Scade Interpreter for Measuring Static and Dynamic Software Metrics

Kilyén Attila Örs, Technical University of Cluj-Napoca, Faculty of Automation and Computer Science, Romania, kilyen.attila.ors@gmail.com
Dr. Barabas Laszlo, evoline srl, Cluj Napoca, Romania, barabas.laszlo@evoline.ro

Abstract—In this paper we present an interpreter framework designed for measuring static and dynamic characteristics of a Scade model. While some of the software metrics have become industrial standards in software development and for popular languages there is a variety of software measurement tools, for Scade there are no such tools. Our main achievement is that we developed an interpreter for metrics, and we provide easy access for all the information gained from these measurements. We also implemented some of the canonical software metrics like Cyclomatic complexity and Halstead’s Software Science.

I. INTRODUCTION

Scade is a model based visual programming language designed for developing embedded, safety critical software [1]. It is applicable on fields where reliability is paramount and no failures and crashes are allowed, such as aerospace, rail transportation and energetics. Scade represents a bridge between Control Engineering and Software Engineering because it provides a common, rigorous graphical and textual language for both communities [1]. It was based on the theory of synchronous languages for real-time applications.

Its synchronous model is very similar to the zero-delay model of circuits. In this model the timing characteristics of the parts (gates or transistors) are neglected and it is assumed that each part of the circuit computes results instantaneously. The interaction with time is based on a discrete time model [2]. In Scade we have only the main loop as follows: read the inputs - run the model - write the outputs. A Scade model is made of so called operators. These operators exhibit function-like behavior and may optionally have states.

Running the Scade model requires the translation of the model into C/C++ program. This program wraps the code corresponding to the Scade model into infinite loop, feeds the model with input data and sinks the data produced by the model. An important advantage of Scade is the built-in model-based safety analysis which helps the developers to integrate safety analysis techniques in everyday development ([3]).

Software measurement is essential to good software engineering. Most of the software engineers use different metrics to measure the quality of design, testability, and complexity of the code. ([4] , [5]). Some of these metrics are good quality indicators, that may predict the fault rate per module.

For example, metrics like Weighted Methods per Class and Coupling Between Object Classes designed by Chidamber and Kemerer [6] are proved to give an accurate prediction for the Mozilla Firefox modules ([7]). Almost every global company has its own methods and practices to measure software. Some of them, like IBM’s Rational Dashboard, built their own tool, others integrate them to their development environment, like Microsoft integrated to Visual Studio (Visual Studio Code Metrics PowerTool).

One of the most common software metrics is cyclomatic complexity described by Thomas J. McCabe in 1976. His aim was to develop a "mathematical technique that will provide a quantitative basis for modularization and allow us to identify software modules that will be difficult to test or maintain”([8]).

His approach is based on an abstraction called control flow graph. This is a directed graph with unique exit and entry nodes. Each node is associated with a part from the program where no decision was made, i.e. the program flow is sequential ([8]). The cyclomatic complexity of the program is calculated using Berge’s formula [9], based on number of linearly independent circuits of the graph.

Another canonical metric group is called Halstead’s Software Science [10] (HSS). HSS defines notions like the Volume of the program, Effort and Difficulty, based on the unique and the total number of operators and operands.

The metrics mentioned above are static metrics because they measure the source code of a program characteristic without executing it. In contrast to static analysis, dynamic analysis is performed by executing the program measuring the program characteristics run-time. An example of dynamic metrics is the code coverage[11]. This metric measures that in how many of the source code are executed for a specific input data set, usually this is expressed in percentage. An extension of this idea is the branch coverage which measures whether each logical condition in the code has evaluated both true and false [12]. Another extension is the state coverage which measures how many states were evaluated from all the states defined in the state machines of a model.

Our interpreter currently is capable to measure the cyclomatic complexity, the Volume of the program, Effort and Difficulty described by HSS, and the tree types of code coverage mentioned before. It is also able to compare different versions of a project, detect added (or changed) parts of the model, and measure these parts with metrics defined by Halstead.

The paper is structured as follows: In section two we present the proposed measurement framework and some of the Scade specific details, in section three is described an example of implemented metric, is section four some experimental result are presented, and in section five we highlighted possible direction of future works.
II. MEASUREMENT FRAMEWORK

Since the goal of the project was to implement a measurement tool, the program architecture differs significantly from classical interpreters. It has five high level components: the parser, the interpreter, interface for static metrics, interface for dynamic metrics and the metrics implementation. This five components are organized in three layers. The parser and the interpreter are the lowest layer; they have direct access to the all the information provided by the Scade model. The two interfaces are the middle layer, they have the role to provide information in a format which is easy to be used by the third layer (the implementation of the metrics). The structure of the project is presented in Figure 1.

A. High level overview

As input we have a Scade model in a specific XML format. This file is the direct equivalent of what the programmer “wrote” in Scade, it can be considered the source code of the model. It contains not only the logic of the model, for example connections and declarations, but also visual information, like exact coordinates of the rectangle representing an operator. We are assuming that the analyzed Scade model is correct, therefore we did not implement the error checking of the model.

We also need input data for the model which is being analyzed. This can be read from outside, but we developed a random input generator, in case the user does not have the inputs required by the model. This random input generator provides a series of correctly typed random input based on the types of the input variables. As it was mentioned before, Scade models are reactive models, consequently input data is needed not only for the first run cycle, but for the entire life cycle of the model.

B. The interpreter and the Parser

Our interpreter is a collection of base classes which mirrors the behavior of different Scade language elements. These base classes are provided for the parser to fulfill them with information gained from the model. The run time behavior is recorded by the recording decorator discussed in Section II-D

We have base classes for user-defined operators, control flow elements, mathematical expression and time-specific Scade operators. These base classes define in general terms how the elements of Scade language work. For example, it is specified that an if statement has to evaluate the condition, followed by the step of running some of its branches. However the content of the branches and the condition will be specified by the parser. These base classes are filled with information gained from the currently interpreted model.

The parser reads in the XML files of the model and produces classes which have the same behavior. The information obtained from the model is packed with the the base classes from the interpreter, forming together classes which have the same behavior as the corresponding model. This mechanism is done by the Python’s meta-class system, in which we can create class factories. The encapsulation is needed since some of the Scade nodes have inner state. We could have developed a system where every state of the model is stored in a global manner, but we find it easier to encapsulate them in classes and use them as we usually do in objected oriented programing.

These run-time created classes are stored following a hierarchical order. For example, the correspondent classes of the simple mathematical expressions from the example on Figure 2 are stored in classes of the branches. These branches are stored in a certain class corresponding to the when statement 1, and this is stored in operator’s class. The operator’s class is on the top of the hierarchy. They are stored in a dictionary, which has the original operator names as keys. In this way each of them can be called separately.

It has to be mentioned that the operator’s classes take as inputs the list of the values of input variables, the values of the constants and sensors 2, and the dictionary mentioned before.

1 when statement is similar to switch statement from C-like languages
2 they are like global variables in other languages

Figure 1. Component Diagram of the project

Figure 2. When block example
The dictionary has to be passed because from the inside of an operator another one can be called. The first time when an operator call happens (in the first run cycle), the class of the called operator is instantiated, and the resulting object is called with the current input values. This object is stored, and in the next run cycle it will be called with the new input values, but the states of the operator are inherited from the previews run cycle. The operators can be called only if all of the input values are calculated before, due to the fact that in Scade we do not have optional variable passing.

The mechanism of calling an internal node inside the of the Scade operator is highly similar to the one mentioned below, the main difference is that these internal nodes act in a common variable scope.

C. Scade specific details

The general implementation of our interpreter and parser has been thoroughly described, in addition we have to highlight on some Scade specific details.

1) Order of the expressions: The order of the expressions is arbitrary in Scade, due to its synchronous model. For example, we have an expression like \( a = b + 2 \) before \( b = 3 \). In order to overcome this difficulty, we decided to use a waiting queue and the well-known exceptional handling mechanism. Let us assume that all of the classes were instantiated (i.e. we are not in the first run cycle), and the resulted objects are stored in a waiting queue. The object associated with some Scade node (in case of the previous example associated with expression \( a = b + 2 \) ) is removed from the end of the queue, and it is called. If it throws a \texttt{VariableUnset} exception (specific exception with the meaning that not all the dependent variables was set in this run cycle) than we will put it back in the beginning of the queue. If the this exception was not thrown, the object is considered to be done for the current run cycle.

For the correct working of this simple algorithm all of the nodes (base classes) have to be implemented in a way, so that the consistency of the common variable scope is assured. All of them have to read first the variables needed, and make changes in the variable scope only if these were set before, otherwise they have to throw the \texttt{VariableUnset} exception.

The implementation of the nested variable scopes has to respect the working of this algorithm too. For instance, a branch of if node was called. This branch has also a few expressions which are called by the branch node in a similar way how the main operator node calls any other node. When \texttt{VariableUnset} exception occurs, the branch node has to decide if it was caused by a variable belonging to the inner variable scope or to the external one, in which case the branch also has to throw \texttt{VariableUnset} exception for the main operator. The consistency of the scopes has to be assured here as well.

2) Delay operators: Another very Scade specific idea is that the developer has access to values of the variable from previous run cycles. The developer has to initialize the first \( x \) run cycles, where \( x \) is the number of delays specified by the developer. This can be done via few built-in operator. In some cases the initialization and the accessing previous values is done by two separated operators \((\textit{Init}} \text{ and Last} \text{ operator for initializing, \textit{Pre} for accessing the previous value}) \), in other cases it is packed in one single operator \((\textit{Follow-by operator})\). Another difference is, that in the first case we can have only one as the value of the delay, in the second case we can have an arbitrary number.

The only way of communication between nodes in our interpreter implementation is the common variable Scope. To implement the first case of the delay operators, we have to modify our variable scope behavior in that way, so that the previous values of the variables are saved. In the beginning of each run cycle the values from the previous cycle are saved, and in this way they are accessible for the \texttt{Pre} operator. On the other hand the initialization can be done by introducing “fake” values to the saved area of the variable scope. This is a viable solution because the nodes can work independently, but seem to be very inefficient to store all of the previous values of the whole variable scope. We decided to store only the previous state of the variable scope.

To implement the \texttt{Follow-by} operator, the previous values of the variable is stored in the object (instantiated from the run time generated class associated with the original \texttt{Follow-by} node from the model). This is possible because no other node needs to read or write the long term history of specific variable.

D. Metrics interfaces

1) Static metric interface: The static metric interface provides an easy access to the information read in by the parser. This data is grouped in tree, which are representing the structure of a Scade model. This structure is similar to the class hierarchy as shown above but it contains only the relevant part of that information. For example, in this case the types of the variables are not stored. Our abstract syntax tree (AST) structure is not as detailed as the conventional ASTs [13]. In our implementation a mathematical expression like \( a = b + c \) is stored in one node and not in a sub tree with five nodes. In the leaf nodes all of the expressions are stored, not only the mathematical ones, but also if and case expressions, and those which are used for accessing the previous values of a variable. Non-leaf nodes are representing the elements of the control flow like if and when statements and state machines.

2) Dynamic metric interface: A similar structure is applied in case of dynamic information, although the nodes of the dynamic abstract syntax tree (DAST) contain additional information. This information is gained by storing the change of a variable scope in these nodes. For example, in case of the expression \( L1 = \text{pre}(L2) \), the variable \( L1 \) is stored with its new values. The result of this recording is a separate DAST for every run cycle. These DASTs contain only those elements of the Scade model, which were interpreted in current run cycle. For example, in case of an if statement only one of the branches is included in a DAST.

The nodes of the DAST are built run-time with the help of the Python’s decorators. This is another meta-programing abstraction used to wrap functions or classes injecting both entry- and exit functionality to the original function without changing its name or any other important characteristic. This
pattern is used to record the calling of every object associated with Scade element. 

Our recording decorator does a few simple steps (let us assume that in global variable current active variable we store DAST node that represents the currently interpreted Scade element):

1) Saves the current_node (to saved_current) and the current state of variable scope.
2) Creates the new DAST node representing the actual Scade element and replaces the current_node with new node.
3) Calls the object of the Scade element.
4) After that, if the VariableUnset exception appears, restores the current_node from the saved_node.
5) Else adds the current_node to the children of saved_current.

Has to be emphasized the hidden recursion of the algorithm. If the interpreted Scade element has sub-elements, in step 3 this decorator will run several times recording the lower order elements as a child, while the higher order element is not finished. For example, an else branch of an if statement is called. The current_node is the one associated with the else branch (step 3) and the presented decorator is called for the expression inside of the branch, adding children to the DAST node of the else branch. After every inner expressions are finished, the else branch node is added as child to the DAST node of the if statement (step 5).

In contrast to AST, in which the order of the siblings is arbitrary, in case of DAST the order of the siblings gives the order of their successfully interpretation. As a result the DAST can give a complete picture of the dynamic behavior of the model.

III. EXAMPLE OF METRICS IMPLEMENTATION

Inspired by Python’s ast.NodeVisitor [14], we also implemented our AST and DAST visitor. These are classes providing a useful interface for the tree traversal. These visitor-based classes walk the AST or DAST and call a specific visitor function for every node found. Sub-classing these visitors gives conformable access to the information stored in the trees described above.

In the instance of Listing 1 a simple example of calculating branch coverage is presented. The first class BranchDinRecorder is a subclass of the DAST visitor, has the functionality to record the ids of the visited branches. (Ids are unique identifier number for every Scade element) The second class (BranchStatRecorder) has the same task but for static trees. The BranchCoverage has as input the DAST recorded run-time (called dasts in the example), and the AST of the model. This class collects the unique ids of the run-time called branches (from all run cycles) and the ids of the branches from the whole model. The ratio between the length of the lists gives the the branch coverage.

As it was presented, the implementing code coverage is straight forward with these interfaces and data structures. Also the implementation of or metrics defined by Halstead is relatively easy. We only have to record the unique operators and operands, and count the total number of them. In case of source code, we have to subclass the AstVisitor and implement the specific functions similarly to the example presented above.

In contrast to code coverage and Halstead’s Software Science, metrics of the control graph is much more difficult to implement. These data structures do not contain information about the relationship between siblings. We do not know whether two sibling nodes from AST tree are independent or they are in the same data flow. For example, an operator node has two siblings, both of them corresponding to an if statement. They can depend on each other (the first if statement’s branches sets a variable used in the second if statement condition), or they can be completely independent (they use same variables in different ways), and the AST will have exactly the same structure. Additionally in Scade, there are also if and case expressions which are modifying in data flow of an Scade operator. Their order and dependencies are also unrecoverable from the AST.

This lack of information is due to the fact that this information is specified in the Scade model, and the presented interpreter also works without them. To convert the presented
Figure 3. Added code measured with HSS metrics

AST to control flow graph and measure the cyclomatic complexity of a Scade operator, we investigated all the data flows inside in the operator staring from the input variables. We also have to determine control flow variables and if they are set in of the branches of if or when statement, or states of the state machines. Summarizing these information we achieved the control flow graph as well.

IV. EXPERIMENTAL RESULTS

In this section some experimental result will be presented. These results was obtained by measuring ten version of the same model developed by a team working in Scade. In the Figure 3 is demonstrated the results of the HSS metrics. In this experiment only the added code in the current version of the software was measured. It can be deduced that the is version 4 and 8 there is no code added. It is also observable that in version 6 and 7 the Difficulty of the added code is bigger than in the version 2 or 3, however the amount of code added (the Volume of the program and the Effort) is higher in the second case.

In the Figure 4 present the result obtained by applying the cyclomatic complexity and the tree kind of code coverages mentioned in the introduction. These coverages were measured with tree sets of random input data. Zero coverage is measured when the specific type of coverage is not applicable. For example in case of the first five version the model does not contain state machines, as a result our interpreter measures zero state coverage.

The average cyclomatic complexity was calculated by averaging the cyclomatic complexity of the different user defined operators, in the particular version of the model.

V. CONCLUSION AND FUTURE WORKS

All in all, we implemented a simple Scade interpreter capable to measure most of the canonical metrics, like HSS, cyclomatic complexity and three types of branch coverage. We focused on the flexible design, which allows us incremental development with the use of the Objected Oriented patterns.

In the future, the optimization of our interpreter is needed with the information gained from control flow tree to obtain faster dynamic analysis. This solution should override the issues discussed at section II-C1 and should lead to serious speed gain in the first run cycle.

Another possible direction of our further work is developing an input generator, which is capable of generating test input with high coverage for the Scade model, or as an option we can improve the existing test input data from the requirement for the same reason. This future will also help us to validate our interpreter implementation and discover unwanted behavior. Another possibility opened by this future is code profiling to help developers to optimize their Scade model.

Finally we also want to implement other known metrics, and integrate our project with version control system to give an automated daily feedback for the Scade developers.

REFERENCES