Abstract

Software engineering researchers have attempted to improve the software development process for over two decades. A primary thrust in this process lies in the arena of measurement. “You can’t control what you can’t measure” [5]. This research applies software metric techniques to the development of multimedia products. Problem areas such as education, instruction, training, and information systems can all benefit from more controlled approaches to development with multimedia tools. As an example, we focus on one multimedia language for creating multimedia products: Macromedia’s Authorware. This paper describes the measurement of various distinguishing properties of this language, together with an evaluation of the measurement process. The evaluation gives insight into the next step in establishing the goal of control, through measurement, of the multimedia software development process.

Keywords: Metrics, measurement, Authorware, multimedia software development.

1 Introduction

Software maintenance, a stage in the software development lifecycle, requires up to 60% of the total software budget [4]. Over the years, many techniques for reducing the effort and cost involved in this stage of a product’s life cycle have been implemented and practiced. One such technique is the use of software metrics. Several types of software metrics exist that measure different properties of a product. This research focuses on software metrics for code complexity.

Software metrics exist for the procedural and object-oriented paradigms. These established metrics have demonstrated the ability to indicate error prone source code in a given module[12][9]. Previous metric studies have provided the community with “Thresholds” for the various metric values. These thresholds provide the programmer with a indication that a module may be error prone [10]. As fourth generation language (4GL) techniques have been applied to task-specific purposes, however, new problem-oriented languages have arisen. Such languages are aimed at developers who are not experts in programming or software engineering, and this class of languages does not belong to either the procedural or the object-oriented paradigm. At present, there are no known software metrics available for this class of languages.

The goal of software metrics is to provide the programmers, developers and maintainers with predictions of problematic software components [10]. In this paper, we describe the creation and validation of a suite of software metrics for problem-oriented languages aimed at multimedia programming tasks. Specifically, these metrics have been applied to software projects developed using Authorware, a Macromedia product. The approach to metrics described in this paper is easily extended to other multimedia languages, and suggests possibilities for developing metrics for problem-oriented languages focused on other problem areas. Problem-oriented languages contain some of the same “structures” as procedural and object-oriented languages, therefore, an investigation into those previously “proven” metrics serves as a starting point for the metric definition for Authorware.

2 Background

2.1 Multimedia Languages

Multimedia languages are a new class of languages that have arisen in the past decade. The term “multimedia” refers to “a presentation or display that involves more than one method or medium of presentation” [13]. Such media may include audio, video, still images, and animations that accompany the standard text display. Therefore, a multimedia application is one that uses and includes more than one of these media in a cohesive
manner. A multimedia language “is a set of software tools for creating multimedia applications” [13]. All multimedia languages present the developer with a set of software tools to aid in the development process. Although they all aid in manipulating similar presentation media, the functionality of these software tools may vary greatly from language to language. As of yet, no standard development environment exists.

Software development for a multimedia language typically takes place in an interactive development environment (IDE) that hides from the developer the underlying low level programming required to handle multimedia objects. To be concrete, our research has focused on one multimedia language: Macromedia’s Authorware. Authorware provides an IDE where software development takes place on a graphic flow line, as shown in Figure 1. The IDE presents the developer with thirteen different types of development icons. These icons are divided into three classes: icons that manipulate multimedia objects, icons that manipulate the flow of control within the application, and one icon that allows scripting.

Creating software in Authorware requires the developer to choose which icon is needed, drag the icon from the palette, and place it on the flow line. Once placed, the icon’s properties can be edited to customize its settings. Flow of control icons allow the developer to add branches to the flow line and affect the path(s) taken by the end user. For sophisticated applications, the flow line can be partitioned into “modules” that can call each other and that can be nested. Flow of control and scripting are described in problem-oriented terms and controlled by direct manipulation, so that the developer does not need any formal programming experience.

Figure 1: Macromedia Authorware’s IDE
2.2 Maintenance

Software maintenance is that phase of the development life cycle devoted to the elimination of post-delivery errors and the enhancement or extension of the product to cover new or changing requirements. The amount of effort that software companies spend on maintenance is large: typically 40-60%, and sometimes as much as 90%, of the total life cycle budget for a product [6].

As a result, to significantly reduce the cost of a software project, reducing the effort devoted to maintenance is a necessity. A large portion of maintenance effort is devoted to the elimination of post-delivery errors [8], and the earlier in the development process such errors are detected and corrected, the less costly they are to fix. As a result, identifying potential “problem areas” in source code during development for the purposes of reducing the number of defects that survive product delivery is one method of controlling maintenance cost. Identifying “problem areas” also helps to reduce the cost of enhancement and extension activities during maintenance, since tricky source code is also likely to be more difficult to modify and more expensive to retest. Software metrics are one tool for identifying problem areas in source code or design, so that the goal of reduced maintenance cost can be achieved [12][11].

2.3 Software Metrics

Software metrics measure code complexity based on a set of properties of a development language. Such measures provide a relative, quantitative evaluation of source code for this development language [4]. Research has shown that a strong correlation exists between the problem areas pointed out by software metrics and the main areas where maintenance efforts have been concentrated. Several different types of software metrics exist for both the procedural and object-oriented paradigms.

The three standard categories of software complexity metrics for procedural languages are: code metrics, structure metrics, and hybrid metrics. Code metrics measure properties of the source code by tallying its syntactic entities to give quantitative results. Some typical examples include: lines of code [4], Halstead’s software science indicators [7], and McCabe’s cyclomatic complexity [14].

Unlike code metrics, structure metrics attempt to measure the interconnectivity between the logical parts of the product being analyzed. Examples include: Henry and Kafura’s information flow metric [8], McClure’s invocation complexity metric [12], and Belady’s cluster metric [1].

Hybrid metrics combine code and structure metrics to capture some information about the syntactic structure of the source code in addition to capturing overall information on the product. Examples include: Woodfield’s view complexity [Woodfield] and the weighted form of Henry and Kafura’s information flow metric [8].

Unfortunately, object-oriented languages contain many properties that do not exist in procedural languages, and as a result, procedural metrics do not effectively capture all the sources of complexity in OO programs. As a result, a new collection of software metrics have been developed for gauging complexity in OO software. Such metrics include: weighted methods per class, depth of the inheritance tree, number of children, coupling between classes, response for a class, and lack of cohesion in methods [3][12].

Just as metrics for procedural languages do not effectively capture the properties of object-oriented programs, neither procedural nor OO metrics capture the properties of multimedia languages. New metrics that measure properties ranging from the number of icons on the flow line to the maximum depth of a module are needed.

3 Metrics for Multimedia Languages

Developing a new suite of metrics for a language is a very difficult and time-consuming effort. The first step in the development is to interview the developers of software product in that new language, in this case Authorware.

All six software metrics were derived from a meeting attended by experienced Authorware programmers. Since these four individuals have a combined total of eight years of experience among attended by experienced Authorware programmers. [BOO96].

This need for a new suite of software metrics for Authorware and other multimedia languages has given birth to the suite of six new software metrics. These six metrics derive themselves from the Lines of Code metric, Halstead’s Software Science Indicators, Henry and Kafura’s Information Flow metric, to a lesser extent the object-oriented metric Depth of Inheritance Tree, and McCabe’s Cyclomatic Complexity. All of these metrics were introduced in Chapter One. However, the basis for all six software metrics came from a meeting attended by experienced Authorware programmers. Since these four individuals have a combined total of eight years of experience among them, they are considered experts in Authorware [2].

The discussion concentrated on two main questions:

What properties of Authorware can potentially be measured?
What makes some Authorware source code better than others?

Lists of these properties were gathered and the question of whether those properties could be quantitatively measured then eliminated some of the properties.

A range of values was developed for these software metrics. The range of acceptable vs. non-acceptable values for these software metrics are based on years of experience. The second question was then explored. The belief that the Authorware properties being measured affect source code complexity
leads to the belief that these new software metrics are good indicators of error prone Authorware source code.

The suite of metrics for multimedia languages consists of six new software metrics. Four of the six are adaptations of existing software metrics, reinterpreted in a new context. The new metrics are: Number of Icons, Audio/Video Icons, Variable Passing, Depth of Module, Flow of Control, and Variables on Display Icons.

### 3.1 Number of Icons (NOI)

The Number of Icons metric (NOI) is based on the traditional lines of code metric. Lines of code measures how many source lines of code are in each function or module in the program being analyzed. As the number of lines increases, so does the likelihood that the particular function or module contains an error [4]. A threshold can be used to flag units deemed overly complex by this metric.

NOI measures the number of icons in a module. As the number of icons increases, so does the complexity of the code. Keeping the number of icons low reduces code complexity and improves readability. Table 1 gives a representative threshold for flagging modules with too many icons. The thresholds in Table 1 were determined by developer experience and assessed by the Authorware experts, but they are by no means meant to convey an exact threshold for all developers. These numbers may vary from company to company or development approach to development approach, and require fine tuning by each organization.

### 3.2 Audio/Video Icons (AVI1 and AVI2)

In Authorware, the use of audio and video icons present an opportunity for reusability. Such icons can be placed in libraries so that the actual icons placed on the flow line are links to the original in the library. This allows audio and video clips to be replaced or modified in the library, affecting all linked occurrences. Further, from the library, the developer can locate all references to a given clip without having to scour the code.

The pair of Audio/Video Icon metrics are based on Halstead’s software science indicators, which count the number of unique operators and the total number of occurrences of operators. For multimedia languages, the counts represent the total number of occurrences of linked audio and video icons in the source code (AVI1) and the number of unique audio and video icons in the libraries (AVI2). Threshold values for AVI1 and AVI2 appear in Table 1.

### 3.3 Variable Passing (VP)

The Variable Passing metric originated from Henry and Kafura’s information flow metric [8], which measures the complexity of source code as the amount of information flowing into and out of procedures or functions. It is defined as:

\[ C_p = (\text{fan-in} \times \text{fan-out})^2 \times \text{LOC} \]

Where
- \( C_p \) = the complexity measure for procedure \( p \).
- \( \text{fan-in} \) = the number of local flows into \( p \) plus the number of global data structures from which \( p \) retrieves information.
- \( \text{fan-out} \) = the number of local flows out of \( p \) plus the number of global data structures that \( p \) updates.
- \( \text{LOC} \) = then number of lines of code in \( p \).

The information flow metric was developed for languages that support user-defined procedures and functions. Authorware does not support this functionality, but Authorware does have a “module” construct that allows a single flow line to be “called” from one or more points in other flow lines. As a result, the information flow metric can be extrapolated to Authorware modules. The number of variables being passed into or out of a module is counted and used in this modified definition:

\[ C_m = (\text{fan-in} \times \text{fan-out}) \]

Where
- \( C_m \) = the complexity measure for module \( m \).
- \( \text{fan-in} \) = the number of local flows into \( m \) from which the module retrieves information.
- \( \text{fan-out} \) = the number of local flows out of \( m \) that the module updates.

As the value of \( C_m \) becomes larger, the complexity of the module increases. A threshold value for VP appears in Table 1.

### 3.4 Depth of Module (DOM)

The Depth of Module metric measures how deep the levels of nesting are for the current module. This metric is based on the object-oriented Depth of Inheritance Tree metric that reports the maximum depth of a given class in the class hierarchy [3]. In Authorware, map and framework icons provide the ability for one flow line to “call” other modules. Thus, every time a map or a framework icon is added to the flow line, another level of nesting is added to the source code. The belief is that as the DOM becomes large, the source code becomes more complex. Table 1 indicates threshold values for the DOM metric.
3.5 Flow of Control (FOC)

Authorware provides the ability to unconditionally branch from one point to another in an application’s flow line through a navigation icon and through a *Goto* function. Counting the number of uses of these two features gives an indication of additional complexity in the application’s flow of control.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Acceptable</th>
<th>Too Great/ Raise Warning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOI</td>
<td>&lt; 500</td>
<td>≥ 500</td>
</tr>
<tr>
<td>AVI1</td>
<td>&lt; 300</td>
<td>≥ 300</td>
</tr>
<tr>
<td>AVI2</td>
<td>&lt; 300</td>
<td>≥ 300</td>
</tr>
<tr>
<td>VP</td>
<td>&lt; 10</td>
<td>≥ 10</td>
</tr>
<tr>
<td>DOM</td>
<td>&lt; 6</td>
<td>≥ 6</td>
</tr>
<tr>
<td>FOC</td>
<td>&lt; 30</td>
<td>≥ 30</td>
</tr>
<tr>
<td>VDI1</td>
<td>&lt; 20</td>
<td>≥ 20</td>
</tr>
<tr>
<td>VDI2</td>
<td>&lt; 20</td>
<td>≥ 20</td>
</tr>
</tbody>
</table>

Table 1: Warning Thresholds for the Multimedia Metrics Suite

3.6 Variables on Display Icons (VDI1 and VDI2)

Display icons allow the content of variables to be displayed on the screen during the application’s execution. These icons allow simple equations to be embedded in the visual display as well. All dynamic output to the screen in Authorware applications, other than predefined graphics, text, audio, or video, is handled through display icons.

Display icons must be used with care, since it is difficult to determine whether or not variables are embedded in the display during development. In particular when a variable is initialized as a null string and is embedded in the display, nothing will appear in its place. An uninitialized variable, on the other hand, will appear as a zero. In addition, the developer can choose to display either the variable value at the current point in the application’s execution, or to display a continuously updated representation of the variable’s value. The Authorware experts considered variables on display icons to be likely sources of some kinds of errors.

The two pieces of data measured for this pair of metrics are the total number of variables in display icons (VDI1) and the total number of display icons with variables on them (VDI2). Table 1 shows acceptable threshold values for these two metrics.

4 Evaluating the Metrics

4.1 Data Collection

To evaluate the six multimedia language metrics, data were collected on two commercial software products created using Authorware, Product “A” and Product “B”. These products were both created by a small, privately owned, commercial software company that develops multimedia applications for education, training, and information systems that are targeted at a diverse clientele. Both products are interactive training applications that incorporate audio, video, text, and graphics; both products were considered to be small products by their developers.

Even though these products are small, this effort should be considered an initial step into the investigation of other Authorware products. If the metrics used in this study appear applicable to other products, then this initial suite may be a starting point for metrics for other problem oriented languages.

Data were collected one product at a time. Product A consisted of nine modules and Product B consisted of
four modules. All metrics from Section 3 were collected on a per module basis. Because Authorware’s binary file format is not human-readable, all measures were taken manually for this evaluation, although it is possible to create measurement software to analyze stored Authorware code directly.

The NOI metric measures the total number of icons for an entire module. The Authorware IDE provides a File Setup command that returns information on the current module, including the total number of icons in the module.

Collecting data for the AVI metric is a process that gathers two pieces of information at the same time. The first piece of information sought (AVI1) is the number of audio/video icons in the source code that are links to a library. To find this information a depth first search of the source code was performed, counting all audio and video icons linked to a library in the process. The second piece of information sought (AVI2) is the number of audio/video icons that exist in the libraries. This number may be obtained by inspecting the libraries themselves.

The information that the VP metric requires is easily obtainable, although all of the information is not available until every module for a product is inspected. There are two ways of calling another module, using the Jumpfile function and using the Jumpfilereturn function. While viewing the source code for the current module in the Authorware IDE, the references to these two built-in functions can be viewed using the Show Functions command on the Data menu. Using this command, all calls out of the module were inspected to calculate the fan-out for the module. Calculating the fan-in requires an inspection of the fan-outs of all the other modules in the product.

In order to compute the DOM metric, the level of the deepest branch of the module is recorded. Authorware source code can be represented by a binary tree, so computing the DOM metric is as simple as finding the longest branch in the tree.

The fifth metric, FOC, requires a count of the number of references to the Goto function and a count of the number of uses of the navigation icon. Gathering the information on the Goto function can be done using the Show Functions command on the Data menu. Gathering the number of uses of the navigation icon required a depth first search of the module during which a running total of the number of navigation icons found in the source code was accumulated.

The last measurement is for the VDI metric. There are two pieces of information to gather here: first, the number of variables that are on display icons (VDI1), and second, the number of display icons that have variables on them (VDI2). To find this data, a depth first search for all display icons was performed and every display icon was inspected to tally both measures.

All six metrics were collected for both Products A and B, along with the number of errors discovered in each module during development. Table 2 summarizes this data.

4.2 Data Analysis

In analyzing the data collected from Products A and B, the collected data were compared against the threshold values shown in Table 1. These results were then subjectively validated by interviewing the application programmers responsible for these two products.

Table 1 shows threshold values for the metrics presented in Section 3. A measured value that exceeds the threshold indicates a greater likelihood that the given module contains errors. These values may need fine tuning by each organization that employs them. Threshold values were suggested by the developers of the Authorware products and should not be assumed to be exact values. All metric values are dependent on factors such as the language, development environment, and product type.

After the data for Products A and B were collected, a comparison against the threshold values in Table 1 was conducted and the results are shown in Table 2. The data cells that are highlighted with a gray background in Table 2 exceed the threshold value for the corresponding metric. As noted in Table 2, module A-6, A-9, and B-4 all exceed the threshold for the NOI metric. For AVI1 and AVI2, B-4 was the only module exceeding the suggested threshold. B-1 and B-4 both exceeded the threshold for the VP metric. The DOM metric flagged modules A-6, A-8, A-9, B-2, B-3, and B-4 as having exceeded the threshold value. For the FOC metric, modules A-8, A-9, B-2, B-3, and B-4 all exceeded the threshold. Both A-6 and A-9 exceeded the VDI1 threshold, and A-9 also exceeded the VDI2 threshold. Modules exceeding the thresholds were then presented to the product developers for review.
5 Conclusions

In this paper, a suite of new software metrics for multimedia languages was presented. Each metric in this suite measures different properties of source code in such a language. Four of these metrics are adaptations of proven software metrics and the other two are completely new.

The new software metrics were applied in the context of a specific multimedia language, Authorware. They were computed on two commercially produced software products and data were collected. The software metrics were designed to indicate source code that has a greater likelihood of containing errors before the product under development is delivered to the client. Analyzing the collected metric data from the two products studied revealed that a number of modules in both products exceeded suggested thresholds for some of the metrics, and that these modules were ones that contained greater numbers of errors. These results were shared with the application programmers who developed the two products, and the programmers provided subjective validation that the metrics were indeed indicating modules that were likely to contain greater numbers of errors. Together, these two indicators suggest that this research is on the right track toward measuring properties of Authorware source code that indicate error proneness.

The strengths and weaknesses of this research can be summarized as follows:

Strengths:

1. A suite of software metrics were proposed for multimedia languages and applied to specific products written in Authorware.
2. The suite of software metrics was evaluated. Even though the sample products were relatively small, the suite of metrics was effective in indicating modules that were more likely to contain errors. This establishes an effective basis for a more complete set of multimedia metrics.

Weaknesses:

1. Additional Authorware products need to be evaluated to fine tune the suite of proposed metrics.
2. Another multimedia language and associated products must be evaluated to establish and further validate the suite of metrics.
3. While the subjective validation provided by this study has given promising results, larger products are also needed to perform statistical validation of regression models.

This work also opens several avenues for future research in the area of software metrics. Within the context of multimedia languages, addressing the weaknesses delineated above is a prime goal. In particular, applying these metrics to larger products, and to products developed using other multimedia languages, are important steps in refining this metric suite.

Further, the programming paradigm taxonomy developed by Bodnar [2] suggests that similar metrics can be designed for problem-oriented languages in other domains. Future research in metrics for other classes of problem-oriented languages shows promise.

Finally, no suite of metrics will ever be accepted by industry until an automated tool is provided to gather the metric values and the necessary data to perform a valida-
tion. Developing a automated method of collecting metric values for this suite is important, both for promoting industrial use of such metrics and for pursuing the weaknesses mentioned above as part of future research.

In the end, however, it is clear that graphically oriented languages designed for a non-programming audience are still amenable to metric analysis. Metrics for such languages can identify potentially troublesome spots in source code, and hold the promise for allowing one to reduce maintenance costs by focusing resources on the portions of the code most likely to contain errors.

References