CodeBreakers Magazine
Security & Anti-Security - Attack & Defense
Volume 1, Issue 1, 2006

Towards a Framework for Assembly Language Testing

Dr. Thorsten Schneider [Schneider@Secure-Software-Engineering.com]
March 2006

Abstract
Testing of software is crucial for assuring software quality, validity and reliability. With the background of many existing software testing frameworks for high level languages, this paper introduces the concept of an Assembly Testing Framework (ATF) including Code Metrics, Code Coverage and Unit respective Functional Testing for the Assembly programming language. There is no testing framework for Assembly language to my knowledge yet.

1 Introduction

Testing of software is an essential method of assuring software quality, validity and reliability as described by Perry [1] and Beck [2]. Most used testing approaches with focus on High Level Languages (HLL) - like Java, C++ and similar - are assisted by testing frameworks. Examples are are given by Burke [3], Marick [4], Clark [5], and Dyuzhev [6]. Actually there is no approach covering the Assembly language software development processes. Most differences between Assembly and HL languages like C++ are the non-existence of direct access to object oriented design and the ability of Assembly to work nearest to the hardware layer. Assembly is a powerful language but the developer has absolutely to know what the code might induce. Even HL languages are most common in software engineering; there still is a need for Assembly code, especially if performance is a fundamental ingredient. Typical examples are server side tools, micro controller applications or embedded systems. Another example is the maintenance of old software systems still in use by banking systems, health care and financial departments. The community behind Assembly language is larger than most HL developers believe: just the *Win32Asm Community Board* alone has more than 5200 users with over 110.500 posts [7]. Considering the importance of such a widespread developing language, it is a necessity to provide a framework for testing Assembly code assisting software quality and validity during development processes.

Talking about software testing one has to differ between several common keywords: Unit Testing, Functional Testing, Performance Testing, Automated Testing, Regression Testing, Code Coverage and Code Metrics. Unit tests are written from a programmer's perspective. They ensure that a particular part of the software (for example a method or a class) successfully performs a set of specific tasks. Contrary to this, functional tests are written from a user's perspective. These tests confirm that the system does what users are expecting it to. Performance testing is mostly done by profiling tools which detect bottlenecks of running applications or processes. Automated tests require skill, patience and above all, organization. Mostly they consist of a suite of tests containing repetitive tests which can be broken down to several test scripts. As difference to the other testing methods, Regression Testing is testing that an application has not regressed, which means simplified that the functionality that was working yesterday is still working today. In contrast Code Coverage and Code Metrics are no testing methods at all. Both belong to the empirical software engineering part (quality aspects) and provide information about several internal application and test statistics.

A framework for Assembly language testing (ATF) needs to cover all the above mentioned methodologies. As well, it needs either to integrate in an Integrated Development Environment (IDE) like RadASM [8] or to provide a standalone solution which one can work with. Additional such framework would offer two testing options instead of only one provided by HLL testing frameworks. Whereas HLL frameworks only inspect at source code level, an Assembly framework is able to analyse the Assembly source code (represented for example by TASM, NASM or MASM syntax) as well as the compiled code represented by disassembled code, which might differ from the original sources due to compiler optimization tasks. This is where Unit Testing is the most effective way to reveal as many errors in code as possible. Additionally such modular testing methodologies have a high cost-profit-ratio, which make them first choice for critical projects at all. Since Assembly language projects are organized very modular, testing of such small modules before integration increases stability and reliability of the final software product. In the end Assembly programmers at all miss well established development environments and

development tools - in most cases such developers work with simple text editors and sequential batch files for compiling and debugging code, which often leads into brute forcing the erroneous code part.

2 Code Metrics for Assembly Programs

Dealing with Assembly programs' code complexity, one is able to achieve information on how difficult it is to comprehend, modify and generally maintain an application. Software metrics can be classified to program size, program control structures (CFG and ICFG) and extensions to control flow metrics [9]. Other metrics are neural net-based metrics as described by Boetticher et al. [10]. One most used method in the past is counting *Lines of Code* (LOC) or *Number of Non-Commented Source Statements* (NCSS) but has been dropped due their questionable significance.

One typical program size metric is the Halstead Metric described by Halstead [11]. This metric measures program vocabulary ($n = n_1 + n_2$), program length ($N = N_1 + N_2$), program volume ($V = N \cdot \log_2 n$), estimated program level ($L' = 2/n_1 \cdot n_2/N_2$) and program effort ($E = V/L'$), with given n_1 as number of unique operators, n_2 as number of unique operands, N_1 as total count of all usages of operators and N_2 as total count of all usages of operands [12, 13].

The McCabe cyclomatic complexity metric (CCM) is a program control structure metric which has been described in detail by McCabe himself [14] but others as well [15]. This metric converts the application into a directed graph (DG) representing a control flow graph (CFG). If this graph confines to sub graphs representing procedural flow, it is named as intraprocedural control flow graph (ICFG). The CCM is defined as $V(g) = Edges - Nodes + 2 \cdot Vertices$, where *Edges* is the total number of edges within the graph, *Nodes* is the total number of nodes in the graph and *Vertices* is the total number of connected components of the graph. It has to be noticed, that the CCM of any application is equivalent to the number of binary predicates *Predicates* in the program ($V(g) = Predicates + 1$).

Logical complexity has been described by Gilb as Absolute Logical Complexity (ALC) and as Relative Logical Complexity (RLC) [12]. ALC is equivalent to McCabes $V(g) - 1$ and RLC is defined as ALC/T_S, with T_S as total number of statement within the application.

One more complexity measurement for CFGs by counting *knots* has been proposed by Woodward et al. [16], where a *knot* is defined as unavoidable crossing of control paths (branches) in the DG representation. Compared to the McCabe complexity it offers a different aspect of routines control complexity characteristics.

Blaine and Kemmerer extended these methods with analysis procedures of Maximum Knot Depth (MKD) and Knots Per Jump Ratio (KPJR) [9]. MKD is defined as the number of knots produced by a branch. Against this the KPJR normalizes the knot count with respect to the number of branches in an application.

Regarding to an Assembly Testing Framework, the developer requires detailed information about these statistical information to keep software maintainable by reducing complexity as much as possible, which reflects the KIS (Keep It Simple) paradigm praised by the Assembly Programming Community.

3 Code Coverage of Assembly Programs

Code Coverage (synonym: test coverage) is an enhanced method in finding areas within an application which are not exercised by a set of test cases. Since HL languages produce more semantic and syntactic complex code than Assembly languages, several coverage

measures branched, as defined by Cornett [17]:

- Basic Measures:
  - (Statement Coverage, Decision Coverage, Condition Coverage, Multiple Condition Coverage, Codition/Decision Coverage, Path Coverage)

- Other Measures:
  - (Function Coverage, Call Coverage, Linear Code Sequence and Jump (LCSAJ) Coverage, Data Flow Coverage, Object Code Branch Coverage, Loop Coverage, Race Coverage, Relational Operator Coverage, Weak Mutation Coverage, Table Coverage)

Even reducing to the simple structure of Assembly language applications, these coverage metrics can be used to identify non-tested fragments within the code.

The Code Coverage results in a percentual value in most cases, which assures the quality of the test sets, but not of the actual product. This means, that using coverage methods do not assure a finally a 100% bug free application, but means that the involved test cases cover the existing code with a resulting percentual value. Therefore the tested application is only as good as the test cases are covering all possible events during the applications life.

In general Code Coverage is a structural testing technique known as glass box testing or white box testing as well. Against this functional testing is known as black-box testing. Whereas structural testing compares test program behaviour against the apparent intention of the source code, functional testing compares application behaviour against a requirements specification. Simplified structural testing examines how the program works (structure and logic), and functional testing examines what the program accomplishes (internal working).

To refer to Cornett [17], faults (a bug or defect) relate to control flow and can be exposed by varying the control flow [18]. Secondly it is possible to look for failures (the runtime manifestation of a fault) without knowing what failures might occur [19]. Additional it has to be stated, that coverage analysis exposes some plausible faults but does not come close to exposing all classes of faults.

4 ATF CFG Reconstruction from Assembly Code

For reconstructing the control flow graph (CFG) from Assembly Code ATF confines to the interprocedural control flow graph (ICFG) as described by Kästner and Wilhelm [20]. Using the ICFG results in two different subgraphs: (1) a call graph (CG) which describes relationships between program's procedures and (2) a basic block graph (BBG) describing the intraprocedural control flow of each procedure.

The CG describes the relationships between procedures. Nodes are procedures and edges (vertices) are procedure calls. A BBG describes the ICFG for every procedure. Nodes are basic blocks, and a basic block is defined as sequence if instructions that are executed under the same control conditions. The interested reader is pointed to references [20] and [21] for detailed information about CFG reconstruction from Assembly code.

5 Software Testing with the Assembly Testing Framework (ATF)

The Assembly Testing Framework (ATF) (see figure 1) provides a testing environment including Code Metrics and Code Coverage

methods. Similar to other HLL testing frameworks it offers Asserts for testing code implementations. As difference to HLL testing environments it assists in using source code as well as disassembly resolved directly from binary compiled code. This comes handy when one is dealing with testing of compiler optimization processes or Reverse Code Engineering (RCE) tasks.

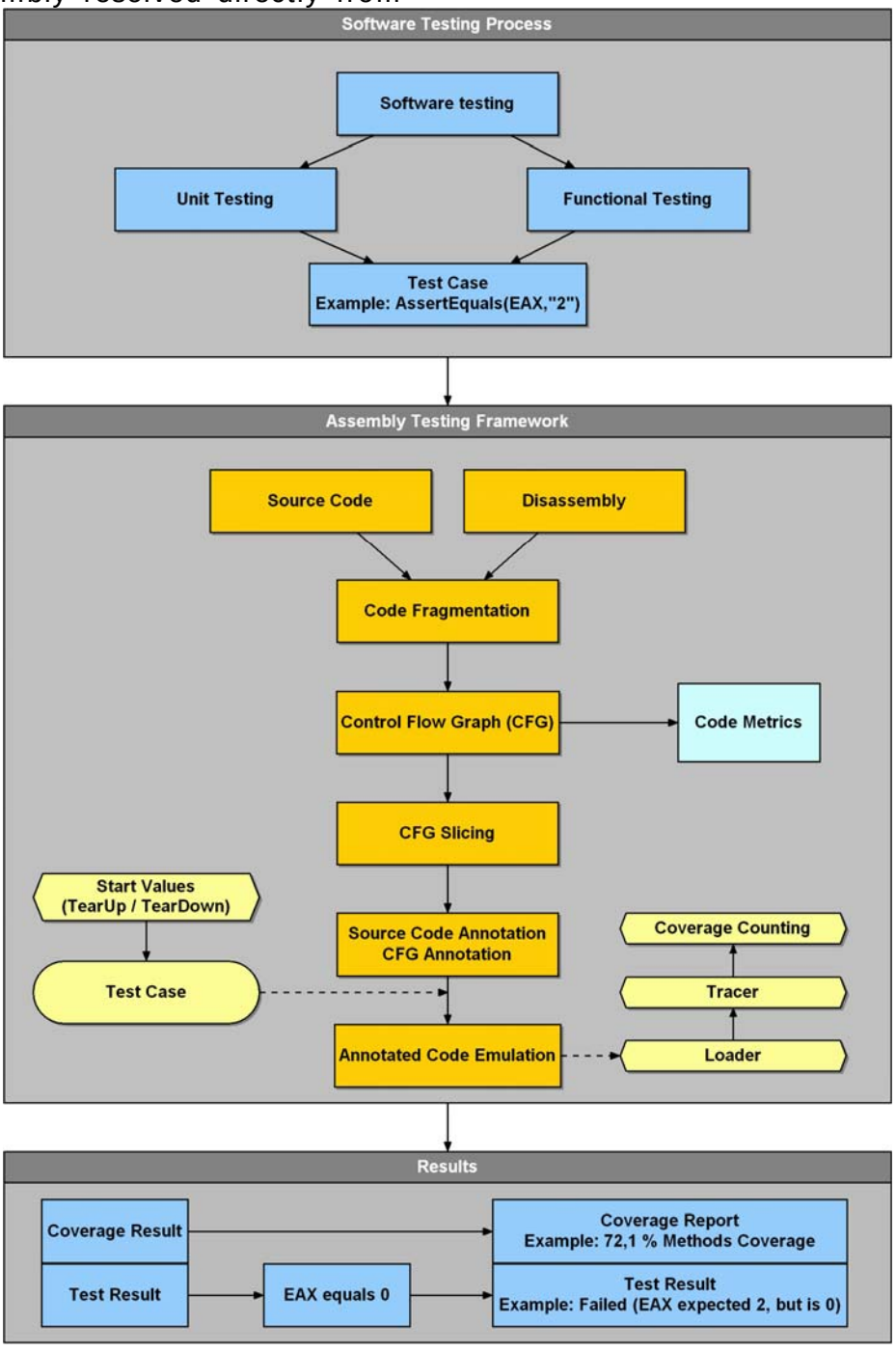

Figure 1: *Overview of the Assembly Testing Framework (ATF). The diagram shows the three main layers of the framework structure: the software testing process layer, the framework layer and the result layer. After defining Test Cases the framework uses either source code or disassembly for code fragementation and CFG construction. Before running source code annotation and emulator for the input code it is possible to detach Code Metrics from the CFG. Using predefined startup values and code annotation, the emulator extracts Coverage Metrics and Coverage Counting using a Tracer and reports the results as Code Coverage and Assert test results to the Result Layer.*

Beginning the software testing process one has to define Test Cases using Assert Statements. Common used Asserts are `AssertEquals` or `AssertTrue` respective `AssertFalse`. As comparison to HLL testing frameworks, Assembly `setUp` and `tearDown` methods differ. Since assembly language is based on heavy usage of processor registers, polymorphic code, self-modifying code (SMC) or anti-debugging and anti-tracing tricks which are common techniques for copy protected applications. Additional one future feature of ATF can be to generate automated tests from code as described by Boyapati et al. [22] and Marinov et al. [23]. This gives the ability to compare self-defined tests against

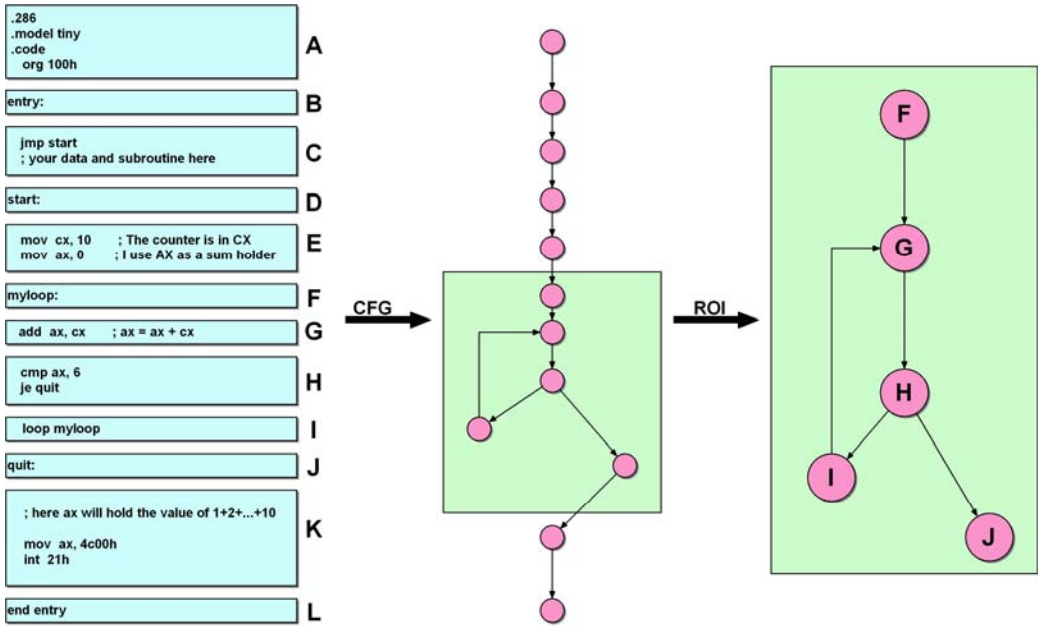

Figure 2: *Code fragmentation of a MASM style source code (left). After source code fragmentation the fragments are converted to the corresponding control flow graph (CFG, center). Using Program Slicing the Region Of Interest (ROI) is extracted (right).*

one is able only to preset and to evaluate values within registers like EAX, EBX, ESI and others including corresponding Flags. This differs from HLL testing since it is not possible to use complex constructs within an Assert Statement, like `[AssertEquals("2",myClass.getResult("1+1"))]`. Instead a valid Assert statement would be `[AssertEquals(EAX,"2")]`. Note the different sequence in comparison to HLL asserts, which reflects the reverse parameters of Assembly opcode mnemonics. Playing with these registers needs careful handling and a secure emulation engine to prevent buffer overflows or abuse of registers which might crash the testers host machine. Especially for high security applications this raises to problems due possible heavy usage of the automated results to increase code coverage productivity.

In the next step the Test Case is connecting to ATF. According to the decision of the tester, ATF takes either source code or disassembly for further testing and analysis. Since source code - given in a special syntax like the MASM, TASM or NASM syntax - differs only slightly from the resulting assembly code, it is an easy task to convert source statements to resulting assembly code.

Within the next steps the Assembly code is fragmented (see figure 2) into its substantial parts and converted into a control flow graph (CFG) as described by Cooper et al. [24]. The resulting CFG is used to gain first information

about the code structure. At this point code complexity measures like Code Metrics are detached. Using the CFG one is able to reduce the working element by using Program Slicing methodologies on the CFG as proposed by Beck and Eichmann [25] to produce slices containing the Region Of Interest (ROI). While non-interesting parts are dropped out of the process, the remaining parts obtain more importance for the following testing process.

After extraction of CFG and Code Metrics, ATF annotates the (source) code for further processing by Test Cases. In difference to HLL testing frameworks, in the case of Assembly it is the most important task to define setUp and tearDown parameters, including initialization of register settings which are needed for program consistency. Missing these parameters, the Annotated Code Emulation (ACE) is not able to emulate the code segment properly and could cause an emulator crash. This is where a good structured exception handling within the emulator engine is necessary, which is part of future research.

The Emulator consists of three main parts: (1) a Loader, (2) a Tracer and (3) a Coverage Counting Method. The Loader takes the annotated (source) code, presets the necessary startup register settings and hooks the injected code to the emulation engine. To force a loader reading dynamic code ATF uses reflection methods similar to those described by Knizhnik [26] and Roiser [27]. For injecting the code routine ATF uses dynamic injected inline assembly structures containing the to-test Assembly code. One main ATF feature is the Tracer which connects to the emulation process using the internal debugging abilities of ATF and reports each hit of an annotated mnemonical opcode operation to the Coverage Counting Method. Finishing the Trace, the emulator reports two main results: (1) The Coverage Result represented by the Coverage Counting Method result and (2) the status of the registers after the trace which is evaluated later to build up the result of Test Cases and Asserts.

Reaching the Result Layer (see figure 1) a Coverage Report is produced by the results of the ATF emulator (Tracer-Coverage Counting) and can be evaluated by further analysis. For checking Test Cases respective their associated Asserts, the register settings of the ATF emulator are checked against the corresponding Assert Statements. One example is to check a register against a given value ([`AssertEquals(EAX,"2"]`) or another register value ([`AssertEquals(EAX,EBX)`]). One extension to the standard Assert is `AssertFlag` which checks directly against flag settings.

6 Conclusions and Future Work

An Assembly Language Framework (ATF) for unit and functional testing of Assembly language opens testing processes even to lower programming languages. With reference to industrial applications, several current systems are not programmed with HL languages like C++ or Java. Mostly such systems need Assembly when a HLL does not come handy - for example increasing performance (e.g. banking) or using very specialised chipset functionalities (e.g. security applications). Since such systems are common within high security or high risk environments (e.g. medicine, banking or space-related projects) it is of high relevance to control the internal quality and validity of developed or maintained code. ATF is leaned on current existing frameworks and adapts to existing standards (e.g. Test Cases or Assert methodology). Additional it is easy to understand from the developer's and user's point of view. Future work will include implementing and enhancing ATF as well as testing within real Assembly coding processes. Additional automated testing methods should be applied to improve testing processes.

7 Acknowledgments

We thank R.A. Kemmerer for giving assistance during our enquiry of existing literature.

8 References

1. Perry WE: **Effective Methods for Software Testing**, 2nd edn: John Wiley & Sons; 2000.
2. Beck K: **Test Driven Development: By Example**: John Wiley & Sons; 2002.
3. Burke E: **eXtreme Testing**. *St Louis Java User's Group,* http://wwwociwebcom/javasig/knowledgebase/Oct2000/ 2000.
4. Marick B: **Testing for Programmers**. http://wwwtestingcom/writings/half-day-programmerpdf 2000.
5. Clark M: **JUnit Primer**. http://wwwclarkwarecom/articles/JUnitPrimerhtml 2000.
6. Dyuzhev V: **TUT: C++ Unit Test Framework**. 2004.
7. Hiroshimator: **Win32ASM Community Messageboard**. http://boardwin32asmcommunitynet/ 2005.
8. KetilO: **RadASM assembler IDE**. http://wwwradasmcom/ 2006.
9. Blaine JD, Kemmerer RA: **Complexity Measures for Assembly Language Programs**. *Journal of Systems and Software, Elsevier Science Publishing Co Inc* 1985:229-245.
10. Boetticher G, Srinivas K, Eichmann D: **A Neural Net-Based Approach to Software Metrics**. In: *Fifth International Conference on Software Engineering and Knowledge Engineering: June 16-18 1993; San Francisco, CA*; 1993: 271-274.
11. Halstead MH: **Elements of Software Science**. *Elsevier North-Holland, New York* 1977.
12. Gilb T: **Software Metrics**. *Winthrop Publishers, Cambridge, MA* 1977.
13. Bailey CT, Dingee WL: **A Software Study Using Halstead Metrics**. *ACM SIGMETRICS Performance Evaluation Review* 1981, **10**(1):189-197.
14. McCabe T: **A Complexity Measure**. *IEEE Transactions Software Eng* 1976:308-320.
15. Watson AH, McCabe T: **Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric**. *NIST Special Publication 500-235* 1996.
16. Woodward M, Hennell M, Hedley D: **A Measure of Control Flow Complexity in Program Text**. *IEEE Trans Software Eng* 1979:45-50.
17. Cornett S: **Code Coverage Analysis**. http://wwwbullseyecom/coveragehtml 2004.
18. Beizer B: **Software Testing Techniques**, 2nd edn: Van Nostrand Reinhold, New York; 1990.
19. Morell L: **A Theory of Fault-Based Testing**. *IEEE Trans Software Eng* 1990, **16**(8):844-857.
20. Kästner D, Wilhelm S: **Generic Control Flow Reconstruction from Assembly Code**. In: *LCTES'02-SCOPES'02, Berlin, Germany: 2002*; 2002.
21. Venkitaraman R, Gupta G: **Static Analysis of Embedded Executable Assembly Code**. 2004.
22. Boyapati C, Khurshid S, Marinov D: **Korat: Automated Testing Based on Java Predicates**. In: *ACM International Symposium on Software Testing and Analysis (ISSTA): July 2002 2002*: ACM; 2002.
23. Marinov D, Khurshid S: **TestEra: A Novel Framework for Automated Testing of Java Programs**. 2002.
24. Cooper KD, Harvey JT, Waterman T: **Building a Control-Flow Graph from Scheduled Assembly Code**. *Rice Technical Report, TR02-399* 1999.
25. Beck J, Eichmann D: **Program and interface slicing for reverse engineering**. In: *Proceedings of the 15th international conference on Software Engineering: 1993; Baltimore, Maryland, United States*: IEEE Computer Society Press; 1993: 509--518.
26. Knizhnik K: **Reflection for C++**. http://wwwgarretru/~knizhnik/cppreflection/docs/reflecthtml 2004.
27. Roiser S: **Seal C++ Reflection Package**. http://sealwebcernch/seal/snapshot/workbook/reflectionhtml 2004.