Vol. 2, No. 1 (2005)
http://www.CodeBreakers-Journal.com

Towards a Framework for Assembly Language Testing

Thorsten Schneider* (University of Hannover)

* Corresponding Author

Received: 10. Feb. 2005, Accepted: 18. Feb. 2005, Published: 20. Feb. 2005

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 an Assembly Testing Framework (ATF) including Code Metrics, Code Coverage and unit respective functional testing for the Assembly programming language.

Keywords: Assembly Testing Framework (ATF); Assembly Language; Software Testing; Code Coverage; Code Metrics; Unit testing

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 focus on High Level Languages (HLL) like Java, C++ and similar are assisted by testing frameworks. Examples are given by Burke [3], Marick [4], Clark [5], and Dyuzhev [6]. Actual there is no approach covering Assembly language software development processes. Even HL languages are most common in software engineering, the need for Assembly code is still there, 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 system still in use by banking systems and financial departments. Next the community behind Assembly language is larger than most HL developer believe: just the *Win32Asm Community Board* alone administrates more than 5200 users with over 110.500 articles [7]. Considering the importance of such a widespread developing language it is a necessity to provide a framework for testing Assembly code assisting for 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. Against 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 not a testing method at all. Both methods 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 needs to cover all the above mentioned methodologies. As well it needs to integrate either in an Integrated Development Environment (IDE) or to provide a standalone solution which one can work with. Additional such framework offers 2 testing options instead of only 1 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 compiler optimization tasks.

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. Several metrics have been developed in the past, including Halstead Metrics [8], McCabe Metrics [9] or neural net-based metrics [10]. Other methods like counting lines of code (LOC) or number of non-commented source statements (NCSS) have almost been dropped due their questionable significance. Blaine and Kemmerer extended these methods with analysis procedures of Maximum Knot Depth (MKD) and Knots Per Jump Ratio (KPJR) [11]. Regarding to an Assembly Testing Framework the developer requires detailed information about such 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 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 [12]:

1. Basic Measures (*Statement Coverage, Decision Coverage, Condition Coverage, Multiple Condition Coverage, Condition/Decision Coverage, Path Coverage*)
2. 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*)

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

4. Testing with the Assembly Testing Framework (ATF)

The Assembly Testing Framework (ATF) (see fig. 1) provides a testing environment including Code Metrics and Code Coverage methods. Similar to other HLL testing frameworks it offers using Asserts for testing code implementations. As difference to HLL testing environments it assists in using source code as well as disassembly resolved from binary compiled code. This comes handy when one is dealing with testing of compiler optimization processes or Reverse Code Engineering (RCE) tasks.

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, one is able only to preset and to evaluate values within registers like EAX, EBX, ESI and others. 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 standard 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 polymorphic code, self-modifying code (SMC) or anti-debugging and anti-tracing tricks.

In the next step the Test Case connects to ATF. According to the decision of the tester, ATF takes either a source code or a disassembly for further testing and analysis. Since source code - given in a special syntax like the MASM, TASM or NASM - differs 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 fig. 2) into it's substantial parts and converted into a control flow graph (CFG) as described by Cooper et al. [13]. 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 [14] 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.

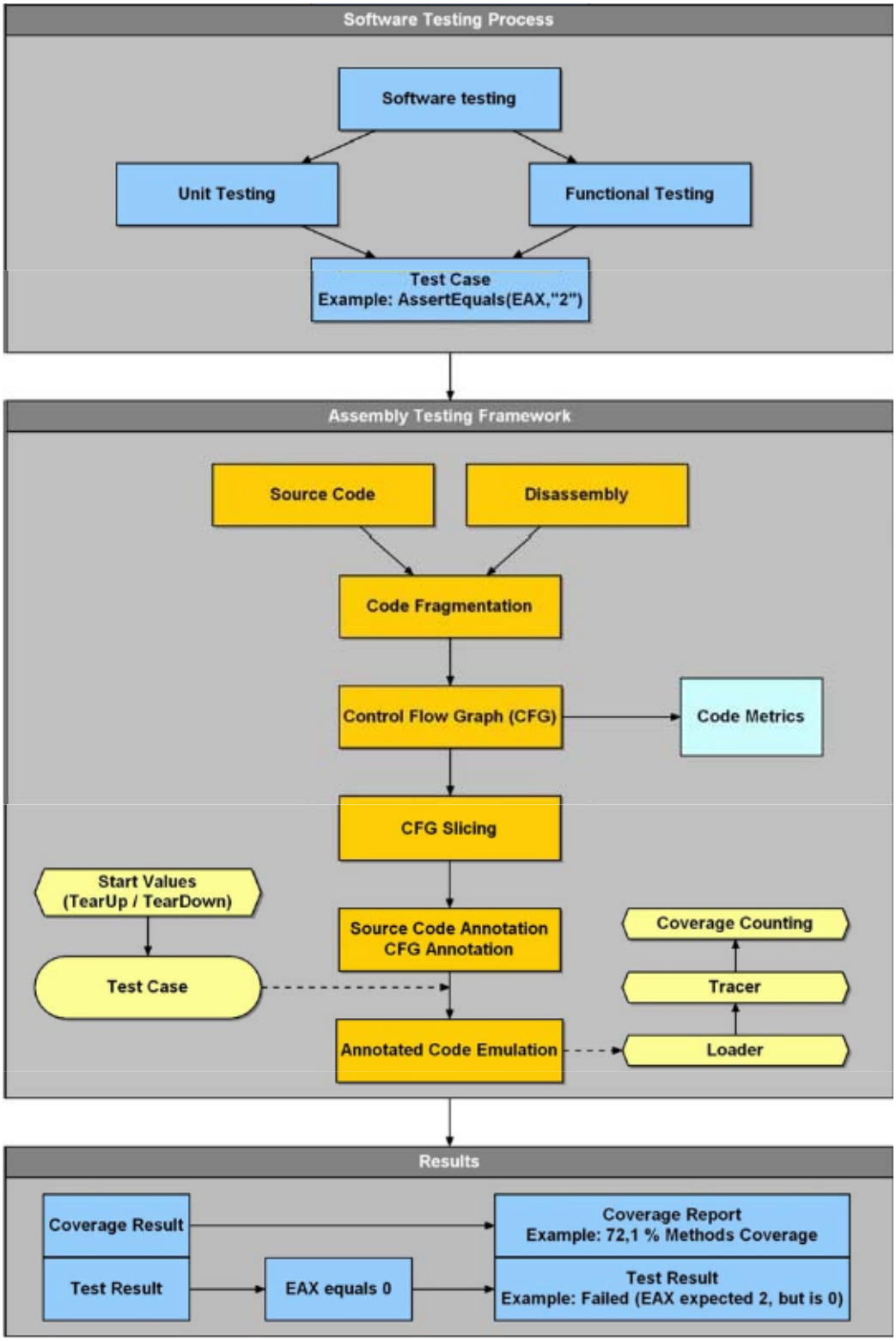

Fig. 1: Overview of the Assembly Testing Framework (ATF). The diagram shows the three main layers of the framework structure: the software testing process layer, the framework layer and the result layer. After defining Test Cases the framework uses either source code or disassembly for code fragmentation 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 start-up values and code annotation, the emulator extracts Coverage Metrics and Coverage Counting Methods using a Tracer and reports the results as Code Coverage and Assert test results.

After extraction of CFG and Code Metrics, ATF annotates the (source) code for further processing by Test Cases. As difference to HLL testing frameworks, it is in the case of Assembly most important to define setUp and tearDown parameters, including initialisation 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.

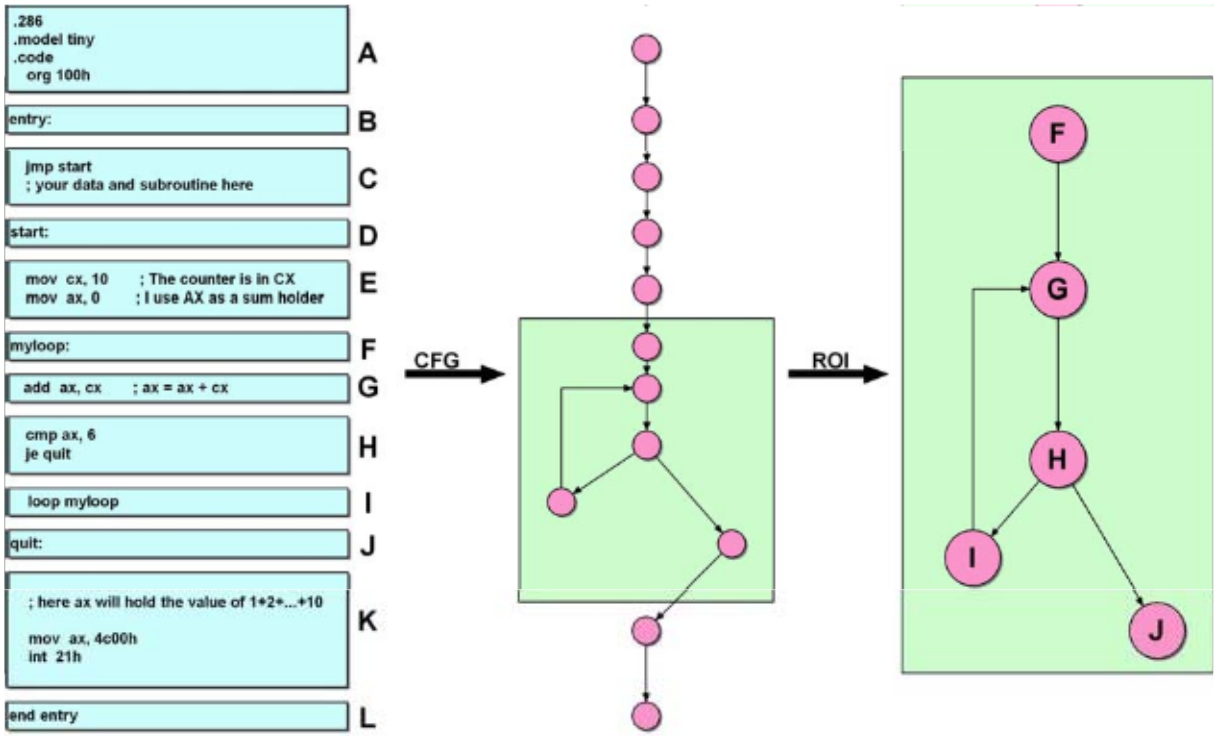

Fig. 2: Code fragmentation of a MASM style source code (left). After source code fragmentation the fragments are converted to the corresponding control flow graph (CFG, centre). Using Program Slicing the Region of Interest (ROI) is extracted (right).

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 as described by Knizhnik [15] and Roiser [16]. For injecting the code routine ATF uses dynamic injected inline assembly structures. One main 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 and (2) the status of the registers after the trace which are evaluated later to build up the result of Test Cases and Asserts.

Reaching the result layer (see fig. 1) a Coverage Report is given by the results of the ATF emulator (Tracer Coverage Counting) and can be evaluated for 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)]).

5. 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 in use 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 importance 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). ATF 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.

References

[1] W. E. Perry, *Effective Methods for Software Testing, 2nd Edition*: John Wiley &Sons, 2000.

[2] K. Beck, *Test Driven Development: By Example*: John Wiley &Sons, 2002.

[3] E. Burke, "eXtreme Testing," *Website accesed Dec. 2004. St. Louis Java User's Group,* http://www.ociweb.com/javasig/knowledgebase/Oct2000/, 2000.

[4] B. Marick, "Testing for Programmers," *Website accessed Dec. 2004,* http://www.testing.com/writings/half-day-programmer.pdf, 2000.

[5] M. Clark, "JUnit Primer," *Website accesed Dec. 2004.* http://www.clarkware.com/articles/JUnitPrimer.html, 2000.

[6] V. Dyuzhev, "TUT: C++ Unit Test Framework," *Website accesed Jan. 2005.* http://tut-framework.sourceforge.net/, 2004.

[7] Hiroshimator, "Win32ASM Community Messageboard," 2004.

[8] C. T. Bailey and W. L. Dingee, "A Software Study Using Halstead Metrics," *ACM SIGMETRICS Performance Evaluation Review*, vol. 10, pp. 189-197, 1981.

[9] T. McCabe, "A Complexity Measure," *IEEE Transactions Software Eng.*, pp. 308-320, 1976.

[10] G. Boetticher, K. Srinivas, and D. Eichmann, "A Neural Net-Based Approach to Software Metrics," 1993.

[11] J. D. Blaine and R. A. Kemmerer, "Complexity Measures for Assembly Language Programs," *Journal of Systems and Software*, pp. 229-245, 1985.

[12] S. Cornett, "Code Coverage Analysis," *Website accesed Nov. 2004.* http://www.bullseye.com/coverage.html, 2004.

[13] K. D. Cooper, J. T. Harvey, and T. Waterman, "Building a Control-Flow Graph from Scheduled Assembly Code," 1999.

[14] J. Beck and D. Eichmann, "Program and interface slicing for reverse engineering," presented at Proceedings of the 15th international conference on Software Engineering, 1993.

[15] K. Knizhnik, "Reflection for C++," *website accesed Dec. 2004.* http://www.garret.ru/~knizhnik/cppreflection/docs/reflect.html, 2004.

[16] S. Roiser, "Seal C++ Reflection Package," *Website accesed Jan. 2005.* http://seal.web.cern.ch/seal/snapshot/workbook/reflection.html, 2004.