag \texttt{AuditRequired}\) is referred to in, it is used in conditionals in 19 places in 11 files. For a given model element, different conditions will lead to the generation of different number of SLOC. Automating the counting of these lines is difficult even with a parser-based line counter because it is difficult to specify specific constructs required to be taken into account.

To measure the SLOC for specific choice of \(\langle d, a, t \rangle\) therefore, we have to insert counter statements in the source of the code generator itself. As stated earlier, there are a number of variations of \(\langle d, a, t \rangle\), which means that random counting of SLOC pertaining to specific tags needs to be avoided and a structured approach is required. Furthermore, it is likely that combinations of design strategies such as audit and persistence will lead to different number of SLOC than when considered individually. Such cases require special treatment as well.

At the model level, the product-line-like nature of code generation is evident as shown previously in Figure 2. We therefore decided to exploit feature models of variations in \(\langle d, a, t \rangle\) concerns for the purpose of automating the counting of SLOC generated for them. Figure 3 shows the cardinality-based feature model [12] of variations in \(\langle d, a, t \rangle\). The SLOC attribute of the \(\langle d, a, t \rangle\) feature model indicates the SLOC count for each of the features which are added together for a given variant \(\langle d, a, t \rangle\) product line.

A. Measuring variations in \(\langle d, a, t \rangle\) with feature models

Listing 2 shows the implementation excerpt of audit strategy from Audit.osc file, which is dedicated to generation of audit variables. It is based on the weaving specification pattern shown in Listing 1. The target is Java code for audit of application classes. The tag \texttt{AuditRequired}\) is what represents the building block for audit strategy. Source statement following every colon \(:\) is the code to be generated. The dollar sign \$\) substitutes the value of the property specified.

Listing 2 only shows one place (in a single .osc file, osc for OMGen scripts) in the code generator source. The lines shown in slanted and bold \textbf{font}\) in Listing 2 are the source statements generated for audit purposes based on various conditions. It can be observed that source code generated based on the inclusion of audit strategy is spread across a number of conditions evaluated against values of properties of a class, method, and attributes.

The tag \texttt{AuditRequired}\) is referred to in 77 different places in 18 .osc files for code generation in the current stable version of generic code generator. In this version, there are altogether 71 .osc files taking care of skeletal model-to-text transformation, business logic, GUI, and data layer code generation. Out of 18 .osc files that the tag \texttt{AuditRequired}\) is referred to in, it is used in conditionals in 19 places in 11 files. For a given model element, different conditions will lead to the generation of different number of SLOC. Automating the counting of these lines is difficult even with a parser-based line counter because it is difficult to specify specific constructs required to be taken into account.

To measure the SLOC for specific choice of \(\langle d, a, t \rangle\) therefore, we have to insert counter statements in the source of the code generator itself. As stated earlier, there are a number of variations of \(\langle d, a, t \rangle\), which means that random counting of SLOC pertaining to specific tags needs to be avoided and a structured approach is required. Furthermore, it is likely that combinations of design strategies such as audit and persistence will lead to different number of SLOC than when considered individually. Such cases require special treatment as well.

At the model level, the product-line-like nature of code generation is evident as shown previously in Figure 2. We therefore decided to exploit feature models of variations in \(\langle d, a, t \rangle\) concerns for the purpose of automating the counting of SLOC generated for them. Figure 3 shows the cardinality-based feature model [12] of variations in \(\langle d, a, t \rangle\). The SLOC attribute of the \(\langle d, a, t \rangle\) feature model indicates the SLOC count for each of the features which are added together for a given variant \(\langle d, a, t \rangle\) product line.

\footnote{A valid variant is a set of features selected from the feature model based on relations and constraints [13].}
We have already described audit and persistence design strategies and counting of SLOC when these strategies are applied. Apart from persistence, there are other commonly implemented object-relational mapping strategies such as caching and attribute value handling. Caching strategy is used to specify caching-based CRUD methods. Similarly, attribute value handling is required for such purposes as special treatment of null value of an attribute.

Architectural specifics such as message handling strategies can be further decomposed into method invocation kinds such as synchronous and asynchronous method calls and middleware can be decomposed into Microsoft Message Queue Server and Enterprise Java Beans. We have shown only the most common design strategies, architectural specifics, and technology platforms in Figure 3. Accordingly, technology platforms are decomposed to main flavors such as Unix, mainframes, and Windows.

The target of the code generation is determined by technology platform. We therefore take into consideration contribution of the target programming language. Since we plan to employ COCOMO II for effort, duration, and cost estimation, we need to convert physical SLOC counts per language and technologies used under technology platform to logical equivalents. Table I shows how physical SLOC is converted to logical SLOC.

In the next section, we propose an automation approach for counting \( (d, a, t) \) related code based on the feature-based decomposition of \( (d, a, t) \) product line shown in Figure 3 and technology platform-specific SLOC conversion as shown in Table I.

### B. Automating SLOC for variations in \( (d, a, t) \)

The counting of \( (d, a, t) \) variations begins by identifying design strategies, architectural specifics, and technology platform in the concerned product. By directing the measurement procedure with the aid of a feature-based decomposition, we ensure that we are covering various details of \( (d, a, t) \) as they are implemented in a specific code generator. A counter data structure is created for each leaf feature in the feature model of \( (d, a, t) \) product line. In the simplest case, it is \( \{\text{feature\_name, class\_name, SLOC\_count}\} \). Counter statements are added in the source of the code generator whenever a specific metadata tag emits code related to the concern that the tag represents. The counter is incremented by \( n \) for \( n \) source statements added. SLOC counts for individual classes are added together. The sum value is taken to be the SLOC attribute for the given feature. This process is carried out step-by-step in the following manner:

1) A feature model of \( (d, a, t) \) is created for the code generator under consideration. Personnel who have been part of the code generator development team and who are acquainted with the domain come up with the feature model. Note that knowledge of both the customer requirements related to \( (d, a, t) \) and the source of the code generator are required.

2) SLOC counts for each feature are computed by running the extended code generator against the model and meta-model for which it was used.

3) In case of code generated for two or more features together as shown in Listing 3 a simple algorithm is used to adjust the contribution of individual features and their combination. Based on the knowledge of domain and customer requirement and actual code generator source, a counter data structure is created for the combination of features. For instance, to adjust SLOC contribution by audit and persistence, a counter data structure is created for audit\_persistence. SLOC counts for audit and persistence are generated first considering all the places where these are used to generate code. SLOC count for audit\_persistence is then computed by searching over the code generator source for places where customer requirements led to combined implementation of audit and persistence strategies. This count is subtracted from each of the audit and persistence counts to obtain SLOC counts for audit and persistence when they are used individually and not in combination. This process is repeated for all combinations of features that are actually used in the code generator source. With this algorithm, it is ensured that SLOC contribution of each feature is not over-computed.

4) For the given variant, i.e., specific choices of features in the \( (d, a, t) \) product line, excluding \( (t) \), SLOC counts of member features are added together.

5) Based on \( (t) \) choices, language adjustment factor is used to obtain adjusted logical SLOC count using size conversion estimates from Table I.

This 5-step procedure is applied to obtain SLOC contribution of metadata-based aspect-oriented code in a particular project created using our MDE toolset. With this process we obtain the complete contribution of \( (d, a, t) \) choices in a project in a detailed manner with contribution of each feature known separately.

### IV. Discussion

In the following, we first discuss preliminary results and challenges. We then present how metrics for metadata-based

<table>
<thead>
<tr>
<th>Technology Platform Language</th>
<th>To obtain Logical SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Third Generation - C, Cobol</td>
<td>25%</td>
</tr>
<tr>
<td>Fourth Generation - SQL, Perl, Oracle</td>
<td>40%</td>
</tr>
<tr>
<td>Object-oriented - C++, Java, Python</td>
<td>30%</td>
</tr>
</tbody>
</table>

TABLE I: Size estimate conversion based on technology platform [10]
aspect-oriented code can be integrated with the rest of the metrics for the entire MDE toolset.

A. Preliminary Results

We used our extensible and configurable code generator mechanism \([3], [6]\) for creating the feature model of \(d, a, t\) concerns applicable to a set of existing products. We used the same mechanism also for the derivation of valid variants from this feature model (specifically variants that are already deployed as well as ones which are being developed). Table II shows SLOC counts for audit, persistence, and audit_persistence concerns irrespective of their interaction with other features and then adjust (subtract) the code contributed by interaction. We do this instead of looking for combined implementation from the beginning, because we found that simultaneously looking for places where a feature is implemented individually and distinguishing it from whether it is used in combination is confusing and error-prone process. Instead with the help of developers of code generators, development documentation, and other sources of information such as commit history, it was possible to identify code generated from combination of features more accurately. We also found that even when a product implements two features from the \(d, a, t\) product line, it is quite possible that no code is generated for their combination as found for the banking products. On the other hand, when the interaction of two features indeed generates code that would not be present, if any one of them was not implemented, is adjusted based on number the SLOC of code generated by their interaction as shown for the test suite generator product.

Since this approach is automated, once the complete feature model is available and code generator source has been extended, our metric measurement procedure can be used as is for counting SLOC of \(d, a, t\) concerns irrespective of the domain or overall size of an application.

The proposed SLOC counting procedure is generic enough to be applicable to code generators from different versions of our toolset and for different products. Yet, we expect a number of challenges for a full-fledged implementation as enumerated below:

1) The code generators for \(d, a, t\) concerns generate not only source files such as Java/C++ based on technology platform, i.e., \((t)\) choice, but may also contribute to a lot of non-code artifacts like data definition language and other database scripts, testcases, and deployment description files etc. We need to have a mechanism in place to count SLOC for non-code artifacts generated by \(d, a, t\) concerns as well.

2) The counter statements have a crosscutting nature with respect to the rest of code generator code. We do not have a mechanism to asectually address the counter statements themselves. In the cases where code genera-

---

### Listing 3: Code Generation for Audit and Persistence together

```java
1 IF(isBulkMthd == 1)
2 {
3   ...
4   ELIF (isBaseMostPersistentClass(cls) == 1)
5       IF(methodname != "mdelete" || hasReferredClsInHier == 1)
6       {
7           :	${cls.Name} inst${parentCls.Name} = new ${cls.Name}();
8         IF (parentCls.AuditRequired == "y")
9             {
10                :	${cls.Name} inst${parentCls.Name}ForImage ;
11           }
12        }
13       }
14 ELIF((methodname == "mdelete") && (loopCounter == 0))
15     {
16         :	${cls.Name}inst${cls.Name}=new ${cls.Name}();
17     }
18   }
19 ...
20 }
```

---

### Table II: SLOC of asceptual features for different products

<table>
<thead>
<tr>
<th>Product</th>
<th>Feature x</th>
<th># classes that require feature x</th>
<th># classes in the Product</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payments &amp; Funds Transfer Product</td>
<td>Audit</td>
<td>457</td>
<td>1892</td>
<td>8254</td>
</tr>
<tr>
<td></td>
<td>Persistence</td>
<td>457</td>
<td>172</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Audit_Persistence</td>
<td>454</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Core Banking Solution</td>
<td>Audit</td>
<td>464</td>
<td>22938</td>
<td>444</td>
</tr>
<tr>
<td></td>
<td>Persistence</td>
<td>464</td>
<td>1827</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Audit_Persistence</td>
<td>456</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Test Suite Generator</td>
<td>Persistence</td>
<td>89</td>
<td>175</td>
<td>adj. 3372</td>
</tr>
<tr>
<td></td>
<td>Audit_Persistence</td>
<td>20</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

---

TABLE II: SLOC of asceptual features for different products
tor source is not available (but binary may be available), it can prove problematic to use the proposed procedure.

Despite of these challenges, we assert that the proposed procedure can be used to calculate SLOC contribution of \( (d, a, t) \) concerns in a structured manner so that effort, duration, and cost calculations are exhaustive in taking into consideration all aspects of MDE-based development.

B. Estimating cost of other MDE artifacts

Our MDE toolset uses four different kinds of artifacts, namely auto-generated code (skeletal class code, queries, GUI code), manually added code (code written in Q++ for method bodies, services, etc.), several non-code artifacts such as tests, deployment descriptors, user documentation, and configuration scripts and finally, metadata-based aspect-oriented code for \( (d, a, t) \) concerns. Starting with models, measurement approach similar to [15] which provides a mechanism to convert model element counts to SLOC estimates, can be used to obtain SLOC of auto-generated code. Manually added code (specifically method bodies) in Q++ as well as non-code artifacts can be calculated at source level using appropriate SLOC counters. Note that non-code artifacts mentioned here are separate from non-code artifacts generated by \( (d, a, t) \) concerns.

For both auto-generated code and non-code artifacts, size estimated may need to be adjusted. McDonald et al. observe that auto-generated code needs to be measured properly lest its measurement is inflated [8]. Lum et al. argue similarly stating that auto-generated code is not free and takes some effort and therefore needs to be considered along with the manually added code [10]. They suggest that since productivity level of developing auto-generated code differs from other code, it must be converted so that it becomes comparable to SLOC of non-auto-generated code. For object-oriented languages, they state that number of auto-generated SLOC must be multiplied by 0.09 and for fourth generation languages, by 0.06. We intend to incorporate these adjustments in counting the complete SLOC of application developed using our MDE toolset. Since we are already counting behavioral code (method bodies written in Q++) and aspectual concern-specific code separately, we only include SLOC of structural elements when employing conversion of model element counts in class models to SLOC estimates for the auto-generated code.

To measure influences of organization and toolset-specific practices on effort, duration, and cost, we plan to map various characteristics of our MDE-toolset and the nature of MDE practices to various scale factors and effort multipliers in COCOMO II. For this work, we intend to use data collection and analysis methods such as interviews and questionnaires, obtain and map responses to COCOMO II rating levels and provide an interpretation within the context of our MDE toolset for the value of each scale factor and effort multiplier in COCOMO II.

V. FURTHER WORK AND CONCLUSION

Further work in measuring metadata-based aspect-oriented code essentially includes extension of our approach to include non-code artifacts and integration of our approach with size measurements of all MDE artifacts. Finally, we intend to bring together size metrics with mapping of MDE characteristics to COCOMO II cost drivers, which gives us all the constituents for calculating effort, duration, and cost of applications developed using our MDE toolset.

In this paper, we addressed the SLOC counting of code generated with metadata-based aspect-oriented code generated in our MDE toolset. Based on the product-line-like nature of aspectual concerns, we came up with a feature model based approach for counting SLOC of these concerns. Initial results suggest that this approach is both scalable and exhaustive.

REFERENCES