Graz University of Technology

Institute for Applied Information Processing and Communications

Master's Thesis

Measuring Code Coverage on an Embedded Target with Highly Limited Resources

Philipp Pani

Graz, Austria, April 2014

Advisor

Prof. Roderick Paul Bloem

Institute for Applied Information Processing and Communications

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Abstract

Microchips are integrated into everyday items like running shoes, membership cards and car keys, to name just a few examples. Hence, we use microchips everyday. We rely on them, whether consciously or not. Consequently, we rely also on the software running on microchips. Testing is essential for achieving confidence in the correctness and the reliability of the software. Measuring the code coverage of a software provides information on the quality of a test suite. A high-quality test suite helps to ensure that a software fulfills its specification.

Measuring code coverage usually increases the execution time, the size of the executable and the memory usage of a software under test. For the majority of systems this is not a problem. However, for embedded systems with limited resources this can be an issue. For such systems, available out-of-the-box systems cannot be easily applied.

In this work we developed, implemented and evaluated two approaches to measure coverage in embedded systems with very limited resources. The first one is based on binary instrumentation and the second one is based on source code instrumentation. The idea of the binary approach is to patch the return instruction of each function with a call to a special function in order to measure function coverage. The second approach, based on source code instrumentation, is able to measure decision coverage. It uses a tool called Coccinelle[1] to get a token stream of the source code. From this we produce an instrumented version of the source code, cross-compile it and deploy the resulting executable on the target device. Then we examine the results from the device and update the instrumented source code. It iteratively repeats this steps until it does not collect new probes. At the end it creates a visual report of the results.

Finally, we evaluated the approaches on a firmware for a MRK-IIIe microcontroller from NXP Semiconductors. Our evaluation shows that our method is applicable to systems with very limited resources. Furthermore, the overhead in the test suite execution time is acceptable in this industrial case study.

[1]http://coccinelle.lip6.fr/

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Contents

1.1 Background and Motivation

The growing demand for embedded microchips [27] has pushed their development over the last decades to the effect that microchips can be found just about everywhere today. They are integrated in items like running shoes, animal ear tags, keys, and even gravestones [1]. Hence people rely on computer systems more than ever. This includes hardware as well as software. It is widely accepted that software testing is essential to develop reliable systems [28]. A lot of tests are needed to be fairly confident that a software behaves as it should. Ideally, every possible input combination for a program should be tested [13]. Unfortunately the number of input combinations is impractically or even infinite large in real world programs, even if we just want to test every possible execution path [35]. For this reason, only a small subset of the possible input combinations can be tested. However, we need a measure to quantify how well such subset tests a given program.

Code coverage is a well known measure to determine how well a program is tested by a test suite with respect to a specific criterion. It provides information on the quality of the program as well as on the quality of the test suite. A high code coverage indicates a suitable

[1]http://www.siliconvalley.com/ci_16866566, last visited on 2014-02-04

test suite and a low amount of unreachable source code. The notion of code coverage is already more than 50 years old. A lose definition was already given by Miller and Maloney [32] in 1963. Many different criteria can be used to determine the code coverage of a program. For instance function coverage, statement coverage, branch coverage, and modified condition/decision coverage (MC/DC) are well known coverage criteria. There exist several techniques to measure code coverage. One of these techniques is the insertion of special statements or code snippets into the given source code. We call such a special statement *probe*. If a probe is being reached during the execution of the program, it reports that it is being covered. If probes are distributed strategically over the entire program, it can be analyzed which parts of the program have been executed during a test run. This allows us to say which parts of the program have been *covered*. This is one technique to measure the *code coverage* of a given test suite. What adequacy in respect to a specific coverage criteria means depends on the criteria. For instance, a test suite is considered as adequate in respect to the statement coverage criteria if every statement of the program has been executed at least once during a test run [31]. Coverage criteria can be compared with respect to the difficulty of achieving them. We say that a coverage criterion A subsumes a coverage criterion B if every test suite that satisfies A will also satisfy B. For instance statement coverage subsumes function coverage. If criterion A subsumes criterion B, we say A is more complex than B, or B is simpler than A. The costs for achieving a more complex criterion are higher than the costs for achieving a simpler one. The costs ranges from very low to practically infeasible. Therefore, it is not possible to always apply the most powerful criterion [35]. The results of an empirical study on testing [29] indicated that faults could be detected and removed with the increase of code coverage (block coverage, decision coverage).

1.2 Problem Definition

The goal of this work is to develop approaches to measure the code coverage of a given test suite on an embedded system with very limited resources and tight real-time constraints. Measuring code coverage on such systems is not always straightforward. For instance, embedded systems often have a very limited amount of memory, and a poorly equipped debugging interface. In addition, embedded systems are usually attached to peripherals like antennas or sensors. To communicate with these peripherals, it is necessary to meet certain timing constraints. These timing constraints can be very tight. One violated constraint can lead to a failed test.

There exist some major reasons why it is difficult to measure code coverage on specific embedded systems.

- Measuring code coverage by inserting statements into given source code increases the size of the source code.
- Furthermore, the execution time and the memory consumption of the program are increased due to the reporting of covered probes. It can also happen that the logging of the coverage information is so time-consuming that it is not possible to meet all timing constraints. This can cause an abortion of a test case, which is a serious issue for measuring code coverage. This timing issue will accompany us through this thesis.
- There is also the possibility that a fully instrumented program does not fit on the chip, or that there is not enough memory to store the collected coverage information on a target device.

There are different stages where probes can be inserted into a program. For example, we can instrument the user source code, the object code, or in a compiler specific intermediate representation. If someone wants to instrument the object code or the intermediate representation the compiler can be an additional challenge. For instance, if a compiler provides no extensive debug information it is hard to map collected coverage information to the source code representation. Another point is that compilers usually provide a lot of useful information for collecting code coverage information. In contrast, if a code coverage tool depends heavily on a compiler it may be affected by updates of the compiler.

In this work we focus on resource-limited embedded systems with external peripherals. We want to measure code coverage for programs that are running on such systems. Therefore, we want to measure the adequacy of the test suites of these programs with respect to a specific coverage criterion. We focus on decision coverage and function coverage, but the approach can easily be extended to other coverage metrics. In addition, we focus on programs that are written in the language C. The results have to be represented in a graphical way in order to help a developer to easily identify which parts of the source are covered or not. Another goal of this work is to identify issues that might come up with more sophisticated code coverage criteria in this special context.

The target platform for the experiments in this work is a RISC microcontroller (MRK-IIIe) provided by *NXP Semiconductors*[2]. The resources of the MRK-IIIe applied embedded

[2]http://www.nxp.com/

systems are very limited in terms of memory and the applications running on them are very limited in terms of execution time. In addition, a proprietary compiler is used to build the executables. Embedded systems with the MRK-IIIe are heavily used in industry, for instance, in car keys. These reasons make them a well suited example of embedded systems for which it is difficult to measure code coverage.

1.2.1 Requirements

This section defines the requirements for a code coverage measurement system that is able to overcome the issues in the above problem definition.

Requirement 1 (Language Support)**:** The code coverage tool should work with programs written in the language C. Some embedded devices also support specific language extensions. The system should be able to handle these.

Requirement 2 (Memory Consumption)**:** The code coverage measurement tool should not occupy more than 1 KB of random access memory on the embedded device.

Requirement 3 (Code Coverage Criteria)**:** The code coverage tool should be able to measure at least decision coverage on the applications running on the embedded device.

Requirement 4 (Test Abortion Handling)**:** The code coverage tool should be able to handle test abortions due to the instrumentation overhead.

Requirement 5 (Total Time)**:** The whole measurement should be doable in an overnight task. For test suites that normally need about two hours for a test run. Therefore, the code coverage system should not multiply the test time by a factor greater than seven.

Requirement 6 (Visual Report)**:** At the end of the coverage measurement a visual report should be created. This report should provide information on the source code level. The report should consist of an overview page and a detail page for each source code file. Therefore, it is important that it is possible to map the code coverage information to the source code.

Requirement 7 (Efficient Instrumentation)**:** The code coverage tool should achieve the instrumentation in an efficient way. Means the number of inserted probes should be as low as possible. There are two reasons for that. First, every additional instruction adds an overhead to the execution time. It can be the case that due to this timing overhead certain timing constraints cannot be met. Second, inserting probes can prevent the compiler to perform optimization at certain code parts. Therefore, the code segment can increase for two reasons, the additional instructions from the inserted probes, and the unoptimized regions. It can be the case that the code segment grows too large and cannot be stored on the embedded device. Therefore, the overhead should be kept small.

1.2.2 Assumptions

In this section we give some assumptions that the code coverage system can make on the input source code.

Assumption 1 (Syntactically Correct)**:** We can assume that the source code of the programs that should be tested is syntactically correct.

Assumption 2 (Semantically Correct)**:** We can assume that the source code of the programs that should be tested is semantically correct.

Assumption 3 (Switch is Complete)**:** We can assume that the `switch` statements in the source of the programs that should be tested is *complete*. With *complete* we mean that every defined option of the expression of the `switch` statement is handled in an own `case` and that there exists a `default` case.

1.3 Contribution

In this work we present a method to find and report code coverage information of software running on a resource-limited embedded device during a test suite run. We started with a market analysis to get an overview of the state of the art in the software-development industry. There are many different code coverage tools for C/C++ available. The majority of them are commercial products. We identified requirements that a code coverage tool must have to be applicable to such a specific microcontroller (Section 1.2.1). The majority of the tools do not support embedded targets with such limited resources like the MRK-IIIe.

Secondly, we investigated the possibilities of retrieving code coverage information of a firmware of an embedded system with limited resources under test. There are very different approaches to introduce coverage points into a program. For example, it could be done in the source code, in some intermediate representation from the compiler or in the final binary. Every approach has its advantages and disadvantages. We decided to experiment with the binary and the source code approach. Therefore, we implemented for each of the two approaches a prototype code coverage tool for resource-limited embedded systems.

The first prototype we implemented is based on binary instrumentation. Basically, this approach patches the return instruction of each function with a call to a special function in order to measure the function coverage. This call stores the address of the called function and returns to the next instruction. Since this approach stores the addresses of the called functions sequentially in order of occurrence in the memory, it not only records function coverage but also the execution trace at function level. This approach consumes two bytes of RAM per executed function. This relatively high resource consumption, the technical difficulty to put this approach into practice, and its low flexibility suggest that more advanced coverage criteria can be better measured with source code instrumentation. Therefore, we shifted to focus to the latter approach.

The second prototype, which is based on source code instrumentation, is able to measure decision coverage (also called branch coverage). In order to get a token stream of a given program, which is the output of the lexical analysis of the program, it uses a tool called Coccinelle. First, our tool creates an exact copy of the original source file. Then it prettifies this copy, which means, for instance, adding missing else branches or curly braces. In this prettified version, the tool identifies appropriate locations to insert the probes. Afterwards the cross-compiler of the embedded system builds an executable with the instrumented version of the source code. Then our tool deploys the binary on the target and starts a test suite run. After the run, our tool examines and evaluates the results and updates the instrumentation by removing probes that were already covered. This procedure is repeated until no new coverage information has been collected. This approach only needs one bit of RAM and two instructions in the binary per inserted probe. Furthermore, we implemented a simple add-on that creates a visual report of the results at the end of the procedure. In this visual report, the results of the code coverage analysis can be studied by the developers.

Finally, we tested these two approaches and evaluated the results of these experiments. The results of the source code instrumentation approach show that the overhead in the test suite execution time is acceptable in this industrial case study, which means it is doable in an overnight task. Even in the worst cases our tool needed only about four times longer than the original test suite. The results also show that there is considerable room for improvement.

As far as we know there are no commercial products available that are able to collect code coverage information even from the smallest embedded targets running very time constrained software. The aim of this thesis is to implement a prototypical system, which can handle such resource limited embedded targets. We also want to identify and document the main issues during this development process.

1.4 Structure of this Document

The rest of this document is structured as follows. We start with some preliminaries to give the reader an overview of all concepts that are important to understand this thesis. In the next two chapters we give a detailed explanation of our methods. First, we describe our binary instrumentation approach and afterwards our source code instrumentation approach. Then we discuss our experimental results. After that we present the market analyses and give an overview of related work. Finally we talk about our conclusions and give an outlook on future work. The following paragraphs provide a more detailed outline of the subsequent chapters.

In Chapter 2 we give some definitions of established code coverage or test suite adequacy criteria. We introduce the concept of binary instrumentation as well as source instrumentation. Furthermore we give an overview of embedded systems aspects that are important for this work, like real-time constraints and debugging methods.

In Chapter 3 we explain in detail how our binary instrumentation approach for measuring function coverage works. Furthermore we talk about theoretical advantages and disadvantages of this approach. Additionally we focus on the practical limitations of this approach under our setup.

Chapter 4 focuses on the source code instrumentation approach. It explains in detail how we insert probes into the source code. In addition we present how we evaluate and report the coverage results to the user. Furthermore we provide precise information how we overcome the difficulties of our resource-limited target system. To complete this chapter we discuss our design decisions during the development of our code coverage tool.

Chapter 5 presents, evaluates, and discusses our experimental results. We describe the setup of our experiments and the outcome.

Chapter 6 gives an overview over the work that is related to the work in this thesis. In addition we use this chapter to highlight the differences between our work and the related work in this field. In this chapter we discuss commercial products as well as scientific related work. In a market research we evaluated a series of commercial code coverage tools.

In Chapter 7 we summarize the major facts to recapitulate this thesis. Furthermore this chapter provides suggestions for future work.

Chapter 2

Preliminaries

Contents

2.1 Test Suite Adequacy

Many developers wonder when they can stop testing a program. There are two criteria, or better, rules, that are widely used in practice [33]. The first one is to reserve a certain amount of time for testing during a development process. When the time is over, testing is finished. The second approach is to stop testing at the point where the test suite do not detect new errors. Since the first rule can be satisfied with doing absolutely nothing and the second rule can be satisfied with tests that not test anything, it is obvious that it is hard to derive any information about the quality of a test suite from these rules. Therefore the developer has no awareness how well his test suite performs. Nevertheless, at some point a diligent developer wants to know how well a certain test suite performs.

Let us start with pointing out the ideal solution. A complete test suite would provide a test for every possible input combination of a program. Dijkstra showed in [13] that even for a small program with only two input variables, the number of possible test cases is impractically large. Huang stressed this point in [19] and gave the following example. Let us assume that we run a program on a 32-bit machine. The program takes two integers as input variables. On average this program takes one millisecond for an execution. To test all possible combinations of the two input variables (which are $2^{32} \times 2^{32} = 2^{64}$) our test would take more than 584 million years. Huang also pointed out that this example indicates that no matter how many test cases we generate in a reasonable amount of time, we will always produce just a small subset of the possible test cases for a given program [19]. The idea of structural coverage criteria is that bugs in untested code can never be detected. Therefore, adding test cases until every part of the source code is covered increases the change to find errors in it. However, this does not mean that these parts are free of errors afterwards.

Let us assume that there is a program P and a test suite $T = \{T_1, T_2, \ldots, T_n\}$. All of the tests T are executed against a program P. If a test case reveals incorrect behavior someone fixes this issue and executes the test suite again. This procedure is repeated until the program P produces correct behavior, against every test of T [31].

The fact that a program P produces correct behavior against the stimuli of a given test suite T does not guarantee that there are no errors left in P. After writing an extensive test suite and testing a given program with it, nothing can be said about the thoroughness of the test suite [31]. Therefore, several *test suite adequacy criteria* have been proposed.

Definition 1 (Test Suite Adequacy [31])**.** *A test suite T is considered adequate for a given program P with respect to a criterion C when T satisfies C. In order to determine if T satisfies C depends on the definition of the criterion C.*

Unfortunately an adequate test suite with respect to a specific criterion cannot guarantee an error-free program either. This does not mean that it is not possible to derive useful information from it. In contrast to an adequate test suite, an inadequate test suite with respect to some criterion C definitely indicates that the given program is not tested accurately in respect to C. For example, let us assume that a criterion C states that for every `if` in P, both possible branches, namely the `true`- and `false`-branch, must be executed at least once. If a test suite is adequate with respect to C, one can be sure that all branches are executed during the test run. If not all branches has been exercised, errors that would have been found if all branches has been exercised remain undetected [31].

Measurement of code coverage can be used as a test suite adequacy criterion. In this thesis we focus on control-flow based adequacy criteria like statement coverage, block coverage, condition coverage or decision coverage to name just a few. These criteria will be defined in the following subsections.

In [16] Glass states that in praxis a code coverage percentage of 100 % code is nearly impossible to reach. Therefore, it is unlikely that a test suite is adequate in respect to a specific criterion in praxis. Nevertheless, to compare code coverage criteria it is very useful to have the concept of adequacy.

2.1.1 The Subsume Relation

In the following sections we will introduce some test suite adequacy criteria. All of these criteria help to select a small set of possible paths through a program. Each criterion selects a set of specific program paths from the set of possible paths for a given program. Therefore, it is interesting to compare the different test suite adequacy criteria by their selected subsets of program paths [31].

There are some definitions of the subsumes relation [9], [35], [10], [17]. The following definition in [46] best fits to the above definition of test suite adequacy criteria.

Definition 2 (Subsume relation between adequacy criteria [46]). *Let C_1 and C_2 be two test suite adequacy criteria. C_1 subsumes C_2, written $C_1 \geq C_2$, or $C_2 \leq C_1$, if for all programs P under test and all test sets T, T is adequate according to C_1 implies that T is adequate according to C_2.*

In [35] Ntafos gives a comparison of test suite adequacy criteria like statement coverage, or branch coverage. He analyses the relations of these criteria to each other. He points out that if C_1 subsumes C_1 does not mean that C_1 is better than criterion C_2 for the reason that cost is not considered in the subsume relation.

2.1.2 Statement and Block Coverage

The statement and the block coverage criteria are among the most widely known test suite adequacy criteria [19]. Whereas the statement coverage criteria requires each statement in the program to be executed, the block coverage criteria needs that every *basic block* in a program is exercised [35]. A statement or a block is covered if it has been executed at least once during the test run. In this section we give a detailed description of both criteria. Furthermore, we explain the notion of a basic block.

Definition 3 (Statement Coverage). *In [31] the statement coverage of a test suite T for a program P is calculated with the formula $|S_c|/(|S_a|-|S_u|)$, where S_c is the set of covered statements during the execution of the test suite, S_a is the set of statements in P, and S_u is the set of unreachable statements in P. If the result is 1, T is adequate for program P with respect to the statement coverage criterion.*

We can also find definitions of the statement coverage criterion that are slightly different to the above definition from [31]. Some authors like Myers [33] do not consider the set of unreachable statements in the calculation. This leads to the formula $|S_c|/|S_a|$. Unreachable statements or blocks are statements or blocks that are in an infeasible path. The determination whether a statement is unreachable or not is undecidable. Therefore, it is very hard to reach full coverage in practice. In practice, we speak of a coverage in percent, for instance 85% coverage.

Since the programs under test in this thesis are written in the programming language C, we want to give an rough overview on statements in C. In a procedural programming language like C, a statement specifies an action to be performed. In the C standard there are six types of statements: labeled-statement, compound-statement, expression-statement, selection-statement, iteration-statement and jump-statement [15]. You can find a syntax definition of these statements in Section A. The semicolon is a statement terminator in C [21]. If we talk about an *inner statement* in this thesis, we mean the body of an iteration or a selection statement, like the body of an `if`-branch, `else`-branch, or a `while` loop. Note that there is a little difference between the use of the terms statement in the above definition of code coverage criteria and the use of the term *statement* in C. The difference has mainly to do with the counting, for instance, a compound-statement that consist of three expression-statements. Do we have one, three, or four statements? It depends on how we count. As far as we know this is not uniformly specified. We only count a empty compound-statement as an own statement.

In general, statements are executed sequentially. There are several exceptions for that rule, for example, selection statements like the `if` statement or the `switch` statement. Such statements change the control flow. In order to measure statement coverage, every statement of a program P has to be executed at least once during testing. For a sequence of statements that is not changing the control flow it is not necessary to have coverage information for every statement. It is sufficient to identify sequences of statements that have an unique entry and exit point. Such sequences of statements are called *basic blocks*. A basic block consists of at least one statement. If there is just one statement this statement is simultaneously the entry and the exit point. Selection, iteration, and jump statements are always exit points. Function calls play an ambiguous role in basic blocks. In flow graphs they are usually considered as normal statements. However, in a coverage related context they represent an exit point [31]. In this thesis, we consider function calls as exit points. In summary, the following C statements are exit points in a basic block: `if`, `switch`, `while`, `do`, `for`, `goto`, `continue`, `break`, `return` and statements that are or include function calls.

Definition 4 (Block Coverage)**.** *In [31] the block coverage of a test suite T for a program P is calculated with the formula $|B_c|/(|B_a| - |B_u|)$, where B_c is the set of covered blocks during the execution of the test suite, B_a is the set of blocks in P, and B_u is set of unreachable blocks in P. If the result is 1, T is adequate for program P with respect to the block coverage criterion.*

Similar to the statement coverage formula, some authors do not consider the set of unreachable blocks. This leads to the formula $|B_c|/|B_a|$, which is also more common in practice. A test suite that is adequate for a program P with respect to the block coverage criterion is also adequate with respect to the statement coverage criterion, and vice versa.

In practice it is very common that a test suite has to produce at least 85% coverage with respect to specific coverage criteria [30]. According to Marick the value 85% is more or less a practically developed value with less scientific basis. Marick [30] stated that one reason for this is that people use this number because a lot of respectable companies use this number. Marick interviewed some of those respectable companies. According to these interviews they took this number because a division with a high reputation inside their company is using this number.

2.1.3 Function Coverage

Function coverage is a very coarse-grained test suite adequacy criterion. It can be seen as a simplified version of the statement coverage criterion: a function is covered if the first statement of the function is covered. More formally, function coverage can be defined as follows.

Definition 5 (Function Coverage)**.** *The function coverage of a test suite T for a program P is calculated with the formula $|F_c|/(|F_a| - |F_u|)$, where F_c is the set of covered functions during the execution of the test suite, F_a is the set of all functions in P, and F_u is the of unreachable functions in P. If the result is 1, T is adequate for program P with respect to the function coverage criterion.*

2.1.4 Conditions and Decisions

Any expression that evaluates to `true` or `false` constitutes a *condition* [31], which is also called *predicate*. Let us assume that `A` and `B` are Boolean variables and `x` and `y` are integers. The following expressions are conditions, `A`, `x < y`, `(A && B)`, `(A || (x > y))`. Note that these expressions are in C syntax and that in C, `x + y` or `x` are also valid conditions and the constant values `1` and `0` correspond to, respectively, `true` and `false`.

A condition can be *simple* or *compound* [31]. A simple condition does no use all logical operators `and`, `or`, and `!`. It only uses the unary negation operator `!`. It consists of variables and at most one relational operator from the set (`<, >, <=, >=, ==, !=`). If a condition is not simple it is a compound one. Of the examples above `A`, `x < y`, `x + y`, and `x` are simple conditions. In this thesis, we will use the term *condition* to mean any simple or compound condition.

Any condition in a program can be seen as a *decision* in an appropriate context [31]. Most of the high level languages provide special statements to give contexts for decisions. In C, these statements are the selection and the iteration statements. It has to be noted that there is a third possible outcome of an evaluation of a decision. It could be the case, that the result is `undefined`. For instance, if the function `foo` never returns, the decision of `if(x < y && foo(y))` cannot be evaluated [31].

2.1.4.1 Lazy Evaluation

Lazy evaluation is a method to shorten the evaluation of a compound condition. It is also called *short-circuit evaluation [31].* The C standard [15] requires lazy evaluation for the operators logical AND `&&`, logical OR `||`, and the conditional operator `?`.

Consider the compound condition `(A OP B)`, whereas are `A` is a simple condition and `B` is a simple or a compound condition. In the case of the logical AND (`OP` = `&&`), lazy evaluation means that if the first operand evaluates to `false`, the second operand is not evaluated. In the case of the logical OR (`OP` = `||`), lazy evaluation means that if the first operand evaluates to `true`, the second operand is not evaluated. The lazy evaluation of the conditional operator is not interesting for this thesis.

2.1.4.2 Infeasibility

It can be infeasible to cover a decision or a condition. Consider the example in listing 2.1. For the reason that the value of `x` cannot be greater than 20 and smaller than 10 at the same time, it is infeasible to cover the condition `x < 10` in line 3. Since the two simple conditions in line 3 are connected with a logical AND the decision in this line is also infeasible to cover.

Listing 2.1: Example of an infeasible decision.

```
if(x > 20 && y > 10) {
  s1 = bar1(x, y);
  if(x < 10 && y > 30) {
    s2 = bar2(x, y);
  }
}
```

2.1.5 Decision Coverage

Test cases that are adequate with respect to the statement and block coverage criterion do not necessarily force a decision inside an `if` statement to evaluate to `true` and `false`. The test suite

$$T = \{t_1 : \langle a = -3\rangle, t_2 : \langle a = -1\rangle\}$$

for the program in Listing 2.2 is adequate with respect to the statement coverage criterion. Nevertheless, it is obvious that possible execution paths are not traversed. For instance, the decision `a < 0` in the `if` is not evaluated to `false` with the given test suite T. Therefore, several errors might stay undetected. A stronger test suite adequacy criterion is be necessary to ensure that a decision is forced to evaluate to `true` as well as to `false`.

Listing 2.2: Function that returns the absolute value of a given integer.

```
int abs(int a)
{
  if(a < 0)
  {
    a = -a;
  }
  return a;
}
```

A coverage criterion that requires the test suite to execute every decision branch of a program under test is the *decision coverage* criterion, sometimes named *branch coverage* or *branch decision coverage*. It states that a decision is covered if it evaluates at least once to `true` and `false` during the run of the test suite. The decision coverage criterion requires that the decision of every selection or iteration statement of a given program P is covered.

Definition 6 (Decision Coverage). *In [31] the decision coverage of a test suite T for a program P is calculated with the formula $|D_c|/(|D_a| - |D_u|)$, where D_c is the set of covered* *decisions during the execution of T, D_a is the set of decisions in P, and D_u is the set of infeasible decisions in P. If the result is 1, T is adequate for program P with respect to the decision coverage criterion.*

Similar to the statement coverage formula, some authors do not consider the number of unreachable decisions. This leads to the formula$|D_c|/|D_a|$. Usually, all statements in a program P are covered if all decisions of P are covered. Unfortunately, there are at least two exceptions for that [33].

- P has no decisions. Here the empty test suite would be decision adequate but not statement adequate.
- P has multiple entry points. Here a statement might be only executed if P starts at a specific entry point. Then a decision adequate test suite would be not statement adequate.

Hence, the decision coverage criterion does not subsume the block or the statement coverage criterion.

In C, the decision of a selection or iteration statement is called *controlling expression*, which should have a scalar type [15], for instance

`if ( `*expression*` ) `*statement*$_1$` else `*statement*$_2$.

According to the C99 standard in [15] arithmetic types and pointer types are collectively called scalar types. With the help of the controlling expression the `if` statement can make the *decision* whether the `true` or the `else` branch should be executed. If the *expression* in the `if` statement is no 0 *statement*$_1$ is executed, otherwise *statement*$_2$. The expression can be a simple or a compound condition which evaluates to `true` or `false`.

2.1.6 Condition Coverage

The outcome of a decision depends on its condition. A simple condition is considered covered if it has evaluated to both values (`true` and `false`) during the test run. Therefore, all simple condition decisions are covered during a test run that achieves decision coverage. A compound condition is considered as covered if every simple condition in the compound condition has exercised to both values by the test suite during the test run.

	`(x < 5)`	`(y < 3)`	`((x < 5) || (y < 3))`
t_1:	`true`	`true`	`true`
t_2:	`true`	`false`	`true`
t_3:	`false`	`true`	`true`

Table 2.1: Evaluation outcome for all conditions for every test in T.

Hence, decision coverage is not necessarily strong enough to cover all simple conditions within a compound condition.

Definition 7 (Condition Coverage)**.** *In [31] the condition coverage of a test suite T for a program P is calculated with the formula $|C_c|/(|C_a| - |C_u|)$, where C_c is the set of covered simple conditions during the execution of T, C_a is the set of all simple conditions in P, and D_u is set of infeasible simple conditions in P. If the result is* 1*, T is adequate for program P with respect to the condition coverage criterion.*

Similar to the statement coverage formula, some authors do not consider the number of infeasible conditions. This leads to the formula $|C_c|/|C_a|$.

It could be the case that a condition is covered and the dependent decision is not. Consider a test set T designed to test the simple program `if((x < 5) || (y < 3))`:

$$T = \{t_1 : \langle x = -3, y = -2\rangle, t_2 : \langle x = -3, y = 4\rangle, t_3 : \langle x = 6, y = -2\rangle\}.$$

T is adequate with respect to the condition coverage criterion. Table 2.1 shows the evaluation outcome for all conditions for every test in T. As it can be seen, for every t in T the decision evaluates to `true`. Therefore T, is not a decision coverage adequate test set for this simple program. For a programming language that uses lazy evaluation this is not true.

Proposition 1 (Condition Coverage and Lazy Evaluation)**.** *If a test suite T for a program P written in a language that uses lazy evaluation is adequate with respect to condition coverage it is adequate with respect to decision coverage.*

Proof. On the one hand, lazy evaluation ensures that a simple condition is only evaluated if the outcome of the expression can influence the outcome of its decision. On the other hand, the condition coverage criterion requires that every simple condition within a decision is evaluated to `true` and `false` at least once. Based on these observations, we can prove Proposition 1 by considering only the last simple condition A of a decision $\Phi(n)$. There are two possible cases to connect A with $\Phi(n-1)$, with $\wedge$ or $\vee$.

$$\Phi(n) = \Phi(n-1) \wedge A$$

$$\Phi(n) = \Phi(n-1) \vee A$$

In the $\wedge$-case, $\Phi(n-1)$ has to be `true`, otherwise A is not evaluated. Due to the condition coverage criteria A has to evaluate to `true` and `false`. If A evaluates to `true` the decision is `true`. If A evaluates to `false` the decision is `false`. In the $\vee$-case, $\Phi(n-1)$ has to be `false`, otherwise A is not evaluated. Due to the condition coverage criteria A has to evaluate to `true` and `false`. If A evaluates to `true` the decision is `true`. If A evaluates to `false` the decision is `false`. Therefore, the fact that the last simple condition evaluates to both `true` and `false` implies that decision coverage will be achieved. □

2.1.7 Condition/Decision Coverage

For languages that do no use lazy evaluation the decision coverage criterion and the condition coverage criterion are incomplete. A test suite that is adequate with respect to decision coverage achieves every possible outcome of all decisions in a program. In contrast to the condition coverage criterion, decision coverage does not imply that every simple condition within a compound condition has evaluated to both values (`true` and `false`). With a test suite that is adequate with respect to condition coverage criterion we can have the exact opposite problem, since this criterion ensures that every possible outcome of every simple condition within a compound one has been taken. This does not imply that every possible outcome of the decision has been exercised [31].

Since neither condition coverage nor decision coverage subsumes the other one, a stronger test suite adequate criteria named condition/decision coverage has been proposed. This test suite adequate criterion combines the strengths and overcomes the limitations of using one of the two coverage criteria in isolation. The condition/decision coverage is also known as *branch condition coverage* [31].

Definition 8 (Condition/Decision Coverage)**.** *In [31] the condition/decision coverage of a test suite T with respect to a program P is computed as $(|C_c|+|D_c|)/(|C_a|-|C_u|+|D_a|-|D_u|)$, where C_c is the set of covered simple conditions, D_c the set of covered decisions, C_a and D_a the sets of simple conditions and decisions, respectively, and C_u and D_u the set of infeasible simple conditions and decisions, respectively. T is adequate with respect to the condition/decision coverage criterion if the condition/decision coverage of T with respect to P is* 1.

Similar to the other coverage formulas, some authors do not consider the number of infeasible conditions and decisions. This leads to the formula $(|C_c| + |D_c|)/(|C_a| + |D_a|)$. The condition/decision coverage criterion subsumes the condition and the decision coverage criterion.

2.1.8 Multiple Condition Coverage

In Section 2.1.6 we stated that a test suite T that is adequate with respect to the condition coverage criterion ensures that every simple condition within a compound condition has taken both values (`true` and `false`). *Multiple condition coverage* ensures that every possible combination of truth values of each simple condition within a compound condition has been taken during the execution of the test suite. This corresponds to examining all lines of the truth table of the compound condition. If we need one test case per combination and C is the number of simple conditions within a compound condition, we need 2^C test cases for each compound condition. In an avionic application it can be the case that a compound condition consists of $C = 32$ conditions, which would lead to over four billion test cases for only this compound condition. If we assume that the execution of one test case take 1 millisecond, the execution of all 4 billion test cases would take over 49 days [31]. For programming languages that are using lazy evaluation the set of test cases is significantly smaller than for languages that are not using lazy evaluation [20]. The multiple condition coverage criterion subsumes the modified condition/decision, the condition/decision, the condition, and decision coverage criteria.

2.1.9 Modified Condition/Decision Coverage

As described in the previous section, multiple condition coverage is very expensive if decisions consist of many conditions. Therefore, a weaker adequacy criterion has been proposed, namely *modified condition/decision coverage* (MC/DC). In [7] Chilenski and Miller describe MC/DC as a coverage criterion that requires that it has been shown by execution that each condition within a condition independently and correctly affects the outcome of the decision. Chilenski and Miller also stated that this criterion has been developed to help meet the need for extensive testing of complex Boolean expressions in safety-critical applications.

The author of [31] defines MC/DC as follows. A test set T for a given program P is considered adequate with respect to the modified condition/decision coverage criterion if after the execution of the test set, the following four requirements are fulfilled [31]:

- T is adequate with respect to the block coverage criterion,
- T is adequate with respect to the condition coverage criterion,
- T is adequate with respect to the decision coverage criterion, and
- "Each simple condition within a compound condition C in P has been shown to independently affect the outcome of C [31]."

As for the the multiple condition coverage criterion the set of test cases is significantly smaller if a programming language with lazy evaluation is used [20]. The modified condition/decision coverage criterion subsumes the the condition/decision, the condition, and decision coverage criteria. For further details on this criterion please refer to [31].

2.2 Instrumentation

2.2.1 Overview

Measuring code coverage requires information that can only be collected during the execution of the program under test. Therefore, several intrusive, non-intrusive, and combined methods are available. The term *non-intrusive* methods usually refers to techniques like program counter logging, also called tracing. Such methods usually need Hardware Performance Monitoring (HPM) support [39]. This is provided by special units in certain processors like the Intel Itanium 2 processor [12] or processors with the Embedded Trace Macrocell from ARM [26], to name just two examples. Another non-intrusive method is emulating the target device in a virtual machine [6]. In such an environment, a lot of information can be easily collected. Usually non-intrusive methods like program counter logging collect information in the object code. One problem with non-intrusive methods is the mapping of the results from the object code into the source code. This strongly depends on the debug information provided by the compiler. If the original source code decisions cannot be fully recovered from the provided debug information, it is not possible to map the covered objects to the source code [6].

Whereas non-intrusive methods like program counter logging usually need read access to several cpu registers or a sophisticated debugging interface, *intrusive* methods can normally be applied on less highly developed environments. Therefore, intrusive methods have to be used in less sophisticated environments to gather the necessary information.

A very frequently used intrusive method is *instrumentation*. The authors in [3] define a probe as an additional program code that does not change the functional behavior of the program but collects some additional information. A probe, also known as point or marker, can be inserted at any step between source code and executable. This can be done directly in the source code (*source code instrumentation*) or in the compiled executable (*binary instrumentation*). A less common way is to insert these points in the compiler's intermediate language.

Probes can negatively affect a program in several ways [3]. A very important point to mention in this context is that a probe can affect the timing in a real-time system. Inserting probes into a program also increases the size of a program. This is usually not a problem with state-of-the-art desktop machines. However, for an embedded system, this can be a serious issue. Furthermore, additional memory is needed during the execution of an instrumented program for storing visited probes.

There is also a considerable difference between just reporting code coverage in contrast to do profiling or program tracing. The reason for this is, that for code coverage it is just necessary to track whether a probe has been covered during the test or not. On the other hand, for profiling or program tracing it is necessary to store the number of executions for each probe or the order of executions of the points respectively.

It has to be noted that the applied test suite adequacy criterion has a tremendous impact on the points above. For instance, decision coverage needs much less probes than condition/decision coverage. In addition, for condition/decision coverage, the expressions of the program have to be somehow rewritten, whereas for decision coverage just some points have to be inserted.

2.2.2 Binary Instrumentation

Binary instrumentation or object code instrumentation is the process of inserting probes in the executable binary of a program [23]. In other words, it is instrumentation on the machine-code level, for instance, manipulating assembly code in an ELF (Executable and Linkable Format) file originally written in C [23]. Another example is the insertion of probes in Java bytecode [31]. This can be done before and after the linking phase.

There is one big drawback with binary instrumentation for code coverage: It can be difficult to map a covered probe back to the source code statement [6]. This is going to be harder if the compiler uses optimizations. It can be the case that some statements that should be covered are not present in the optimized executable. Furthermore, it can be that there is no or insufficient debug information available in the binary. This makes it very hard to connect the coverage results with the source code.

On the other hand, there are some advantages for code coverage on the machine-code level. An advantage within binary instrumentation is that machine-code is very easy to parse. For instance, assembler instructions for comparing or jumping are very easy to find. These are very common points for instrumentation. Another advantage is that differences between the version of the binary instrumented executable and the original executable is smaller than the differences between the original version and the executable from source code instrumentation [6]. Another drawback is that instrumentation depends on the instruction set of the target architectures.

2.2.3 Source Code Instrumentation

Source code instrumentation is the process of inserting probes directly in the source code of a program. In the case of the language C, this can be, for instance, a macro, a function call, or another statement. It is instrumentation before the compiling phase.

Of course this technique has advantages as well as disadvantages. Let us start with the advantages. A major benefit with this method is that information about covered probes can be easily mapped to the original source code [31]. Another advantage of source code instrumentation is that inserting probes is much simpler and more understandable than inserting something in the object code and map it back to the source level.

Now let us come to the disadvantages. Source code instrumentation can affect the compiler optimization and therefore increases the size of the executable more than binary instrumentation. The reason for this is that an inserted point can prevent an optimization by the compiler in the area of code around the inserted point. Another drawback is the need for parsing the source code. Parsing the source code is very important to find the right locations for inserting probes in the code. Unfortunately, writing a parser for the language C is not a trivial task. In contrast, if the source code is correctly parsed into a good representation, it is very easy to instrument. If parts of the source code are not available, or it is not possible to recompile parts of the program, this technique is not applicable [31].

2.3 Embedded Systems

2.3.1 What is an embedded System?

In the literature, many authors have proposed definitions for the term *embedded systems*. Almost every author somehow mentions that the proposed definition is not the end all solution. It is possible to provide a definition for a specific time in the past. However, with the fast development of hardware and the decreasing costs these definitions do not hold for long. The author in [5] gives the following definition. *"An embedded system is a combination of computer hardware and software - and perhaps additional parts, either mechanical or electronic - designed to perform a dedicated function."* He also stated that the design of an embedded system is to perform a dedicated task, whereas a personal computer is not designed to perform a specific task. In [34] Noergaard gives an similar definition.

Whereas the differentiation from a *general-purpose* computer system is a frequently used part in an embedded system definition there are examples that weaken this circumstance. For instance, with the rise of PDAs (Personal Digital Assistance) we got "embedded systems" with the possibility to provide many functionalities. Furthermore, today we use smart phones that are better equipped than state-of-the-art PCs some years ago. Therefore, the definitions have been weakened due to new technical developments.

In contrast to other authors, Heath stated in [18] that the best way to define what an embedded system is, is to describe it in terms of what it is *not* and give examples of how it is used. Furthermore, he stresses that an embedded system is not designed to be programmed by the user.

As it can be seen, it is not easy to give a one-sentence-straight-to-the-point definition for embedded systems. However, the embedded system on which we focus in this thesis is definitely not in the group of ambiguous systems.

2.3.2 Peripherals

With peripherals an embedded system can communicate with the external world. Generally, a peripheral is everything that is not on the processor. They exist in all shapes and sizes. The range is practically endless. For the reason that the processor has direct access to the memory, the memory hardly counts as a peripheral [43]. Some peripherals are very specific and just used in a small domain of embedded devices, whereas others, like timers or serial ports, are widely used [5]. The majority of the commonly used peripherals are located on the same chip as the processor [5] . These devices are called *internal*, or *on-chip* peripherals. Devices that are not located on the same chip as the processor are called *external* peripherals.

It is also common practice that peripherals a directly connected to interrupt pins of a processor or with the interrupt controlling unit to signal the processor an event [5] .

2.3.3 Real-Time Systems

A subgroup of embedded systems are *real-time* systems. A system is considered as a real-time system if it processes external stimuli in a limited amount of time [5]. For a real time system, a late response is as bad as a wrong one. In other words, real time systems have computations with a *deadline*. To meet these deadlines protocols with tight timing constraints are used [5]. In summary, timing is a very important factor for this group of embedded devices.

To measure code coverage on real-time systems can be a challenge. The reason for this is that the instrumentation can influence the timing. If a deadline is missed due to the timing overhead of the instrumentation the execution of a test or even the whole test suite can be aborted. This is a serious issue, because if the test suite cannot perform the tests due to the instrumentation, code coverage cannot be measured.

2.3.4 Debugging

Unfortunately, not all embedded systems can be debugged like a normal program on a PC. The reason for that is that the majority of embedded systems do not provide resources to be debugged directly on the system itself. To debug an embedded system, a cross-debugger is needed. This cross-debugger runs on a “host” PC and communicates with a debug interface. This debug interface is usually called JTAG even if it has not implemented this standard [43].

Debugging requires some processor resources and can therefore influence the performance of the embedded device during debugging. This can change the timing of the execution. Consequently, this can lead to bugs that only occur during debugging. Another issue is that, the more debugging features a processor supports, the more resources are needed. This increases the costs for the processor. In order to keep the costs low, in a lot of processors just a subset of the possible debugging features are implemented. [43].

Chapter 3

Binary Approach

Contents

3.1 Overview

Before we started with this thesis we discussed ideas how to measure code coverage on embedded targets with very limited resources. Soon, we narrowed the selection to two approaches, a binary instrumentation approach and a source code instrumentation approach. First, we decided for the binary approach because we wanted to avoid to parse C. After a first evaluation of our binary approach, which focused on function coverage, we decided to change our strategy to our source code based approach (see Chapter 4), which is able to measure a combination of function coverage and decision coverage.

The rest of this chapter is structured into three sections. In Section 3.2, we discuss the idea of our binary instrumentation approach. Section 3.3 presents the proof of concept implementation of this approach. In Section 3.4 we discuss the restrictions. Furthermore, we will explain why we changed our strategy to the source code instrumentation approach.

3.2 Idea of the Approach

Before we present our approach, we want to point out two main difficulties of binary instrumentation for code coverage. First, inserting additional instructions at a point shifts the subsequent instructions. This changes the addresses of the subsequent instructions. Therefore, also the references to these instructions have to be changed. Note that this is only an issue if we modify the machine code. Second, to find the associated source line for all instructions is not an easy task. If it is possible or not, depends on the provided debug information. Therefore, it is not trivial to map the coverage information to the source code if it has been instrumented at binary level.

The first idea for instrumenting the binary was to start with a minimal invasive approach. Note that this approach focuses on function coverage. We had two reasons for that. The first reason was that we searched for a starting point to get into the whole task. The second reason was that we did not want to be confronted with all the difficulties simultaneously. Therefore, we only wanted to overwrite instructions and not to insert new ones in order to avoid relocations. We started to look for possibilities to instrument the binary only with overwriting existing instructions. During this research we came up with the following idea.

Cover all *return from subroutine* instructions to obtain function coverage information. Before we start to explain our approach in more details, we have to introduce three instructions.

- Instruction `RET` (*return from subroutine*) performs a return from a subroutine. Therefore, it pops the return address from the current stack and then the execution continuous from this address.

- Instruction `USR` is a software interrupt with a context switch to user mode. It pushes the return address (PC+1) and the program status word on the current stack. Then it sets the PC to the interrupt vector and the code executions continues from there.

- Instruction `RETI` (*return from interrupt*) performs a return from an interrupt service routine. Therefore, it pops the program status word from the stack and restores it. Afterwards it pops the return address from the stack and the executions continuous from there.

First we rewrite all `RET` instructions in the ELF[1] executable binary file with a software interrupt user call `USR`. The user call jumps to a special function called `covreturn` (see Section 3.3.5). This function copies the return address of the user call into an array reserved for coverage data (`cov_array`) and jumps back to the return address of the function which belongs to that patched `RET` instruction. The stored address is the address after the address of the caller instruction. Hence, it is easy to calculate which function we covered. It is the function that is called in the address before. After the execution of the software that should be tested, we extract the coverage array from the embedded device and create a report.

3.3 Implementation

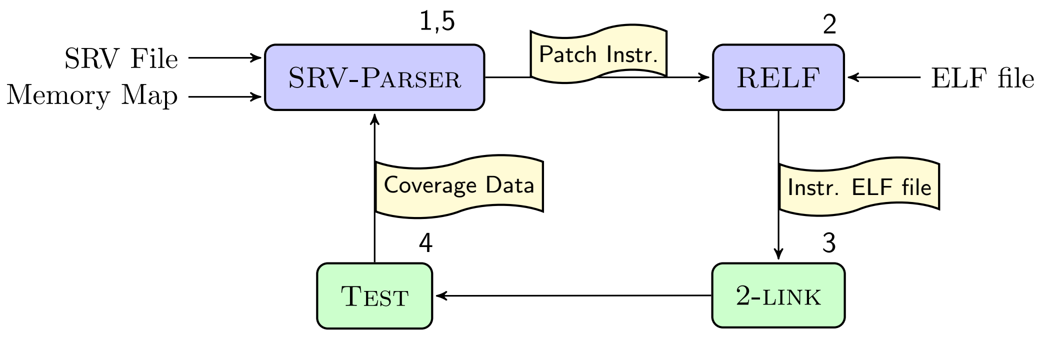


Figure 3.1: A schematic overview of the structure of our proof of concept implementation.

In order to test this approach we did a proof of concept implementation, which we discuss in this Section. Figure 3.1 gives a schematic overview of the implementation. The SRV-Parser and RELF, presented with blue rectangles, are implemented by us and the `Test` and `2-link` are specific third party tools of the embebbed target.

3.3.1 SRV-Parser

The SRV-Parser has three purposes. First, it extracts the instructions that should be patched from a SRV file and stores the source references, i.e. it stores which `RET` belongs to which function. The SRV file is an intermediate output of the compiler and holds the instructions names, the associated addresses, and the source references. Furthermore, it

[1]Executable and Linkable Format [11]

```
assembly void covreturn(void) property(isr) clobbers(R4,R6)
{
  asm_begin
    .undef global data cov_array
    .undef global data cov_ptr
  asm_text
                      // R7 is the stack pointer (SP)
  PUSH R4,R6          // save registers
  MOV R4,@R7+8        // copy return address to register
  MOV.s R6,cov_pter   // move cov_ptr into R6
  MOV @R6,R4          // copy the return address into cov_array
  ADD.w cov_ptr,#2    // increment cov_ptr
  MOV R4,@R7+4        // prepare Frame for RETI (step 1)
  MOV @R7+6,R4        // prepare Frame for RETI (step 2)
  POP R4,R6           // restore registers
  ADD R7,#2           // set SP, discard unused value
  RETI                // returns to the return address of
                      // the to be covered function
asm_end
}
```

Listing 3.1: Coverage function of the proof of concept implementation.

generates a patch instruction file with the offsets to these instructions. These are the offsets in bytes from the start of the instructions in the ELF file to the instructions that should be patched. Second, it extracts the addresses of the `cov_array` and the `cov_ptr`. This is important for the extraction of the coverage information after testing. Third, after the test it receives the extracted coverage date and generates a report.

3.3.2 RELF

RELF (`RET` instrcution patcher for ELF files) is the tool that locates the instructions of the patch instruction file in the ELF file and overwrites it with a `USR` to the function `covreturn`. Before patching an instruction it has to find the beginning of the code segment in the ELF file. The output of this tool is an instrumented ELF file.

3.3.3 2-link

The *2-link debug probe* is the debug interface to the MRK-IIIe. It can be connected to a host machine with a USB cable. The 2-link can be addressed over an API. There is also a command line interface for this API. The 2-link provides the functionality to set or read registers or memory locations, set the program counter or breakpoints, and start or stop the execution. Furthermore, binaries can be loaded onto the embedded target with this tool. This is the only way to debug the target. In this system we mainly use the 2-Link to deploy a binary to the embedded system and extract the results from it.

3.3.4 Test

This component is not really a component in the usual sense. For this approach we had no test suite. We tested a demo applications without external test stimuli as input.

3.3.5 Coverage Function

The `USR` jumps to a special function `covreturn` (Listing 3.1). This functions is supposed to store the return address of the function that should be covered. In order to do this it performs five steps.

1. First it saves the values of the registers R4 and R6 onto the stack (line 8, 9).
2. In lines 10 to 13 the function stores the return address of the function that should be covered in the coverage array. The `cov_ptr` points to next free field in the array. Therefore, it has to be incremented at the end (line 13).
3. In line 14 and 15 the functions prepare the stack frame for the `RETI`. Therefore, it rewrites the return address of the interrupt subroutine with the program status word.
4. Then it restores the registers R4 and R6 to their original values (line 15).
5. In line 16 we set the stack pointer to the prepared stack frame for the `RETI` and in line 17 we can perform the `RETI`. It pops the program status word from the stack and then it pops the return address of the to be covered function from the stack and the execution continuous from there.

Note that the instruction syntax is from right to left, the left operand is the destination and the right operand is the source. Furthermore, the register R7 in the Listing 3.1 is the stack pointer (SP).

3.3.6 Script

A control script manages the communications between the tools. As you can see in Figure 3.1, five steps are necessary to measure function coverage.

1. The SRV-Parser receives a SRV file and a memory map and creates a file with a list of all instructions that should be patched with a `USR`.
2. The RELF receives the patch instructions file from the SRV-Paser and the ELF file. It overwrites all instructions of the patch instruction file with a USR and returns an instrumented version of the ELF file.
3. The script generates a runnable binary from the ELF file and deploys it with the help of the 2-link.
4. Then the scripts starts the execution of the demo application. After the execution it extracts the coverage information from the embedded device.
5. The SRV-Parser receives the coverage data and creates a report.

3.4 Restrictions and Change of Strategy

Unfortunately, this approach fulfills only a view of the requirements from Section 1.2.1.

R1 (Language Support) Is fullfilled.

R2 (Memory Consumption) The approach did not consume more than 1 KB, but we tested it only with demo applications. Since we store the addresses one after another into an char array and do not prevent overflows other applications can produce an overflow. However, for this proof of concept implementation this was acceptable.

R3 (Code Coverage Criteria) Is not fulfilled. For a stronger coverage criterion like decision coverage we have to insert new instructions. Therefore, we would need relocations.

R4 (Test Abortion Handling) Since we had no test suite, we cannot really say something about this requirement.

R6 (Total Time) Since we had no test suite, we cannot really say something about this requirement.

R5 (Visual Report) Is not fulfilled. With the available debug information it is hard to map the coverage data to the source, which is important for this requirement.

R7 (Efficient Instrumentation) We need one probe per `RET` instruction. Hence, it fulfills this requirement.

Besides these shortcomings regarding the requirements the approach has the restriction that it is not possible to cover `inline` functions or tail calls. The issue with the `inline` functions is that there is no debug information left that can give us a hint that a statement belongs to a called `inline` function. This is a compiler specific issue. Furthermore, there is no `RET` instruction that we can patch. Another issues is that in the case of a tail call, we cover only the *callee* function and not the *caller* function of the tail call. This is also a compiler specific issue. With the compiler of our test embedded system it was possible to turn this tail call optimization off. Nevertheless, it can be a restriction with other systems. With this facts in mind we decided to change the focus on the source code instrumentation approach and see where there are the problems.

Chapter 4

Source Code Instrumentation Approach

Contents

4.1 Overview

In order to achieve code coverage measurement of a resource-limited embedded device with source code instrumentation we identified three major points to address.

1. **The Instrumentation of the Source Code**
   In order to achieve this, we have to solve at least three subpoints. First, the navigation inside of the source code. We need this for the third point. Second, we have to ensure the uniformity of the source code. It will be easier to instrument the source if statements like `if` or `for` can always be processed in the same way. Finally, we need to find the right locations inside of the source code to place a probe. In order to measure a specific code coverage metric it is important to insert probes at the right locations.

2. **The Reporting Mechanism During Runtime**
   How can we store the information that a probe has been covered?

3. **The Handling of Timing Issues**
   What can be done if the test aborts due to some timing issues caused by the instrumentation?

The rest of this chapter is divided into two main sections and a summary. In Section 4.2, we discuss the above described issues in detail and present our approaches to solve them. Section 4.3 presents the prototypical implementation of our source code instrumentation approach.

4.2 Issues and Approaches

4.2.1 Instrumentation

In order to instrument source code written in C, it is necessary to navigate through code and to insert probes at the right locations. Therefore, at least three major issues has to be solved. The first issue is that we cannot navigate through the source code without parsing it. Since code in C is not easy to parse, this is not a trivial task. The second issue is that important statements for the instrumentation like an `if` can occur in different ways, for instance with our without curly braces. Maybe the `if` has an associated `else` branch. There can be sequences like `if-else-if` etc. Therefore we want standardize every occurrence of such a statement. We call this process of standardizing from now on *prettifying*. The last issue is the appropriate placement of the probes in order to measure a specific code coverage criterion.

For this work we reduced the complexity of the task by disregarding preprocessor statements. This means that we instrument the user source code and not the preprocessed version of the user source code. We are aware that this might distort our results a little bit because we might instrument switched off code snippets.

The appropriate placement of the probes will be much more easier if we do not have to deal with the first two issues. For that reason our approach for instrumenting the source code works as follows. First, we provide the functionality to navigate through the source code. Second, we prettify the source code. Finally, we insert the probes in order to measure a specific code coverage criterion. In the following three subsection we describe how we want to overcome each of these issues.

4.2.1.1 Source Code Navigation

In order to provide the functionality to navigate through the source we have to analyze it. The source code can be seen as a stream of characters. The analysis part of a compiler is doing such an analysis of the source code of a program.

The analysis part of a classical compiler consists of three phases or components [2]. The first component that is doing the *lexical analysis* is called *scanner*. It reads in the source code and groups the characters into meaningful fractions, called *tokens*. The output of the scanner is called a *token stream*. The second component is the *parser* and it performs the *syntax analysis*. It uses the tokens from the token stream to produce a *syntax tree*, which is a representation that characterize the grammatical structure of the token stream. It uses the information from the second phase to check the source code for semantic consistency with the language definition [2].

To navigate to locations that have to be instrumented for decision coverage we need the token stream from the lexical analysis. Having the syntax tree from the parser would be good but is not absolutely necessary. It should be possible to navigate to the appropriate positions in the token stream with some easy rules. In summary, we need at least a token stream and a reduced set of syntax rules to navigate inside of the token stream.

Let us assume we have a tool that provides a token stream, then we only need to think about how to navigate to the right tokens. In order to prettify or to instrument the source code we need to navigate to the beginning and the end of the inner statements of selection and iteration statements. This can be done with a set of recursive functions. The idea is to iterate through the token stream until we find a selection or iteration statement. Then call the function that handles this statement, for instance `handleIf()`. This function navigates to the beginning of the inner statement and iterates through the tokens until it reaches the end of the inner statement or it finds a selection or an iteration statement. If the latter is the case, it calls the function which handles this statement.

With this navigation functionality we can identify all locations that we need to prettify or instrument the source code.

Coccinelle's Lexer Output/Token Stream *Coccinelle* is da program matching and transformation engine by LIP6[1] (Laboratoire d'Informatique de Paris 6). It provides a language called SmPL (Semantic Patch Language) for specifying desired matches and transformation in C code. *Coccinelle* performs a semantic analysis of the given C code.

[1]http://www.lip6.fr

Therefore, it needs to perform lexical analysis. With the command line interface it is possible to return the produced token stream in a file. Listing 4.2 shows such a token stream output of the small code snipped in Listing 4.1. In order to shorten this representation, we disregarded all the tokens associated to any kind of white space.

Each line of the token stream file represents exactly one token. The prefix of each line is `Tag#`, where the `#` denotes the token number. For instance, `Tag80` represents an `if` token, `Tag7` indicates a string token, etc. Every token has a 5-tuple like `("a", 15, 3, 2, "if.c")`. The first value is the printed representation of the token in the source file. The second value is the number of the token, for instance 18th token in this source file. Value number three is the corresponding source code line number. The fourth value is the column number at which the token begins in the source file. The last value is the filename. All the other information in this file is not required for our approach.

```
if(a < 5)
{
  a++;
}
```

Listing 4.1: Simple code snippet for a sample token stream output (if.c).

```
Tag80 (((("if", 0, 1, 0, "if.c")), (0), ((0, 0, 0, 0)), 0))
Tag12 (((("(", 2, 1, 2, "if.c")), (0), ((0, 0, 0, 0)), 0))
Tag8 (("a", ((("a", 3, 1, 3, "if.c")), (0), ((0, 0, 0, 0)), 0)))
Tag38 (((("<", 5, 1, 5, "if.c")), (0), ((0, 0, 0, 0)), 0))
Tag4 ((("5", (0, 2)), ((("5", 7, 1, 7, "if.c")), (0), ((0, 0, 0, 0)))))
Tag13 ((((")", 8, 1, 8, "if.c")), (0), ((0, 0, 0, 0)), 0))
Tag14 (((("{", 11, 2, 0, "if.c")), (0), ((0, 0, 0, 0)), 0))
Tag8 (("a", ((("a", 15, 3, 2, "if.c")), (0), ((0, 0, 0, 0)), 0)))
Tag21 (((("++", 16, 3, 3, "if.c")), (0), ((0, 0, 0, 0)), 0))
Tag30 ((((";", 18, 3, 5, "if.c")), (0), ((0, 0, 0, 0)), 0))
Tag15 (((("}", 20, 4, 0, "if.c")), (0), ((0, 0, 0, 0)), 0))
```

Listing 4.2: Sample token stream output from *Coccinelle*.

4.2.1.2 Prettifying of the Source Code

In this section we address the issue that iteration and selection statements can occur in different ways. In order to instrument these statements it is helpful to prettify them, or in other words to unify or standardize the structure of these statements. The structures should be like in Listing 4.3 and 4.4. At this point, please disregard the *probes* in these two Listings. We will explain them in Section 4.2.1.3.

```
if (expression)
{ probe
  statement;
}
else
{ probe
  statement;
}

while (expression)
{ probe
  statement;
} probe

for (expression)
{ probe
  statement;
} probe
```

Listing 4.3: Instrumented `if`, `while`, `for`.

```
switch (identifier)
{
  case 0: probe
    statement;
    break;

      .
      .
      .

  case n: probe
    statement;
    break;

  default: probe
    statement;
    break;
}
```

Listing 4.4: Instrumented `case` and `default` in a `switch`.

It is important for the instrumentation step that all inner statements of iteration, selection, and `else` statements are compound statements. For example, if we want to instrument a selection statement like `if(a < 3) x = 5;` we have to insert a probe like this `if(a < 3) cover(id); x = 5;`. In this case the inserted probe kicks the statement `x = 5;` out of the `if` branch. In order to prevent this, we need to ensure that every inner statement is a compound statement, which means it has to be surrounded with curly braces. If we also want to instrument the `else` branch in this example, we have to add the whole `else` branch. As already said, this process is called *prettifying*. Since we can assume that a `switch` statement is complete (Assumption 3), which means that there is a `default` case and for each option there is an associated `case`, we are done. With the ability to navigate inside of the token stream, we can identify the beginning and the end of the inner statements of selection and iteration statements. Hence, we only have to check whether the inner statements are already surrounded by curly braces or not. If not we insert them. If there is no associated `else` branch for an `if` statement, we add it including the curly braces. With *Coccinelles's* token stream and the navigation rules from the previous section this is be doable.

A shortcoming of this approach is that we cannot instrument the `do-while` statement. The reason for that is that for decision coverage only inserting probes is not sufficient, because the first execution of the inner statement of a `do-while` statement does not depend on the controlling expression.

Since we can assume that our input programs are syntactically and semantically correct (Assumption 1 and 2), we can be sure that after the prettifying step the selection and iteration statements have exactly the structures as in Listing 4.3 and 4.4.

4.2.1.3 Placement of Probes

Inserting the probes at the right positions in the source code is the basis for measuring a specific code coverage criterion. We want to instrument the source for a combination of the decision coverage criterion and the function coverage criterion. Therefore, we need to insert probes at the following locations:

- At the beginning of every function,
- in the inner statement of every selection and iteration statement.
- after the colon of every `case` or `default`, and
- in the inner statement of every `else` branch.

Listings 4.3 and 4.4 show the placement of the probes in the selection and iteration statements.

If you are able to identify the beginning of a statement, the identification of the beginning of a function in the token stream of a C source code file is not a problem. According to the C Standard [15], the body of a function is a compound statement. Therefore, the beginning of the first function in a source code file is the beginning of the first compound statement in the file. The second body of a function is the next compound function that is not inside of another compound function etc.

Since we can navigate to all locations in the token stream, where we want to insert a probe, we only have to be able to insert the probes. It is possible to specify new token types with special fields to store additional information. Furthermore, new tokens can be easily inserted into this token stream. Hence, our idea is to instrument the token stream and then produce an instrumented version of the source code from this instrumented token stream.

4.2.2 Efficient Reporting of Covered Probes

In this section we address the problem of reporting a covered probe during runtime. The difficulty lies not in the reporting itself, but in doing it efficiently. According to Requirement 2 only up to 1 KB of RAM is available to store the coverage information. Furthermore, the number of used instructions to report a probe should be reduced to a minimum in order to affect the timing as little as possible.

Our approach here is to first solve the memory problem and than optimize the reporting mechanism in terms of minimum number of instructions. Two things should be sufficient to report the probe. First, an array of characters that reserves the memory for the probes. Second, an `inline` function `cover()` that takes the ID of the probe as parameter. The ID should be a unique number for the probe and also the bit index of the coverage bit array, for instance the probe `cover(14)` sets the 14th bit in the coverage bit array if it is executed. The function `cover()` calls a macro that sets the bit with the index ID in the bit array. For instance, let the ID be 29, then the function should set the sixth bit of the character stored at index four in the array of characters. This approach only needs one bit per inserted probe.

4.2.3 Error Handling

This section addresses handling of errors due to timing violations. The execution overhead of reporting covered probes can cause timing violations, which can affect the test result or even lead to an abortion of the test. This is a very serious problem, because if test cases fail to execute we cannot measure their coverage. Requirement 4 addresses this issue and asks for a solution.

Our approach to overcome this difficulty is to remove the covered probes from the instrumentation after a test run and ro run the test suite again. Since a probe can only cause a timing violation if it has been executed. In other words, if a probe affects the timing it must has been covered before. Therefore, we can remove all the covered probes from the instrumentation and do another test suite run. We repeat this procedure of updating the instrumentation and rerunning the test suite until we do not cover new probes or at least all test cases end as expected.

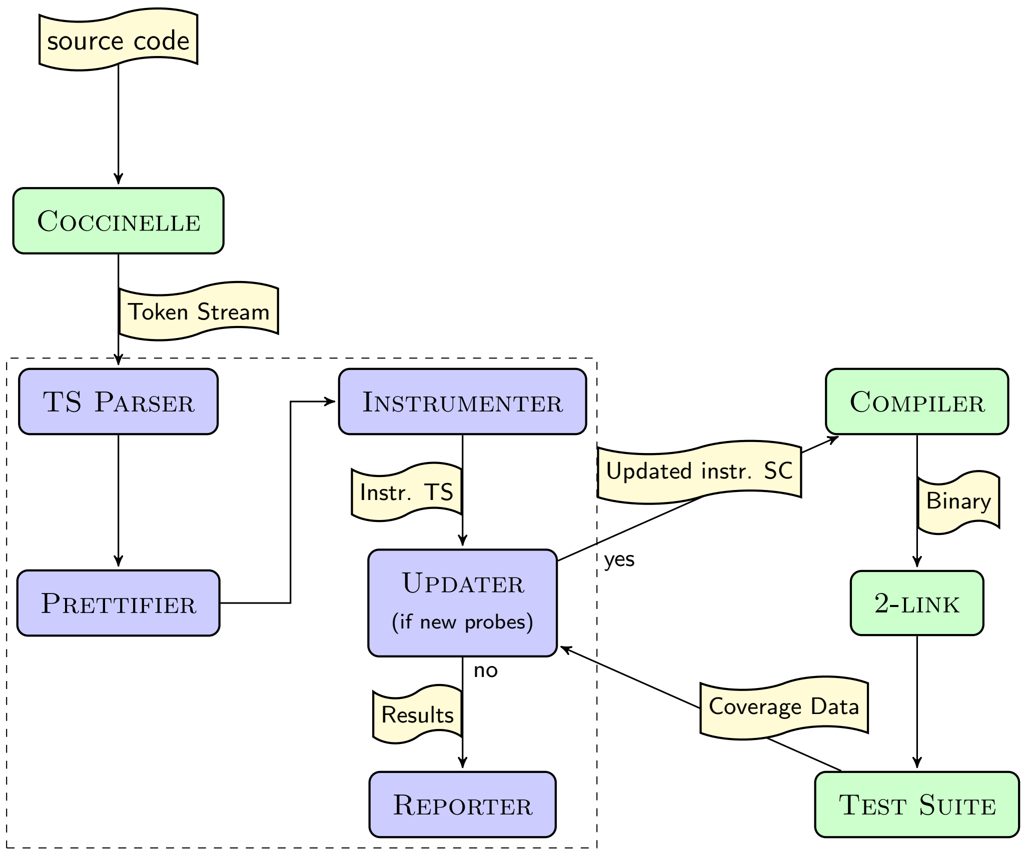


Figure 4.1: A schematic overview of the structure of our code coverage measurement system.

4.3 Implementation

4.3.1 Overview

In this Section we discuss the implementation of our code coverage measurement system, which consists of five different programs. Figure 4.1 gives a schematic overview of the system. All the blue rectangles in the dashed container are modules of our code coverage measurement tool, which from now on is called C-CoCATool (C Code Coverage and Analysis Tool). The four green rectangles represent third party software. The tool Coccinelle has been described in Section 4.2.1.1. It provides the token stream of the source code that should be analyzed. The compiler builds the binary for the embedded system. The *2-link* is the debugging interface of the embedded system and deploys the binary to it. The Test Suite holds all the tests for the embedded system. We modified the Test Suite and give an explanation in Section 4.3.4. A controlling script manages the dataflow and the communication between the five programs. We treat this in detail in Section 4.3.5.

4.3.2 C-CoCATool

The *C-CoCATool* is the heart of our code coverage measurement system. It consists of five parts or modules (the blue rectangles in Figure 4.1). In the following paragraphs we present every part of the *C-CoCATool*.

4.3.2.1 TS-Parser

The TS-Parser parses the tokens from Coccinelle's token stream. In addition to that, it builds a data structure for the later processing of the stream. Therefore, it creates a token instance for each token of the stream. Every instance provides the functions for an easy access to the tokens information.

4.3.2.2 Prettifier

The Prettifier implements the idea described in Section 4.2.1.2. Which means that the Prettifier ensures that every inner statement of a selection or an iteration statement becomes a compound one. In addition to that, it ensures that every `if` gets an associated `else` branch. We implemented this functionality in an recursive approach, which can be seen in Listing B.1 in the Appendix. In order to facilitate the mapping of coverage information to the associate line in the uninstrumented original source code the Prettifier does not add lines during the prettifying process. The code snippet in Figure 4.5 illustrates this. Another important point is, that we prettify the data structure created by the TS-Reader and not the source code itself. Means that we insert new tokens, for instance the curly braces or an `else` token in the data structure and create an instrumented version of the source code from this data structure.

Listing 4.5: Original and prettified version of an `if` statement.

```
if(a < 5)        -->       if(a < 5) {
  a++;           -->         a++ } else { }
```

4.3.2.3 Instrumenter

The Instrumenter receives the prettified data structure and inserts probes as tokens. A probe token consists of the normal 5-tuple with two extra values: an unique identifier, and a boolean field *covered*. The boolean field stores whether the probe is already covered or not. A probe will only appear in the instrumented source code if it is not set covered. The technique to navigate through the tokens is similar as in the Prettifier.

4.3.2.4 Updater

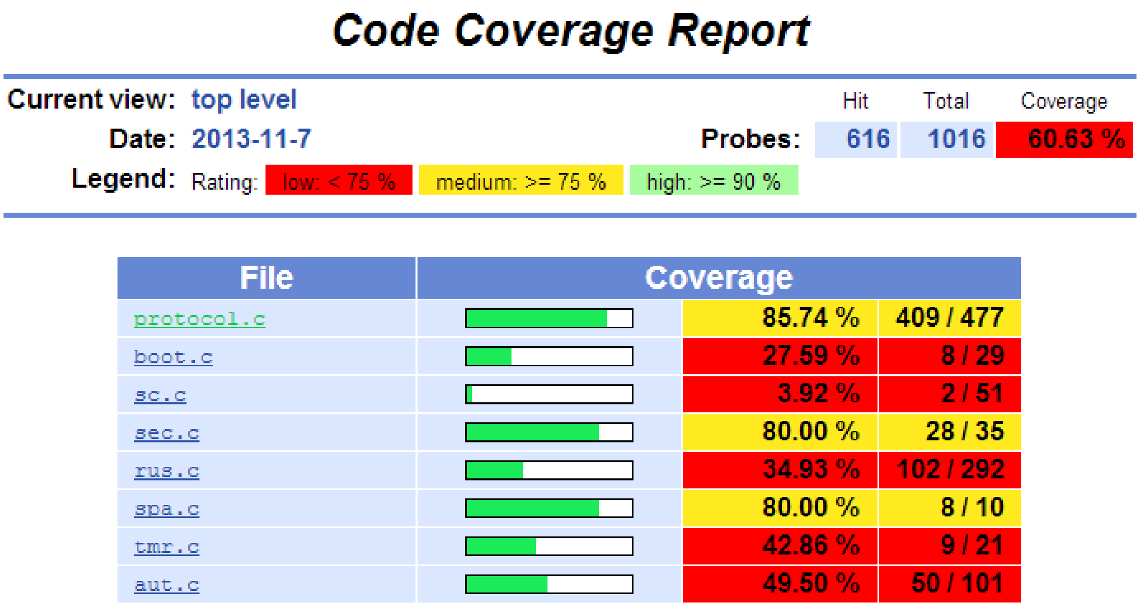
The purpose of the Updater is to evaluate the coverage information of a test run and adjust the instrumentation according to these results. Therefore, it receives the coverage information after a test run. This coverage information is the content of the coverage bit array. The Updater iterates through this bit array and sets the value *covered* of all newly covered probes to `true`. The Updater provides the functionality to create an instrumented source code file from the instrumented token stream data structure. Therefore, it only iterates through the token stream and writes the printed representation of the token into a file. If at least one new probe has been covered in the last test run it creates an instrumented version of the source code from the updated token data structure. Otherwise, the work is done and the Reporter takes over.

4.3.2.5 Reporter

If no new probes have been discovered in the last test run the work is done and a report has to be generated. This is the task of the Reporter. It creates a graphical HTML web report. With this report a developer can study the coverage of his source code. Figure 4.2 shows the overview page of a sample report and Figure 4.3 shows a specific file. The layout is strongly inspired by LCOV[2], which is a graphical front-end for GCC's coverage testing tool gcov[3]. The coverage percentage in the report is the number of the covered probes in relation to the total number of inserted probes. In the file specific report it can be seen if a branch has been taken or not. A green line indicates that a branch has been taken at least once and red one indicates that it has not been taken. This module should satisfy Requirement 6.

[2]http://ltp.sourceforge.net/coverage/lcov.php

[3]http://gcc.gnu.org/onlinedocs/gcc/Gcov.html



Code Coverage Report

			Hit	Total	Coverage
Current view:	top level				
Date:	2013-11-7	Probes:	616	1016	60.63 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %				

File	Coverage		
protocol.c		85.74 %	409 / 477
boot.c		27.59 %	8 / 29
sc.c		3.92 %	2 / 51
sec.c		80.00 %	28 / 35
rus.c		34.93 %	102 / 292
spa.c		80.00 %	8 / 10
tmr.c		42.86 %	9 / 21
aut.c		49.50 %	50 / 101

Figure 4.2: Overview page of the HTML code coverage report.

4.3.2.6 Summary

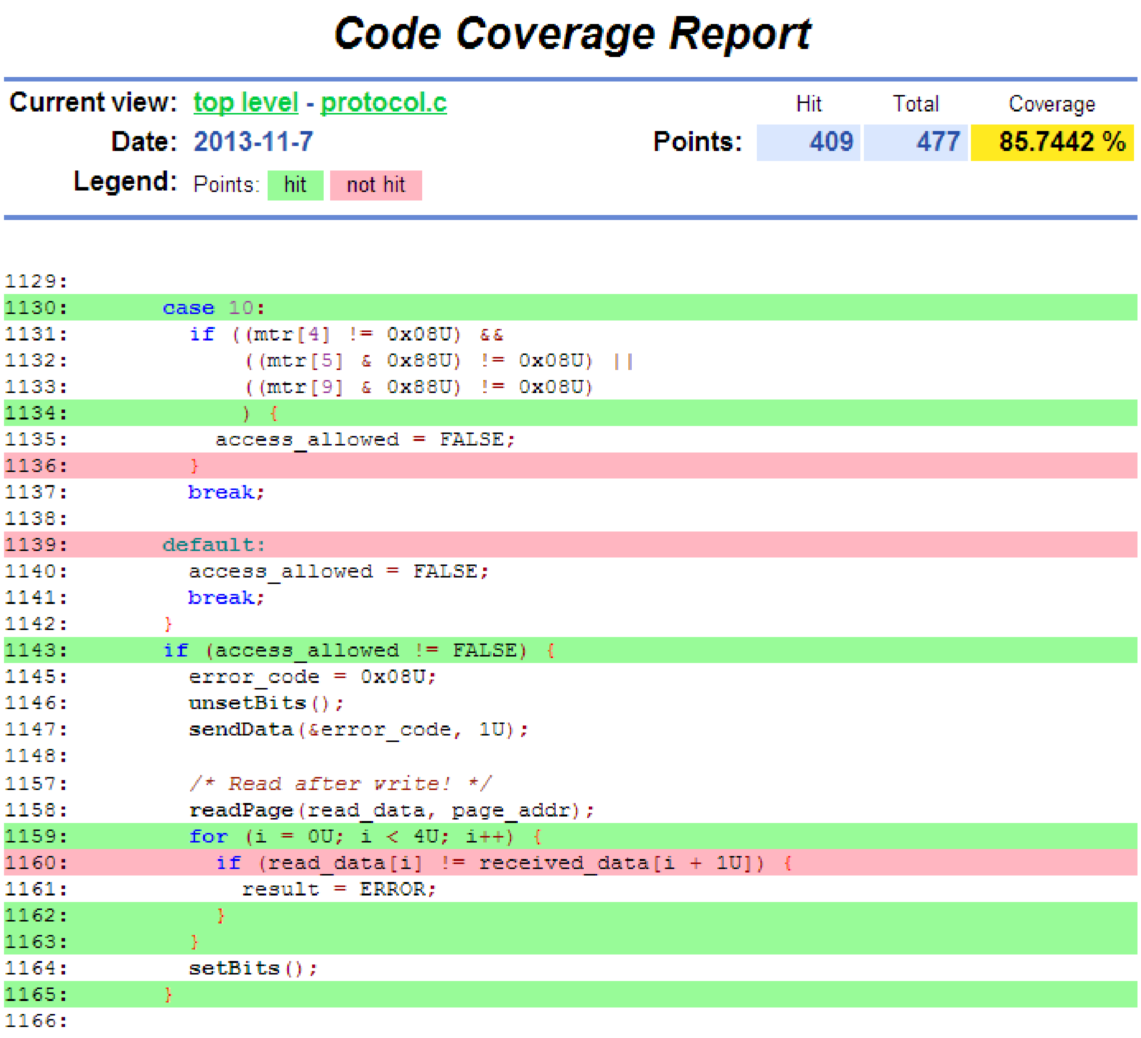
The *C-CoCATool* provides the functionality to instrument source code. Furthermore, it is able to update this instrumentation based on the coverage information collected in the previous test run. At the end of the measurement procedure it creates a visual HTML report. How the *C-CoCATool* interacts with the other components of the system is treaten in Section 4.3.5.

4.3.3 2-link

We discussed the 2-LINK already in Section 3.3.3. In our code coverage measurement system we mainly use the 2-LINK to deploy a binary to the embedded system.

4.3.4 Test Suite Modifications

A test suite that is integrable within our code coverage measurement system has to provide some extra functionality. We implemented this functionality in a C++ class called `Coverage`. This class provides functions to zero out the coverage memory area in the RAM of the embedded system or to read this area out and append it to the already gath-



Code Coverage Report

Current view: top level - protocol.c

Date: 2013-11-7

Legend: Points: 🟩 hit 🟥 not hit

	Hit	Total	Coverage
Points:	409	477	85.7442 %

```
case 10:
 if ((mtr[4] != 0x08U) &&
   ((mtr[5] & 0x88U) != 0x08U) ||
   ((mtr[9] & 0x88U) != 0x08U)
   ) {
  access_allowed = FALSE;
 }
 break;

default:
 access_allowed = FALSE;
 break;
}
if (access_allowed != FALSE) {
 error_code = 0x08U;
 unsetBits();
 sendData(&error_code, 1U);

 /* Read after write! */
 readPage(read_data, page_addr);
 for (i = 0U; i < 4U; i++) {
  if (read_data[i] != received_data[i + 1U]) {
   result = ERROR;
  }
 }
 setBits();
}
```

Figure 4.3: HTML code coverage report of a specific file.

ered coverage information. Furthermore, it provides the functionality to create a result file with the coverage information collected during the test run.

Before we explain where the modifications take place during the execution of the test suite, we describe how a test case is designed in the test suite we used for our experimental results. Normally, the embedded system is stopped. Before executing a test case the test suite stops the embedded system and after executing the test case it stops the device. We call this starting process a *power up* and this stopping process a *power down*. However, it could be the case that there are unexpected power downs, for instance if a test case is aborted. If we only read the coverage memory out after a test case we would miss covered probes. In order to be sure that we do not miss covered probes we read the memory out after every power down. There exists also test cases that perform more than one power down.

Now let us explain at which steps of the execution our modifications take place. The test suite holds a buffer that has the same size as the coverage bit array on the embedded device. We call this buffer *coverage image*. Our modified test suite zeros out the memory area that is reserved for coverage information (the bit array) before starting the very first test. After each power down of the embedded system during the test run, the test suite reads out the coverage memory from the embedded system and updates the coverage image. After each power up of the embedded system during the test run, the test suite sets the coverage memory with the stored image, to ensure that the memory is not corrupted. At the end of the test run, the test suite writes the image of the coverage memory area in a result file.

4.3.5 Toolchain

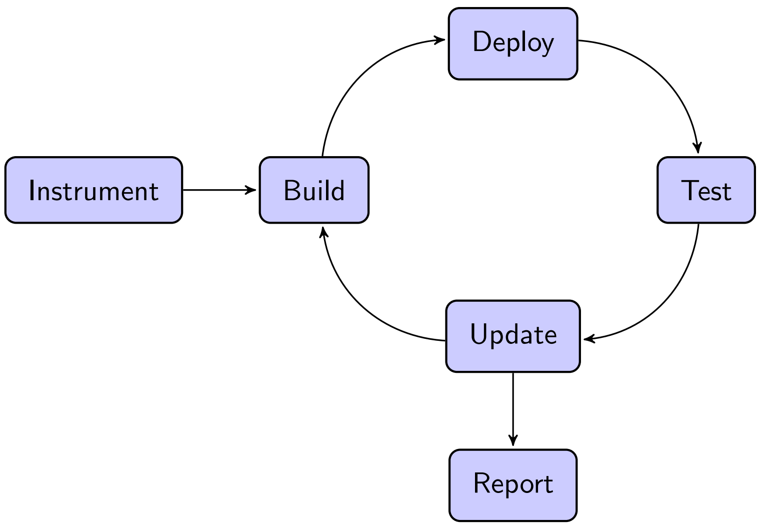


Figure 4.4: A schematic overview of the toolchain of our code coverage measurement system.

Our code coverage measurement system can also be seen as a toolchain. The system consists of six different steps or phases, namely Instrument, Build, Deploy, Test, Update, and Report. Figure 4.4 illustrates this toolchain view. In order to overcome the issue of aborted test runs due to timing violations (see Section 4.2.3), we developed and implemented an iterative approach. This approach is based on Algorithm 1. A controlling script manages the communication between the components of the system in order to provide the functionality of the algorithm. In the following subsections we give more information on each of the steps in the toolchain.

Algorithm 1 Code coverage algorithm

```
1: instrument source code
2: repeat
3:    build updated executable
4:    deploy executable on chip
5:    run test-bench on protocol
6:    update the instrumentation
7: until no new probes are found
8: create visual report
```

4.3.5.1 Instrument

Two components of the system are needed to perform this step, Coccinelle and C-CoCATool. The first thing in this step is the creation of the token streams by Coccinelle. The token stream of the source code is the input for the C-CoCATool, which performs the instrumentation and returns an instrumented version of each source code file.

4.3.5.2 Build

For this step only the Compiler is needed. The instrumented source code files from the Instrument step is the input for the the Build step. The controlling script first copies the files in the right folders and then starts the build.

4.3.5.3 Deploy

This step is done by the 2-link debug probe. The input for the Deploy step is the binary produced in the Build step. To be entirely accurate, a little bit more has to be done to produce an executable binary for the embedded device. Nevertheless, these steps are not really interesting in the context of this thesis.

4.3.5.4 Test

After deploying the instrumented binary to the embedded device a test run can be launched. For this step only the TEST SUITE is necessary. To be exact, the TEST SUITE uses components to communicate with the embedded target during the test, but these components are not of interest in the toolchain view. Nevertheless, we discuss them in Section 5.1.1. The output of this step is a file with the collected code coverage information during the test run.

4.3.5.5 Update

This step is performed by our C-CoCATool. The input for the Update step is the collected code coverage information of the Test step. The first thing to do here is to check if new probes have been covered in the test run. The term *new* in this context means that in the last test run at least one probe has been covered that has not been covered in earlier test runs. If new probes have been covered in the last test run, the C-CoCATool updates the instrumentation, which means set for all newly covered probes the value *covered* to `true`. Therefore, they will not appear in updated version of the source code. After that it creates the updated instrumented source code files as an output of this step. If new probes have been covered in the last test run, the script continues with the deploy step. Otherwise, the controlling script proceeds with the Report step.

4.3.5.6 Report

As said in the last section, after not finding new points we are almost finished. This is the final step of the toolchain and it is performed by the C-CoCATool. The input for this step is the previous gathered code coverage information and the output is a little HTML coverage report.

4.4 Summary

In this chapter we presented a code coverage measurement system that is able to measure a combination of function and decision coverage of applications running on embedded systems with very limited resources. The systems consists of two major parts. On the one hand we have our instrumentation and analysis tool C-CoCATool and on the other hand we have COCCINELLE, the test suite, a compiler and a debugger.

The C-CoCATool instruments the program at the source code level. With this system we realized an iterative approach, which is able to update the instrumentation by removing covered probes. Furthermore, we overcome the problem of aborted test cases due to timing violation caused by the instrumentation. At the end of the instrumentation process we generate a visual report of the gathered code coverage information. Our approach has two restrictions. It does not support `do-while` loops and it also instruments source code that might be inactive due to preprocessor statements.

This code coverage measurement system is designed to fulfill the requirements in Section 1.2.1. Therefore, we can say that this system fulfills the Requirements 1, 3, and 6. To determine if the other requirements are fulfilled we need experimental results. Therefore, we discuss them in Section 5.3.6.

Chapter 5

Evaluation

Contents

In this work we proposed two approaches for measuring code coverage on embedded systems with very limited resource (Chapter 3 and 4). We started with our binary instrumentation approach that is able to measure function coverage. We implemented this approach in a prototype tool and tested it. After the evaluation of this approach we decided to change our focus to a source code instrumentation approach. One reason for that was that we did not want to restrict our tool to a specific instruction set. Furthermore, we can measure more sophisticated code coverage metrics with the source code instrumentation approach. The following results will confirm this. Hence, we focus in this section on the results of our source code instrumentation approach and mention results of the binary approach only very briefly. The following results will show that this approach is able to measure a combination of function and decision coverage on an embedded system with highly limited resources in an industrial use case. First of all we describe our test setup. Afterwards we will present the results and give a discussion.

5.1 Test Setup

We tested our source code instrumentation approach on an MRK-IIIe attached embedded system provided by *NXP Semiconductors*. This is a RISC controller that uses the third generation of NXP's Micro RISC Kernel (MRK-IIIe). The MRK-IIIe is 16-bit microcontroller using a Harvard architecture. The processor supports a maximum clock frequency of 16MHz. The size of the user ROM for application on the board is 32 KB and the size of the RAM is 2 KB. Depending on the software up to 1 KB of RAM is available for coverage data.

We tested our approach on four protocols of industrial use with a modified version of the original test suite of the protocols. Therefore, we used basically the same test setup as for a normal test suite run without applying our test coverage measurement approach. To get a better understanding of the test setup, we will start with a description of the original test setup and afterwards we will describe how we modified this setup for our coverage measurement approach.

5.1.1 Normal Test Setup

The original test setup for the MRK-IIIe consists of hardware and software parts. The hardware part includes a host machine, a *2-link debug probe*, a *TED-Kit* and a development board with a MRK-IIIe applied on. Figure 5.1 illustrates how the hardware parts are connected. The software part of a normal test run consists of the test suite and the API-interfaces for the 2-link and the TED-Kit.

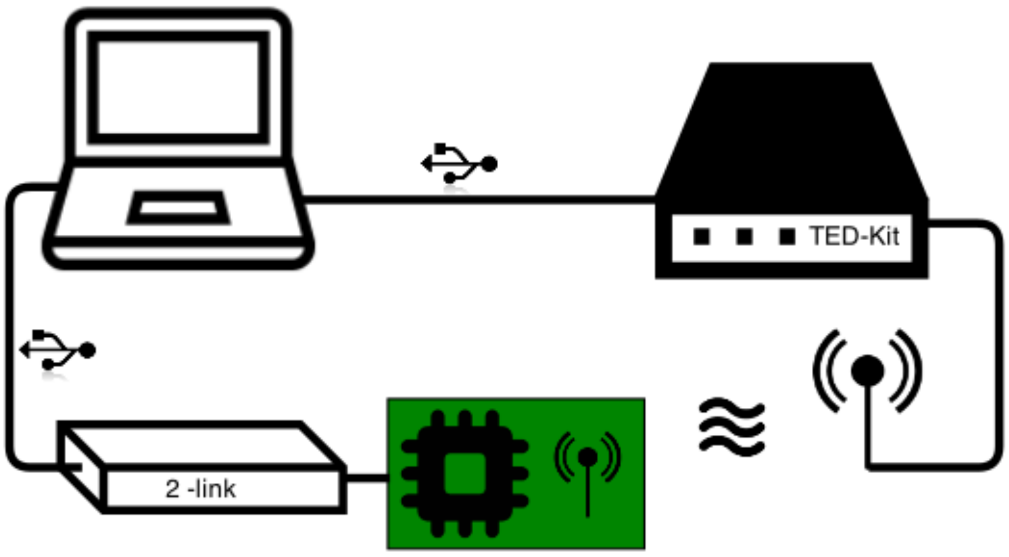


Figure 5.1: Test setup for the embedded device.

The test suite provides several test benches for different protocols on the MRK-IIIe. It can address the 2-link and the TED-Kit by calling their API-methods. Additionally, the test suite evaluates the results of the test cases.

To test the firmware on a MRK-IIIe we used an embedded development board. This board is equipped with the MRK-IIIe and all the necessary peripherals. Figure 5.1 illustrates such a board with a mounted chip and an antenna as a peripheral.

The *2-link debug probe* is the debug interface to the MRK-IIIe and the only way to debug the target. It has been described in Section 4.3.3.

The TED-Kit (Transponder Evaluation and Development Kit) is the tool that is able to stimulate the embedded device via wireless communication. In our test setup, this was the device that applied the test stimuli on our embedded device under test. The TED-Kit is equipped with an API-interface and can be connected to a host machine with a USB cable. Note that this wireless communication adds nondeterministic timing behavior to the test suite.

In the host machine everything comes together. It is connected with the 2-link and the TED-Kit. The test suite, which applies all the test stimuli over the TED-Kit, also runs on it. Additionally, a new binary can be deployed from the host machine on the target device. In conclusion, the host is the "brain" of the whole system.

5.1.2 Modified Test Setup

To apply our code coverage measurement approach it was necessary to make some modifications to the test setup (see Section 4.3.4). We made all this modifications to the software side of the test setup.

The major change was the introduction of our code coverage toolchain. As described in Section 4.3.5, the toolchain consists of six steps, namely instrument, build, deploy, test, update, and report. We integrated the test suite application into the test step. For this integration, some adaptations in the test suite were necessary. These modifications were responsible for keeping track of the covered probes and are described in detail in Section 4.3.4

Algorithm 1 illustrates our arrangement of the six steps in this test setup. The instrumentation step is a preliminary one and is only applied once. The report step is a final step and also only applied once. Therefore, only the first iteration consists of five steps. Every other iteration needs just four steps (build, deploy, test, update).

During our test runs we measured the execution time for each step in every iteration. Furthermore, we measured the number of newly covered probes in every iteration. We separated all source files in two disjunct groups. The first group consisted of all files that were related to a specific protocol and the second group consisted of all auxiliary files. Note that the test benches focused on the protocol-related files. Hence, we measured not only the overall coverage but also the coverage for these disjunct groups separately. In the next section we present the results of these test runs.

5.2 Results

5.2.1 Binary Instrumentation Approach

Since we achieved much better results with the source code instrumentation approach, we mention our results with the binary approach only very briefly. We tested the approach with a demo software application on the MRK-IIIe. The results showed that it is possible to measure function coverage on a resource-limited embedded system with our approach. In our tests we got no problems with timing constraints. Hence, we were able to cover all probes in the first iteration. This makes this approach very fast. We are aware that the function coverage criterion is a very coarse grained one. Nevertheless, it can be useful for a first function profiling. This is possible for the reason that we store the address of each covered function consecutively.

5.2.2 Source Code Instrumentation Approach

In our experiments we tested four different firmware protocols, we named them Protocol I, Protocol II, Protocol III, and Protocol IV. Since the behavior of the system is not fully deterministic we performed three fully instrumented test runs to reduce the effect of outliners in the evaluation. Fully instrumented means that all associated files, protocol related as well as the auxiliary files, were fully instrumented. We measured the time for all steps of each iteration. Additionally, we measured the covered probes per iteration. Tables 5.1 (Protocol I), 5.2 (Protocol II), 5.3 (Protocol III), and 5.4 (Protocol IV) show the results. Figures 5.2 (Protocol I), 5.3 (Protocol II), 5.4 (Protocol III), and 5.5 (Protocol IV) coherences of the different runs of the same protocol.

For a better understanding of the data presentation in these tables and figures, we give a brief description in the following paragraphs. Let us start with the explanation of the tables. The first column, which is named *Iteration*, represents the iteration number. The iteration number is the counter of the loop in the toolchain, for instance if we are in the third iteration, we did already two loops of build, deploy, test, and update before. In the second column (*Instrumentation*) the instrumentation time is listed. This step is only needed in the first iteration. Therefore, the cells for the other iterations are empty. In the next column (*Build*) the time that was needed to rebuild the source code with the added or updated instrumentation is recorded. The fourth column (*Deploy*) shows the time in seconds for loading the executable binary to the target. Column five (*Test*) represents the runtime of the test suite. For the sake of readability, we present these values in minutes instead of seconds. The next column (*Evaluation*) lists the time that was needed to evaluate the results of the iteration and to update the instrumentation in the source code with the determined information. In the column *Offline Overhead* the overall offline overhead for this specific iteration is listed. This overhead is the sum of the steps instrumentation, build, deploy, test and update. The next column *Overall Runtime* represents the time of the whole iteration. In the column *New Probes* the overall number of probes that were newly covered in this iteration is listed. The next two columns, which are listed under the name (*Protocol*), refer to the protocol related files. The first column shows the number of probes in a protocol related file that were newly discovered in this iteration. The second column shows the overall coverage for the protocol related files after each iteration in percentage. In the second to the last column (*Auxiliary*) the number of newly discovered probes in all auxiliary files is listed. The symbol in the last column (*Test Result*) gives a hint on the number of tests which had produced the expected result. The symbol $\times$ indicates that either only a very low number of tests produced the expected results or that the test suite run did not produce any results. The symbol $\sim$ indicates that a lot of tests produced the expected result. If every single test produced the expected result, we indicate this with the symbol $\surd$.

The row with the bold text represents the overall numbers of the whole test run. To get a comparison between the runtime of the original test procedure and the code coverage test procedure, we calculated an average time of the original test suite run for each protocol. It can be found in the last section of the table, for instance, *Protocol I - original* in table 5.1. The slowdown factor value in the headline of a protocol is the total overall runtime of this protocol divided by the overall runtime of the original test-suite.

For each of the protocols we provide a number of diagrams (Figures 5.2 on page 63, 5.3 on page 64, 5.4 on page 65, and 5.5 on page 66) to illustrate certain aspects of our results. The diagram *Total Test-Suite Runtime* shows the differences and similarities between the original run and the code coverage runs with respect to the execution time per iteration. The bar chart *All Files - New Cov. Probes per Iteration* illustrates for each code coverage run the newly covered probes per iteration. *All Files* means that we do not differentiate between protocol or auxiliary files. The next diagram (*All Files - Total Probes Covered*) presents the different coverage runs with respect to the total number of covered probes and iterations. A probe in a graph shows the total number of covered points after an iteration. The next three diagrams (*All Files - Coverage in %*, *Protocol Files - Coverage in %*, and *All Auxiliary - Coverage in %*) show the increasing coverage percentage after each iteration. The first of the latter three diagrams illustrates the coverage percentage for all files. The last two of them show only the coverage percentage for either the protocol files or for the auxiliary files.

We checked the binaries of the instrumented source code files and we saw that the source code approach only needed two instructions for reporting a probe as covered. One instruction to set save the right bitmask to a register and the other one for oring this register with the right byte in the code coverage memory area. For instance, if the probe with the ID 607 should be stored. The first instructions copies the constant 128 into a register and the second instructions performs an OR operation with the 76th byte of the coverage data in the memory.

5.3 Discussion

We tested the adequacy of a test suite with respect to a combination of function coverage criterion and decision coverage criterion for four different protocols on the MRK-IIIe. For each of the four protocols we did three test suite runs to reduce the effect of nondeterminism and compared the results afterwards. Note that this four protocols are applications that are under development and not final products. The results of the different protocols are very similar. Therefore, we will start with an exemplary discussion of just one of them. For the other three protocols we will just point out things that are different or of particular interest. At the end we will compare the results, point out the similarities as well as the differences and identify general trends.

5.3.1 Protocol I

The performed results for this protocol are summarized in Table 5.1 (page 55) and Figure 5.2 (page 63). In this paragraph we point out how the overall runtime spread across the five different steps of our method. Execution run one of Protocol I took eight iterations with an overall runtime of 576 minutes. The instrumentation step, which has to be done just once, took two or three seconds. In comparison with the overall runtime this is

Iteration	Instrumentation	Build	Deploy	Test	Update	Offline Overhead	Overall Runtime	New Probes	Protocol		Auxiliary	Test Result
Protocol I - fully instrumented - Execution Nr. 1									slowdown factor: **8.44**			
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1	3	53	8	6	1	65	7	227	132	53.0	95	×
2		33	9	6	1	43	7	2	0	53.0	2	×
3		12	9	93	1	22	94	150	87	88.0	63	∼
4		22	6	93	1	29	94	17	2	88.8	15	∼
5		17	6	93	1	24	94	5	0	88.8	5	√
6		13	7	93	1	21	94	1	0	88.8	1	√
7		12	6	93	1	19	94	2	0	88.8	2	√
8		12	6	93	0	18	94	0	0	88.8	0	√
8	**3**	**174**	**57**	**572**	**7**	**241**	**576**	**404**	**221**	**88.8**	**183**	
Protocol I - fully instrumented - Execution Nr. 2									slowdown factor: **7.08**			
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1	2	53	9	6	2	66	7	224	129	51.8	95	×
2		28	9	6	1	38	7	2	0	51.8	2	×
3		12	9	93	1	22	94	152	90	88.0	62	∼
4		22	6	93	1	29	94	20	2	88.8	18	∼
5		17	6	93	1	24	94	6	0	88.8	6	√
6		12	6	93	2	20	94	1	0	88.8	1	√
7		12	6	93	0	18	94	0	0	88.8	0	√
7	**2**	**156**	**51**	**479**	**8**	**217**	**483**	**405**	**221**	**88.8**	**184**	
Protocol I - fully instrumented - Execution Nr. 3									slowdown factor: **7.02**			
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1	3	53	7	6	2	65	7	224	129	51.8	95	×
2		34	10	6	1	45	6	2	0	51.8	2	×
3		13	8	90	3	24	90	152	90	88.0	62	∼
4		24	6	94	1	31	94	20	2	88.8	18	∼
5		17	7	94	1	25	94	6	0	88.8	6	√
6		11	6	94	1	18	94	1	0	88.8	1	√
7		11	6	94	0	17	94	0	0	88.8	0	√
7	**3**	**163**	**50**	**475**	**9**	**225**	**479**	**405**	**221**	**88.8**	**184**	
Protocol I - original												
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1		**41**	**7**	**67**		**48**	**68**					√

Table 5.1: Comparison between instrumented execution and normal execution of Protocol I.

Iteration	Instrumentation	Build	Deploy	Test	Update	Offline Overhead	Overall Runtime	New Probes	Protocol		Auxiliary	Test Result
Protocol II - fully instrumented - Execution Nr. 1									slowdown factor: **6.79**			
[-]	[sec]	[sec]	[sec]	[~min]	[sec]	[sec]	[~min]	[-]	[-	%]	[-]	[-]
1	3	42	7	10	2	54	10	215	62	36.3	153	×
2		26	7	143	2	35	143	138	85	86.0	53	×
3		20	7	186	2	29	186	25	6	89.5	19	√
4		20	7	186	2	29	186	2	0	89.5	2	√
5		20	7	186	2	29	186	2	0	89.5	2	√
6		19	7	186	0	26	186	0	0	89.5	0	√
6	**3**	**147**	**42**	**895**	**10**	**202**	**899**	**382**	**153**	**89.5**	**229**	
Protocol II - fully instrumented - Execution Nr. 2									slowdown factor: **4.18**			
[-]	[sec]	[sec]	[sec]	[~min]	[sec]	[sec]	[~min]	[-]	[-	%]	[-]	[-]
1	3	42	7	9	2	54	9	214	62	36.3	152	×
2		26	7	7	1	34	8	97	52	66.7	45	×
3		21	6	178	1	28	179	67	39	89.5	28	√
4		17	6	178	1	24	179	3	0	89.5	3	√
5		12	7	179	2	21	179	0	0	89.5	0	√
5	**3**	**118**	**33**	**551**	**7**	**161**	**554**	**381**	**153**	**89.5**	**228**	
Protocol II - fully instrumented - Execution Nr. 3									slowdown factor: **5.16**			
[-]	[sec]	[sec]	[sec]	[~min]	[sec]	[sec]	[~min]	[-]	[-	%]	[-]	[-]
1	3	37	9	9	1	50	9	214	62	36.3	152	×
2		70	7	137	2	79	138	141	85	86.0	56	×
3		22	7	178	1	30	179	27	6	89.5	21	√
4		16	6	178	1	23	178	1	0	89.5	1	√
5		11	7	178	1	19	178	0	0	89.5	0	√
5	**3**	**156**	**36**	**679**	**6**	**201**	**683**	**383**	**153**	**89.5**	**230**	
Protocol II - original												
[-]	[sec]	[sec]	[sec]	[~min]	[sec]	[sec]	[~min]	[-]	[-	%]	[-]	[-]
1		**41**	**6**	**131**		**47**	**132**					√

Table 5.2: Comparison between instrumented execution and normal execution of Protocol II.

negligibly small. The time for building the executable is also very small. It took less than a minute in each run. The fewer files are affected by updating the instrumentation during one iteration, the smaller is the effort for rebuilding the executable. Hence, sometimes it took only 12 seconds. Deploying the binary on the target was also a minor factor in the overall runtime, it usually took less than 10 seconds. The time needed for the evaluation of the results of an iteration is also negligibly small. The offline overhead, which consists of instrumentation, build, deploy, and update, took approximately half a minute per iteration, which is only a very small part of the overall runtime. On the other hand we have

Iteration	Instrumentation	Build	Deploy	Test	Update	Offline Overhead	Overall Runtime	New Probes	Protocol		Auxiliary	Test Result
Protocol III - fully instrumented - Execution Nr. 1									slowdown factor: **4.02**			
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1	3	14	7	6	1	25	7	204	86	41.7	118	×
2		30	9	84	1	40	84	138	90	85.4	48	×
3		19	8	98	1	28	99	37	9	89.8	28	√
4		18	6	98	0	24	99	0	0	89.8	0	√
4	**3**	**81**	**30**	**287**	**3**	**117**	**289**	**379**	**185**	**89.8**	**194**	
Protocol III - fully instrumented - Execution Nr. 2									slowdown factor: **5.18**			
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1	4	46	8	6	2	60	7	201	86	41.7	115	×
2		24	9	80	2	35	81	143	93	86.9	50	×
3		22	8	94	2	32	95	37	6	89.8	31	√
4		16	6	94	2	24	95	1	0	89.8	1	√
5		13	6	94	0	19	95	0	0	89.8	0	√
4	**4**	**121**	**37**	**369**	**8**	**170**	**372**	**382**	**185**	**89.8**	**197**	
Protocol III - fully instrumented - Execution Nr. 3									slowdown factor: **5.17**			
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1	2	53	7	6	1	63	7	201	86	41.7	115	×
2		26	8	80	1	35	81	141	93	86.9	48	×
3		19	8	94	1	28	95	39	6	89.8	33	√
4		16	7	94	1	24	95	1	0	89.8	1	√
5		17	7	94	1	25	95	0	0	89.8	0	√
5	**2**	**131**	**37**	**368**	**5**	**175**	**371**	**382**	**185**	**89.8**	**197**	
Protocol III - original												
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1		**41**	**6**	**71**		**47**	**72**					√

Table 5.3: Comparison between instrumented execution and normal execution of Protocol III.

the test step (execution of the test suite) that took the major part of the overall runtime and represents the online overhead. If the test suite run was not aborted as in the first two iterations of each run, the execution of the modified test suite took around 94 minutes per iteration. This is 27 minutes longer than the original test suite needed for one test run. In comparison to the run step all other steps are not very time consuming.

The first two iterations of each run took only a fraction of the normal runtime of the test suite. The reason for this is that the test suite first checks for functionality that is required for running the tests. With the fully instrumented executable, the system is too slow to meet certain timing constraints. Hence, the system aborts the execution.

Iteration	Instrumentation	Build	Deploy	Test	Update	Offline Overhead	Overall Runtime	New Probes	Protocol		Auxiliary	Test Result
Protocol IV - fully instrumented - Execution Nr. 1									slowdown factor: **3.92**			
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1	3	49	8	0	1	61	1	142	72	15.1	70	×
2	0	41	7	251	3	51	252	464	335	85.3	129	×
3	0	39	7	251	1	47	252	12	2	85.7	10	∼
4	0	20	7	251	0	27	252	0	0	85.7	0	∼
4	**3**	**149**	**29**	**754**	**5**	**186**	**757**	**618**	**409**	**85.7**	**209**	
Protocol IV - fully instrumented - Execution Nr. 2									slowdown factor: **3.91**			
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1	4	57	8	0	1	70	1	133	63	13.2	70	×
2	0	40	7	251	2	49	251	477	344	85.3	133	×
3	0	38	7	251	1	46	252	12	2	85.7	10	∼
4	0	38	7	251	2	47	252	0	0	85.7	0	∼
4	**4**	**173**	**29**	**753**	**6**	**212**	**756**	**622**	**409**	**85.7**	**213**	
Protocol IV - fully instrumented - Execution Nr. 3									slowdown factor: **3.92**			
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1	3	56	8	0	1	68	1	133	63	13.2	70	×
2	0	39	7	251	3	49	252	475	344	85.3	131	×
3	0	34	7	251	2	43	252	14	2	85.7	12	∼
4	0	18	7	252	1	26	252	0	0	85.7	0	∼
4	**3**	**147**	**29**	**755**	**7**	**186**	**757**	**622**	**409**	**85.7**	**213**	
Protocol IV - original												
[-]	[sec]	[sec]	[sec]	[∼min]	[sec]	[sec]	[∼min]	[-]	[-	%]	[-]	[-]
1		**41**	**5**	**192**		**46**	**193**					∼

Table 5.4: Comparison between instrumented execution and normal execution of Protocol IV.

Nevertheless, in this aborted run our tool covered more than 50% of all inserted probes and removed them from the instrumentation in the update step. In the second iteration, the test suite was also stopped earlier due a violated timing constraint. However, the probes that cause a violation during the test step are removed at the update step. This is exactly what happens in the second iteration. Here we only covered two points. These two points were responsible for the timing violation and were removed at the update step. As can be seen in the column *Test Result* in Table 5.1, we obtained no test success in the first two iterations. This is indicated by the symbol $\times$.

In iteration three, the test suite performed a full run for the first time. The protocol coverage also increased from around 52% to 88%. In this run almost all tests succeeded, just two test cases failed. This is indicated by the symbol $\sim$. In iteration four, we covered just two new probes and increased the protocol coverage to the final value of 88.8%. Only one test case failed.

In iteration five, all test cases of the test suite succeeded for the first time. We covered no new probes in protocol related files. However, we covered five auxiliary related probes. Since we specified that we only stop if the tool does not cover any new probes in the last iteration, we ran two respectively three more iterations. In these two or three iterations, we improved only coverage in auxiliary files. It is remarkable that after the iteration in which all tests succeed for the first time, from now on called the *succeeding iteration*, the tool only covered one or two auxiliary related probes.

A very interesting question is how much longer our method needs compared to the unmodified test suite. Therefore, we divide the runtime of our code coverage tool with the original runtime by the unmodified test suite. We call the result of this division *slowdown factor*. In the case of a full run with seven or eight iterations we got an average slowdown factor of approximately 7.5 for this protocol. This is still in a range of an overnight task. Diagram *Total Test Suite Runtime* in Figure 5.2 illustrates the runtime of each run. The results are also interesting if we stop after an earlier iteration. The following list discusses the effects of an earlier stop.

- Stop after the fifth iteration. If we stop after the succeeding iteration, we get a slowdown factor of only 4.35. Since the test suite is designed to focus on the protocol related functions and not on the auxiliary functions this would be a reasonable option. Additionally, we already reached here 100% of the possible protocol coverage and 99% of the possible overall coverage[1].

- Stop after the fourth iteration. It is remarkable that in each run of Protocol I we reached 100% of the possible protocol coverage one iteration before the succeeding iteration. Here we get a slowdown factor of approximately 3.

- Stop after the third iteration. Another interesting point is that if we stop after the first full run of the test suite we almost get the 100% of the possible protocol coverage. This would give us a slowdown factor of only 1.6 in case of Protocol I.

As said at the beginning, the Protocol I is typical but there are some exceptions we want to point out. All other protocols reached the full 100% of the possible protocol

[1]With possible coverage we mean the highest coverage we measured in our execution runs.

coverage with the first full test run, which was iteration number three in each run of every protocol. Therefore, Protocol I is an outlier in this respect. Furthermore, Protocol I is the only protocol where in one of the first three iterations just two probes were covered. In the other protocols the number of newly covered probes per iteration where significantly higher between the first and the third iteration. This behavior can also be seen in the diagrams in figure 5.2. In the other protocols the number of newly covered probes is very high in the first three iterations. After the third iteration the curve flattens out.

5.3.2 Protocol II

The results for this protocol are summarized in Table 5.2 and Figure 5.3. As pointed out at the beginning of this section, the results of the different protocols are very similar. In the preceding subsection we discussed Protocol I exemplarily. Therefore, we discuss Protocol II briefly and point out differences to Protocol I or aspects that are of particular interest. The average slowdown factor for the full run of our code coverage tool for Protocol II is 5.37. However, if we had stopped after the succeeding iteration it would be only 2.51. For this protocol the succeeding iteration is also the first iteration where the test suite performs a full run at the first time. Another remarkable thing is that we had an outlier at iteration two of the second run. Here, the test suite could not execute all test cases and aborted after seven minutes. This behavior is illustrated in several diagrams in Figure 5.3 and led to a significant better overall runtime. As in Protocol I, one run took also one iteration more than the other runs.

5.3.3 Protocol III

The results for this protocol are summarized in Table 5.3 and Figure 5.4. The results of Protocol III reveal nothing new. The average slowdown factor for the whole run of our code coverage tool is 4.79. The third iteration is the succeeding one and also the first one where the test suite performs a full run for the first time. If we would stop after this iteration, we would get an average slowdown factor of only 1.86. Furthermore, as in Protocol I and II, one run took one iteration more than the others.

5.3.4 Protocol IV

The results for this protocol are summarized in Table 5.4 and Figure 5.5. The results of Protocol IV are very similar to the results of Protocol III except for two points. The first difference is that one test case failed in every execution, also in the execution of the unmodified test suite. If we disregard this test case we get the following results. The average slowdown factor for the whole run of our code coverage tool is 3.92. The third iteration is the succeeding one and also the first one where the test suite performs a full run for the first time. If we would stop after the third iteration, we would get an average slowdown factor of only 1.86. The second difference is that Protocol IV is the only protocol where each run took exactly the same number of iterations.

5.3.5 Differences Due to Timing

In this section, we discuss differences between test runs due to nondeterministic timing aspects. Sometimes, an execution run needs one iteration more than the other runs. The reason for this is most likely some nondeterministic behavior. This causes variations from the normal runtime behavior. To get a better understanding of these nondeterministic timing aspects we investigated some of the probes that we identified as outliners. With outliners we mean probes that were covered after the succeeding iteration.

After the iteration where all tests succeed for the first time (succeeding iteration), we still covered some new probes in subsequent iterations. To be precise, in all runs of all protocols we covered six different probes after the succeeding iteration. We were interested in these probes and did some further examination. Four of these six probes occurred only one or two times after the succeeding iteration. Three of these four probes were covered just once and the other one was covered in two runs of the same protocol. These probes were covered in other runs in an earlier iteration. The reason for this is most likely some nondeterministic timing behavior. The majority of these probes occur in a loop like `while (value != 0x001) { cover(76) }`. Such probes are covered only in the case that the value is not set as expected at the first time. Note that four of these six probes were related to protocol files.

Two of this six probes of interest are related to auxiliary files. One of these two was outstanding because this probe was covered after the succeeding iteration in seven out of the twelve runs. Additionally, in six of the twelve runs this probe caused an extra iteration, where just this single probe was newly covered. Such an issue should be treated in an extra test or disregarded, otherwise such a probe will cost a lot of time. In our case this probe was in a while loop as described above. Here we waited until a read has finished. The second of these two probes was covered after the succeeding iteration in two different protocols but also in an iteration before the succeeding one in other runs. This probe occurred in an `if` branch. The `if` asks for a peripheral running and if it is not running it waits. In this case we covered the probe.

5.3.6 Summary

As already pointed out, the results of the different protocols are very similar in general. Our results show that it is possible to measure code coverage with respect to the decision coverage criterion on a resource-limited embedded system in reasonable time. The best slowdown factor of all protocols is only 3.91 (Protocol IV - Execution Nr. 2). On the other hand the worst slowdown factor of all protocols is 8.44 (Protocol I - Execution Nr. 1).

Now we are able to determine if the rest of the requirements in Section 1.2.1 is also fulfilled.

R1 (Language Support) Is fulfilled because the system is designed for the language C.

R2 (Memory Consumption) Since we managed to measure code coverage information for each protocol with less than 1200 probes, which means that we needed only a small amount of the one kilobyte, we claim that Requirement 2 is fulfilled.

R3 (Code Coverage Criteria) Is fulfilled. The system measures a combination of function and decision coverage.

R4 (Test Abortion Handling) According to test experimental results, where we can see that test suite aborts the execution in the first iteration of almost every Run, this requirement is also fulfilled

R6 (Total Time) Is fulfilled. In average one run took less than 10 hours. The longest execution took 15 hours (Protocol I, first run) and the shortest less than 5 hours (Protocol III, first run). Since the run that took 15 hours is an outlier, because the other two runs of Protocol I needed 9.2 and 11.4 hours, we claim that it is still doable in an overnight task.

R5 (Visual Report) Is fulfilled (see Section 4.3.2.5).

R7 (Efficient Instrumentation) Since we need only one bit to store the coverage information and only two instructions per probe, we state that this approach also fulfills this requirement.

In summary, our results show that after the iteration where the test suite performs a full run for the first time, which is iteration three in each run, the coverage percentage is not going to improve significantly. Furthermore, in three protocols more than a third of the possible coverable points had been covered in the first iteration. The diagrams regarding the coverage in the Figures 5.2, 5.3, 5.4, 5.5 illustrate this. As it can be seen, each curve in each of these diagrams levels off after the third iteration.

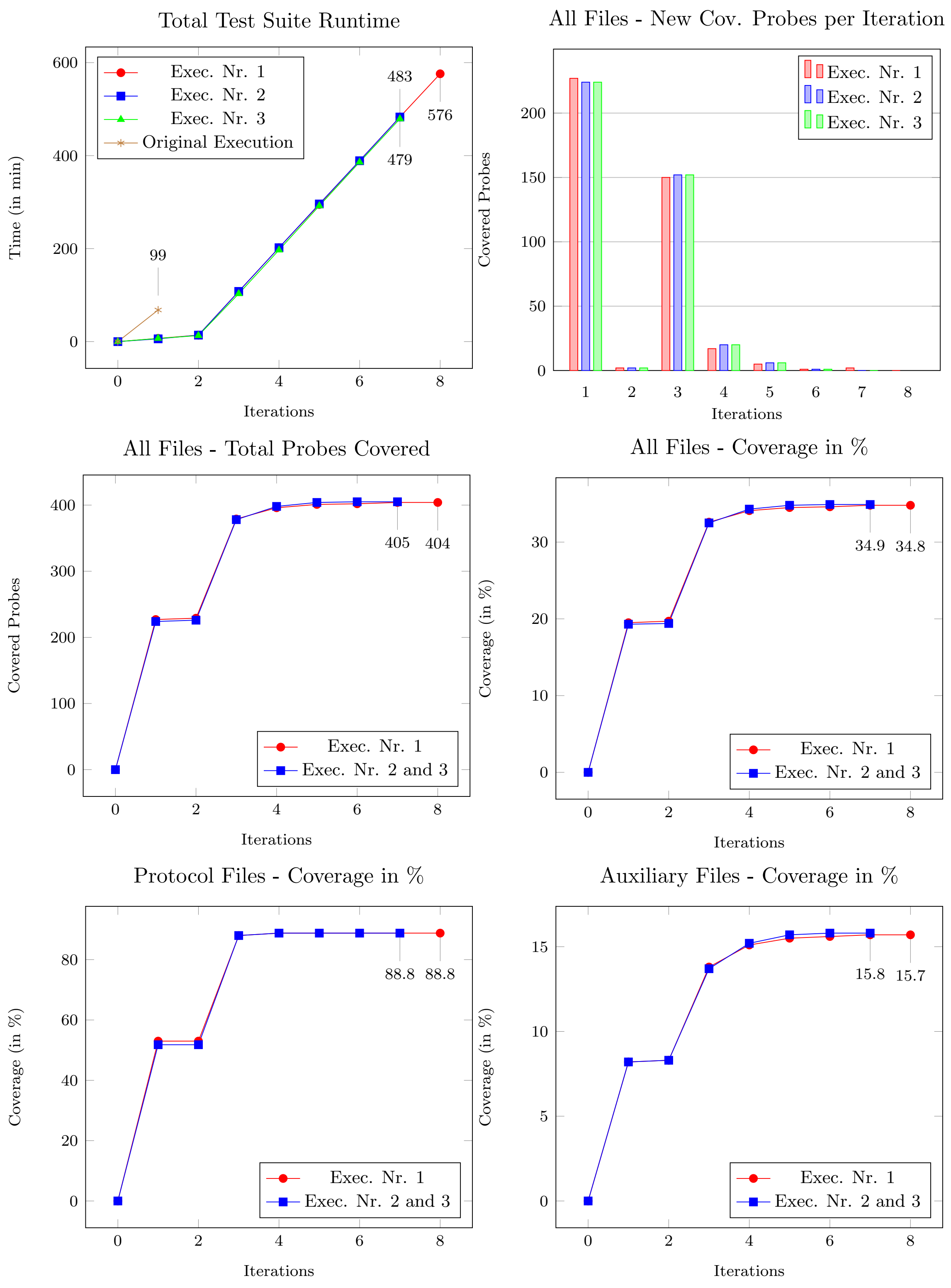


Figure 5.2: Protocol I

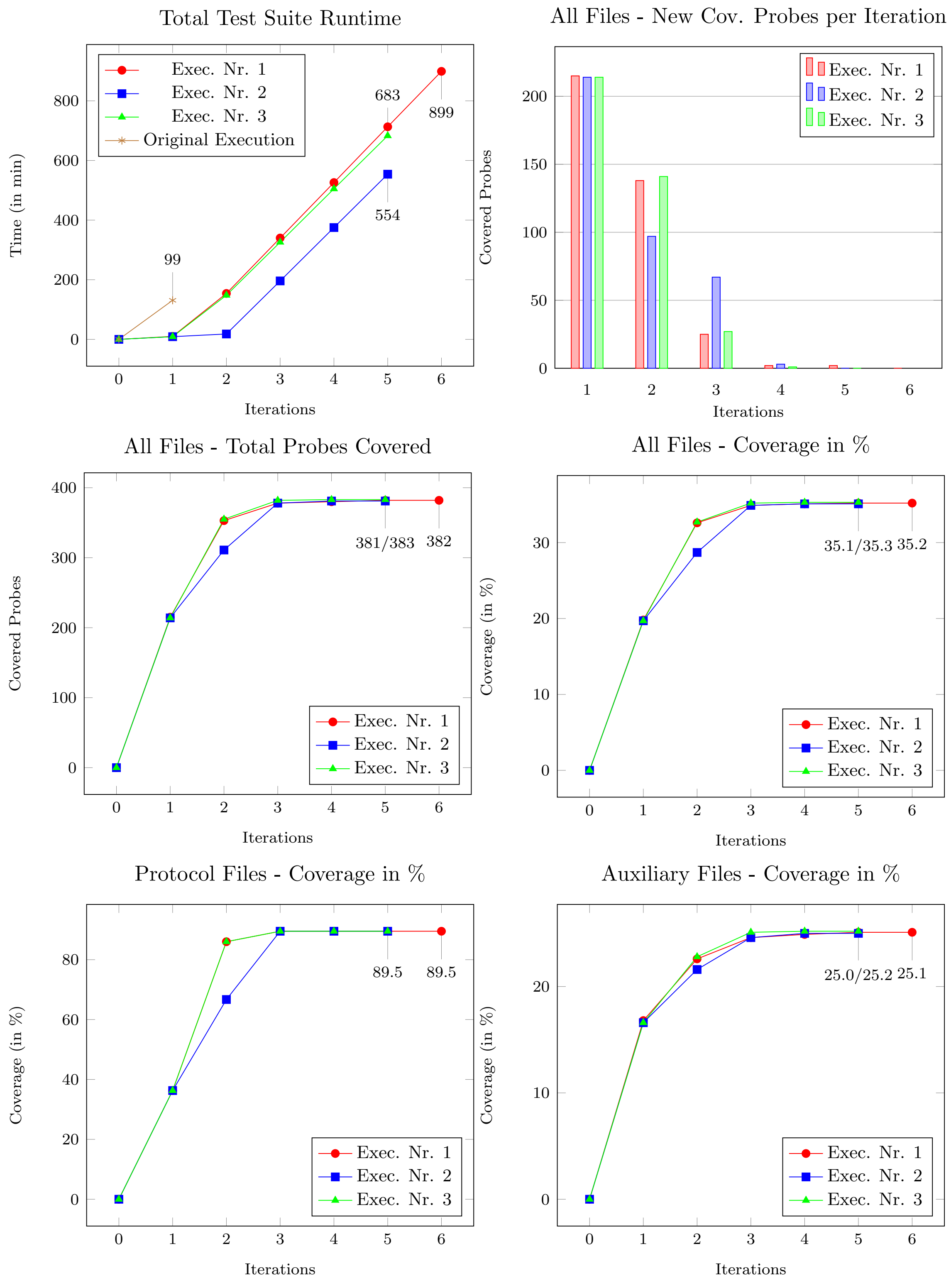


Figure 5.3: Protocol II

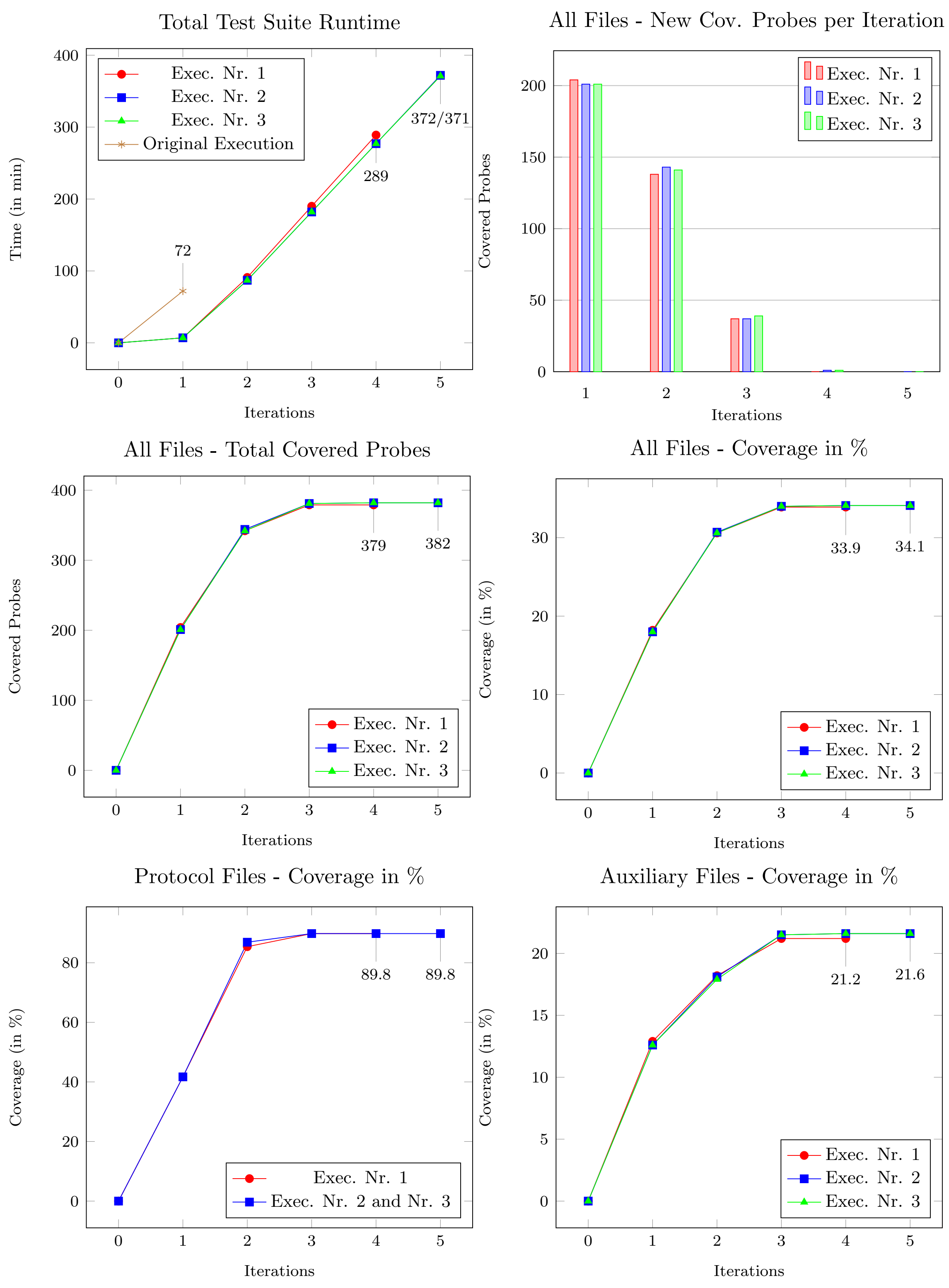


Figure 5.4: Protocol III

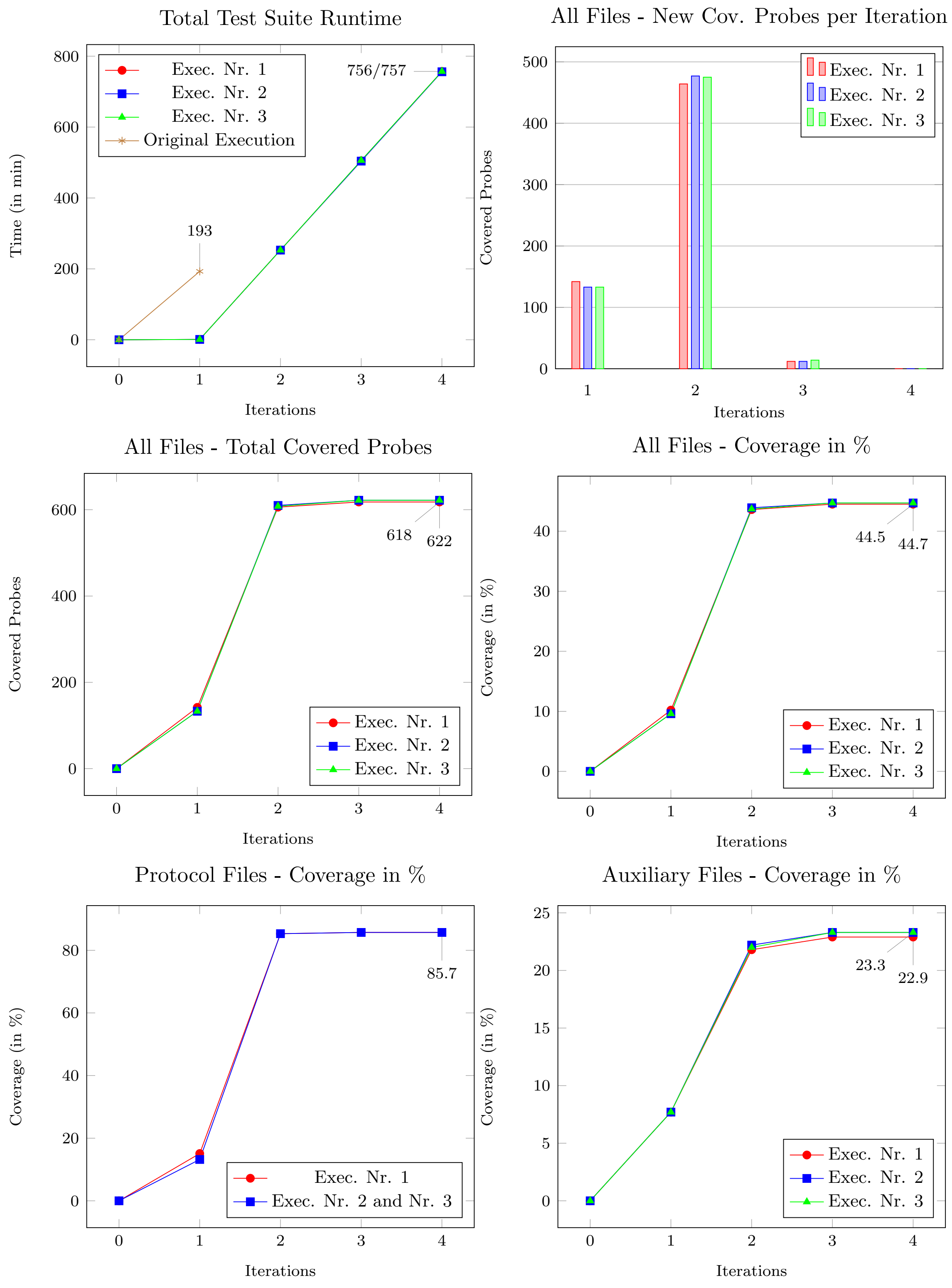


Figure 5.5: Protocol IV

Chapter 6

Related Work

Contents

In this chapter we discuss related work. This chapter consists of two parts. Part one is a market analyses of available commercial tools. First, we introduce key points for the evaluation of the tools. Second, we evaluate and discuss available tools with respect to these introduced key points. Part two is a discussion of literature related to this work.

6.1 Commercial Tools for Measuring Code Coverage

There is a wide range of commercial code coverage tools on the market. It would exceed the scope of this work if we would identify and evaluate all of them. Therefore, we focus on well-established tools. This comparison of commercial tools is based on the requirements in Section 1.2.1, and should give an indication of the state-of-the-art in this market. First, we want to discuss the some additional key points for our comparison and then we will present the results of our evaluation.

6.1.1 Additional Key Points for the Comparison

To evaluate code coverage tools regarding the needs of an resource-limited embedded system we used the requirements in Section 1.2.1 as key points. Furthermore, we identified additional points. Before we start with the tools we want to introduce these additional points.

6.1.1.1 Instrumentation

An interesting point is how the different tools instrument the programs. For instance, at which layer do they insert probes, source code, binary or at an intermediate representation. The last one requires more control over the underlying compiler toolchain. They may also use non-intrusive methods. Tools that require control over the toolchain usually provide more flexibility to instrument and gather information than tools that assume that the compiler toolchain is a black box.

In our case study the compiler toolchain is a proprietary third-party software. Hence, we do not have control over the compiler. Consequently, we have to insert our instrumentation points either in the binary or in the source code.

It is also interesting to find out what the majority of commercial tool developers do. In addition, we want to find out why they instrument at this specific layer. Furthermore, it would be interesting to know why they do not use a specific technique or method. All this information can give an indication on the practical feasibility of some methods.

6.1.1.2 Supported Coverage Criteria

This point is already covered with Requirement 3. We mention it again, because it somehow provides information of the maturity of the tool. Another interesting question is the connection between the instrumentation technique and the achieved code coverage metrics. For instance, is there a product on the market that is instrumenting the binary and supports the MC/DC test suite adequacy criterion?

6.1.1.3 Supported Host Platforms

Another point that should not be as restrictive as the points before, is the question of supported host operating systems. The integration of a code coverage tool is easier if the tool supports the same host operating system as the compiler toolchain.

6.1.2 Evaluation of the Tools

We compared six different commercial code coverage tools. In the following we present our evaluation of the tools according to the requirements and the above additional key points. Table 6.1 gives an overview of the results of this evaluation. We also added *Our System* to this table. It has to be noted that the subsumption relation for coverage criteria is disregarded in this table. It just illustrates the information we found during our research. In the row instrumentation sometimes there is the symbol * after the term SC (Source Code instrumentation). This indicates that it is not totally clear but very likely that the investigated tool instruments on source code level. The row *Embedded Targets* of the table needs further explanation. The single + means that the producers of this system claim that they support embedded systems. If there is a double +, the company have published more details about their embedded target support, for instance a list with supported targets or specifications.

The last seven rows show which tools fulfill which requirement of Section 1.2.1. It has to be noted that the following data consists mainly of information from product websites. For four of the seven requirements in Section 1.2.1 we were not able to find the information that is necessary to determine if the offered tool fulfills the requirement or not. We were not able to find clear information about the memory footprint. Therefore, we cannot say something about Requirement 2. If the tools are able to handle test abortions due to the timing overhead caused by the instrumentation, is also not clear. Therefore, we have no information to determine if Requirement 4 is fulfilled or not. Requirement 5 relies on experimental results. Hence, we cannot say anything about this requirement. Furthermore, it is not an trivial task to determine if a tool is instrumenting efficiently or not (Requirement 7). Therefore, we had to rely on the published information of the companies of the tools. We gathered the information primarily in May 2013.

6.1.2.1 VectorCAST/Cover (Vector Software)

VectorCAST/Cover[1] is a code coverage solution by the company Vector Software Inc. According to the datasheet of VectorCast/Cover[2] this tool provides code coverage for embedded development and therefore supports embedded targets. Nevertheless, it seems that VectorCAST/Cover does not support resource-limited embedded systems as our system does. It is not clear how Vector Software Inc. inserts probes into a program. Therefore, we cannot be sure how the tool instruments. However, as far as we know, it seems that VectorCAST does source code instrumentation. VectorCast/Cover supports statement, branch, and multiple condition/decision coverage critera. The tool is able to handle Ada83/95, C, C++. Windows, Unix, and Linux are the supported host operating systems.

[1]http://www.vectorcast.com/software-testing-products/embedded-code-coverage
[2]http://www.vectorcast.com/sites/default/files/pdf/resources/vectorcast_cover.pdf

6.1.2.2 Testwell CTC++ Test Coverage Analyser (Verifysoft Technology)

The test coverage analyser tool Testwell CTC++[3] is a software product from Verifysoft Technology GmbH. As far as we know the tool does source code instrumentation with the help of a preprocessor. For embedded targets, Testwell has an add-on named bitCov. With that tool the needed memory consumption on the embedded target can be reduced. For one probe one bit of memory is needed. Verifysoft claims that Testwell produces a very low instrumentation overhead. Nevertheless, the amount of the instrumentation overhead in the executable is not clear. Verifysoft also claims that their system can perform code coverage in all embedded targets. Furthermore, the company states that the tool works even with the smallest targets and microcontrollers and additionally the tool works with all compilers or cross-compilers. However, adaptations are needed to support a new target, at least of two points. First, the preprocessor has to be able to process the target-specific C extensions. Second, there has to be some mechanism to extract the gathered data from the device.

Testwell supports C, C++, C# and Java. Furthermore, the tool support a wide range of code coverage metrics: function, statement, decision, condition, multiple condition, and modified condition/decision. Windows and Linux are supported as host platforms.

In summary, this tool looks like a tool that is applicable to resource-limited embedded device. However, at least two points remain unclear. It is not clear what happens if the instrumentation changes the behavior of the program at runtime and affects a timing requirement. Furthermore, the exact overhead of the instrumented executable in comparison to the original executable is unclear.

6.1.2.3 BullseyeCoverage (Bullseye Testing Technology)

BullseyeCoverage[4] is a product of the company Bullseye Testing Technology. The tool instruments the program on the source code level and supports function coverage, and condition/decision coverage. The tool supports the languages C and C++. BullseyeCoverage claims that they support embedded targets and provide a list of supported embedded devices. Windows, Unix and Linux are the supported host operating systems.

[3]http://www.verifysoft.com/en_ctcpp.html
[4]http://www.bullseye.com/

6.1.2.4 LDRAcover (LDRA Software Technology)

LDRAcover[5] is a code coverage reporting tool from LDRA Software Technology. As far as we know, the tool inserts probes at the source code level. They support function, statement, decision, branch decision condition, branch condition combination, and multiple condition/decision. The following programming languages are supported: C, C++, Ada, and Java. Embedded targets are also supported. Unfortunately, we were not able to gather more detailed information about whether LDRAcover can handle resource-limited embedded targets or not. The tool works on several kinds of Windows and Linux platforms.

6.1.2.5 Test Coverage Tools (Semantic Designs)

The company Semantic Designs provides Test Coverage Tools[6]. This tool works with probes that are inserted at the source code level. Therefore, the tool is not dependent on a specific compiler. Semantic Designs claims that their tool has a very low overhead per *executed* probe, about one or two machine instructions. This must refer to overhead in terms of time.

With their probe based approach they can support statement coverage and branch coverage. Test Coverage Tools supports a long list of programming languages, for instance C (ANSI, MSVC6, GNU, C99), C++ (ANSI, MSVC6, MSVS2005-MSVS2012, GNU, and C++11), C#, Java and many more. Only Windows can be used as host operating system.

6.1.2.6 Tessy (Razorcat, Hitex)

Tessy[7] is a tool for testing embedded software. Tessy is from the company Razorcat Development GmbH. It supports condition, decision, multiple condition/decision, and multiple condition code coverage criteria. Ansi C , C++ and some target compiler extensions are the languages that are supported by the tool. Only Windows can be used as host operating system.

6.1.3 Summary of the tool evaluation

Each of the examined code coverage measurement tools focuses on slightly different aspects of code coverage. Therefore, they have different strengths and shortcomings. We

[5]http://www.ldra.com/index.php/en/products-a-services/ldra-tool-suite/ldracover
[6]http://www.semanticdesigns.com/Products/TestCoverage/
[7]http://www.razorcat.com/

		VectorCAST	Testwell	BullseyeCoverage	LDRAcover	Code Coverage Tools	Tessy	Our System
	Instrumentation	SC*	SC*	SC	SC*	SC	SC*	SC
Coverage	Function		√	√	√			√
	Statement	√	√		√	√		
	Decision	√	√		√	√	√	√
	Condition		√		√			
	Cond./Dec.			√	√			
	MC/DC	√	√		√		√	
	MCC		√		√		√	
Host OS	Windows	√	√	√	√	√	√	√
	Linux	√	√	√	√			
	Embedded Targets	++	++	+	+	++	+	√
Requirements	R1 Language Support	√	√	√	√	√	√	√
	R2 Memory Consumption							√
	R3 CC Criterion	√	√	√	√	√	√	√
	R4 Test Abortion Hand.							√
	R5 Total Time							√
	R6 Visual Report	√	√	√	√	√	√	√
	R7 Efficient Instr.	√	√			√		√

Table 6.1: Comparison of commercial code coverage tools.

are mainly interested in finding a tool for C that is able to handle resource-limited devices. Unfortunately we were not able to gather more clear information regarding the requirements in Section 1.2.1.

The companies of the tools Testwell and Code Coverage Tools, claim that their tool is able to handle resource-limited targets. Besides these two tools, the majority of the tools that state that they can work with embedded targets seems to be restricted to some specific, more powerful embedded devices. However, it is not perfectly clear how the tools deal with real-time embedded devices with very tight timing requirements. For instance, we could not find out how they handle the case that the instrumentation causes timing issues. It would be interesting to know if they handle such circumstances, and if so, how do they handle it.

Extensions to the languages are needed for efficient software development on some embedded devices. This also applies to the MRK-IIIe. Some tools provide the possibility for the user to implement language extensions. Unfortunately, it is difficult to say how much development overhead this is.

The mechanism with which one can read out the gathered code coverage information from the embedded target. Semantic Designs claims that for their tool it is possible to write customized extensions for a specific embedded device in order to provide an interface. Again it is very hard to estimate how difficult and time consuming it would be to write such an extension.

The more sophisticated an applied coverage criterion is, the more likely a timing constraints will fail and the more overhead in terms of executable size and memory consumption is needed. Therefore, it is hard to apply those metrics on very resource-limited systems.

In summary, we could identify only two tools on the market that are comparable to our solution, Testwell CTC++ Test Coverage Analyser from Verifysoft Technology and Code Coverage Tools from Semantic Designs. All other tools are not really or not obviously applicable to very small targets.

6.2 Related Research on Code Coverage for Embedded Systems

There are a lot of papers that propose code coverage tools. However, there are only a few that are applicable on embedded systems. There are tools that insert probes into the user source code, others instrument the object code, even hybrid approaches exist. A few of those tools work with simulators or emulators. In this section we discuss some of these approaches to show the differences to our methods. We group the tools into source instrumentation, binary instrumentation, and tools that use simulators or emulators.

6.2.1 Source Instrumentation

W. E. Wong et al. have proposed in [44] a tool that uses source code instrumentation. They describe a solution that works with embedded devices with Symbian/OMAP platforms only.

Another example of a tool that works with source code instrumentation has been proposed by Yong-Yoon et al. in [8]. It is a system for performance evaluation of embedded software. It consists of a code analyzer, testing agents, a data analyzer, and a report viewer. The tool is capable of analyzing code coverage of embedded software. They tested their approach with a strong ARM chip with Linux on it.

The existing research that is most similar to ours is probably [45]. The authors describe a coverage based testing technique for real-time embedded systems. This approach has been implemented into an automatic coverage tool named eXVantage. The authors identify the overhead resulting of inserting probes as one major issue of code coverage testing. This overhead becomes even more a problem for resource-limited embedded devices. To overcome this issue, the authors of [45] focus on minimizing the instrumentation overhead. In order to achieve this, they optimize three aspects. First, they record coverage information instead of a full execution trace. This saves memory. Second, they store the information directly in the memory instead in a file. This results in a reduced number of instructions per recorded probe. Third, they store the information in a binary format instead of a human readable format. The experiments of [45] showed that the instrumentation did not alter the behavior of the tested system. Therefore, they do not talk about how to handle this circumstance.

We use similar ideas for our approach. For instance, we also store just the coverage information. Furthermore, we need one single bit in the memory for each probe in the source code. It has to be noted that storing coverage information on files is not even an option in our resource-constrained devices, because the have no file system.

The authors of [45] tested their tool on embedded device with MIPS architecture with a reduced image of VxWorks, a Real-Time operating System (RTOS) requiring 4KB. Even if eXVantage works on resource-limited targets, at the moment it is limited to devices running VxWorks or Linux. In contrast, our tool needs one C macro in an inline function and an array as the whole framework. Hence, it is independent of embedded operating systems.

The authors of [45] also had to overcome a shared memory issue that is related to the minimal version of VxWorks and the original version of eXVantages. The minimal version of VxWorks uses a flat memory model whereas the original version of eXVantage uses shared memory for trace memory buffer maintenance. In contrast to [45], embedded systems using flat memory model are not an issue for our system.

In [25, 40, 41] dynamic or online instrumentation approaches have been proposed. Here the instrumentation is altered during the runtime. The idea is to remove probes that cannot bring additional information, for instance an already covered probe in an `if` branch. Our iterative technique of updating the instrumentation is more or less the same with the difference that we alter the instrumentation offline and recompile the source code. Note that removing probes at the runtime also produces timing overhead that can lead to an abortion of a test case. As far as we know these dynamic approaches do not handle this issue.

6.2.2 Binary Instrumentation

The majority of commercial code coverage tools focus on source code instrumentation, whereas in the literature there are also implementations that do binary instrumentation. Examples include PEBIL [23], PIN [38], and the method of Thane et al. [14]. PEBIL (PMaC's Efficient Binary Instrumentation Toolkit for Linux) is a static binary instrumentation toolkit for Linux on the x86/x86_64 platforms [23]. The system tries to produce efficiently instrumented object code.

Another binary instrumentation tool called PIN[8] was introduced in [38]. PIN is a tool for investigating workload characteristics. Another goal of PIN is to provide a tool for students to learn how software performs on hardware in a more practical way. Therefore, this tool provides methods to experiment with the characteristics of real application workloads in order to understand modern computer architectures. PIN is a tool from Intel Corporation[9] and works on Intels Itanium systems.

The authors of [14] propose a method to dynamically patch embedded software in order to measure coverage. They performed their experiments on a Motorola 68000 platform running the real-time operating system VxWorks 5.4[10]. In version 6.9, VxWorks achieves a footprint of 75KB with the small footprint profile[11]. In contrast to that, the authors of [45], which propose a source code approach based on eXVantage (see Section 6.2.1), state that they use a reduced image of VxWorks that has a footprint of only 4KB. They did not specify the version of VxWorks. It is not clear how much this approach depends on VxWorks.

As it can be seen, there are interesting developments in the literature. The three approaches we have outlined have one thing in common: they all focus on embedded systems that are more powerful than the embedded systems we are focusing on. PEBIL is for x86 Linux platforms, PIN is for Intels Itanium systems, and also the Motorola platform of the last approach seems to exceed the power of the embedded systems we are interested in. Therefore, none of the above tools is suitable to our problem of measuring code coverage information on embedded targets with highly limited resources.

[8]http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[9]http://www.intel.com
[10]http://www.windriver.com/products/vxworks.html
[11]http://www.windriver.com/products/product-notes/PN_VE_6_9_Platform_0311.pdf

6.2.3 Simulation and Emulation

There are also approaches that are depending on simulators or emulators. Wang et al. proposed in [42] a tool named SciSim (Source code instrumentation based Simulation).SciSim is a framework that combines source code instrumentation with microarchitecture simulators in order to model runtime interactions between embedded systems and their running software. They support widely used instruction set architectures (ISA) like PowerPC, ARM and SPARC.

An example for a tool that uses an emulator is COUVERTURE[12] from the Open-DO Initiative. This is a non-intrusive, virtualized execution environment.

In our case, emulation or simulation is not easily possible because our embedded devices have complex peripherals that cannot be emulated easily. Therefore, we have to find a solution where it is not necessary to emulate the target device. We want to measure code coverage on the original device.

6.2.4 Summary

There exists a lot of promising tools. Scientists are researching in different directions: intrusive, non-intrusive, source code instrumentation, object code instrumentation, simulation or emulation of targets, and hybrid approaches. Unfortunately, the majority of the research is focusing on powerful embedded systems, for instance, on embedded systems that are able to run specific embedded operating systems on it, or even Linux [8] [44]. Since operating systems occupy a lot of space in the storage of an embedded device, such systems are not applicable for resource-limited targets. To the best of our knowledge, it seems that embedded devices with highly limited resources and peripherals have not been addressed so far.

[12]http://www.open-do.org/projects/couverture/

Chapter 7

Conclusion and Outlook

Contents

7.1 Summary

At the beginning of this thesis we defined requirements (Section 1.2.1) that a code coverage system has to fulfill to be able to handle resource-limited embedded devices with tight real-time constraints. These requirements were used as a roadmap for the whole thesis. We used them as basis for the design of our code coverage measurement tool and we compared commercial tools with these requirements to get an overview of the state of the art in the software-development industry. We were not able to find a tool that fulfills all of our requirements.

In this thesis we tried two approaches to measure code coverage on a resource-limited device. One is based on binary instrumentation and the other is based on source code instrumentation. For both approaches we developed a prototypical implementation. The binary instrumentation approach is capable to measure function coverage only by patching return instructions. After a first evaluation we saw that it could be difficult to fulfill all our requirements with the binary instrumentation approach and changed the focus to the source code instrumentation approach.

We developed a code coverage measurement system based on source code instrumentation. This system is capable of handling embedded devices with very limited resources and tight real-time constraints. Very limited resources means that the memory capacity on the embedded device is very limited, for instance, we tested an embedded device with only up to one kilobyte of memory available for storing coverage data. We tested our code coverage system with four prototypical firmware applications of industrial use on an embedded device provided by NXP Semiconductors. These experimental results showed that our system fulfills all defined requirements. The system measures test suite adequacy with respect to a combination of function, and decision coverage criterion. It requires less than one kilobyte of RAM. For the tested firmware applications it needed less than 0.25 kilobyte. It needs only two instruction per inserted probe and only one bit of memory is needed to store if a probe is covered or not. The system prolonged the test time only to an extent so that it was still doable in an overnight task. It is able to gracefully handle test abortions caused by timing violations due to the instrumentation. We achieved this by removing the already covered probes from the instrumentation and started another test run. This iterative approach stops after the iteration where no new probes have been covered.

In this thesis we showed that it is possible to measure code coverage on embedded systems that are very resource-limited and have very tight real-time constraints. Nevertheless, our solution has some restrictions and there is still room for optimization. We discuss this in the next section.

7.2 Future Work

The presented results show that it is possible to measure code coverage on resource-limited embedded systems in a reasonable time. However, as we already mentioned in the previous section, there is still room for improvement.

After the second iteration the tool always performs a full test suite run. Therefore, a reduction of the number of iterations would improve the slowdown factor considerably. For instance, after the third iteration we got at least 93.32% of the possible overall coverage and at least 99.1% of the possible protocol coverage. Protocol II, III and IV reached 100% of the possible protocol coverage after the first full run of the test suite. Stopping after the succeeding iteration would half the worst slowdown factor of each protocol. The worst slowdown factor when stopping after the succeeding iteration is 4.35 (Protocol I - Execution Nr. 1) and the worst slowdown factor after the first full run of the test suite (iteration three) is only 2.64 (Protocol III - Execution Nr. 1). Furthermore, after this iteration the protocol coverage did not increase in a subsequent iteration. This was the case for each coverage run, no matter which protocol.

Another idea for improvement is reducing the number of target memory read outs. Such a read out retrieves the list of covered instruments from the target. On average, we have a slowdown factor of 1.35 per test suite run (one run during one iteration) due to the target memory read outs. For our experiments, we retrieved the coverage data from the target after every power down of the target. The reason for this is that after a specific amount of time we cannot guarantee that the data in the RAM is still valid. If we do this only at every tenth power down we could reduce this overhead tremendously. Maybe a timer is needed to ensure that the RAM remains consistent. For instance, we could perform at least one readout if the last one was more than a minute ago.

Since it seems that the test suite does not focus on auxiliary files, instrumenting only protocol related files and not auxiliary files should also bring a reduction of the number of iterations. Note that in our experiments it would bring the same reduction as stopping after the succeeding iteration.

Probes that are only covered due to nondeterministic timing behavior should be covered in dedicated test cases for to reasons. First, probes should not be covered by luck. Second, it is not good to run our system an iteration longer for only one probe.

Literature provides a lot of methods to reduce the number of probes for measuring code coverage [4, 22, 24, 36, 37]. In [40] dominator tree information is used to reduce the number of needed probes for basic block coverage. In [1], Agrawal proposed a super/mega block method to reduce the number of probes required by the basic block method. Therefore, it is still room to reduce the number of probes.

Another point for future work is to extend the system with more powerful test suite adequacy criteria, for instance condition coverage. Of course the shortcoming of not instrumenting `do-while` loops should also be corrected. Both improvements require to modify the source code in a more invasive way than now, for instance modifying expressions. At some point also the preprocessor statements should be considered.

In summary, we developed a practical approach and tool for measuring code coverage on very resource-limited devices. This is witnessed by the fact that NXP Semiconductors already uses this tool in an industrial context. However, as discussed above, there is still room for improvements and future work.

Appendix A

C Statement Definitions

Here we give the syntax definition of statements in C. We extracted this definitions from the C Standard [15]. For more definitions and details please refer to the the C Standard [15].

A.1 Labeled statements

Syntax

labeled-statement:

  identifier : statement

  `case` *constant-expression : statement*

  `default` *: statement*

A.2 Compound statement

Syntax

compound-statement:

  { *block-item-list*$_{opt}$ **}**

block-item-list

  block-item

  block-item-list block-item

block-item:

  declaration

  statement

A.3 Expression statements

Syntax

expression-statement:

expression$_{opt}$;

A.4 Selection statements

Syntax

selection-statement:

`if` (*expression*) *statement*

`if` (*expression*) *statement* `else` *statement*

`switch` (*expression*) *statement*

A.5 Iteration statements

Syntax

iteration-statement:

`while` (*expression*) *statement*

`do` *statement* `while` (*expression*) ;

`for` (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*

`for` (*declaration* *expression*$_{opt}$; *expression*$_{opt}$) *statement*

A.6 Jump statements

Syntax

jump-statement:

`goto` *identifier* ;

`continue` ;

`break` *expression*$_{opt}$;

Appendix B

Selected Source Code Extracts from the Implementation

Listing B.1: Source code of the Prettifier.

```
void Prettifier::prettify(SrcFile *file)
{
  vector<Token*>::iterator it = file->_tokens.begin();
  while(it != file->_tokens.end()) {
    condStmtSwitch(file, it);
  }
}

void Prettifier::condStmtSwitch(SrcFile *file,
                                vector<Token*>::iterator &it)
{
  if((*it)->_type == Tif) {
    it++;
    handleCondStmt(file, it);
    handleElse(file, it);
  } else if((*it)->_type == Twhile ||
            (*it)->_type == Tfor) {
    it++;Co
    handleCondStmt(file, it);
  } else if((*it)->_type == Tdo) { // do while
    it++;
    handleBlockAfterCond(file, it);
    jumpOverWhiteSpace(it);
    if((*it)->_type != Twhile) {
      puts("ERROR:␣no␣while␣after␣do!");
      exit(-1);
    }
    it++;
  } else if((*it)->_type == Tcase) {
    it++;
    getNextDotDotPos(it);
    it++;
  } else if((*it)->_type == TOBrace) { // a block { ... } without an
                                       // if, for, while, etc
      it++;
      jumpOverBlock(file, it);
  } else {
    it++;
  }
}

void Prettifier::handleCondStmt(SrcFile *file,
                                vector<Token*>::iterator &it)
{
  getNextOParPos(it);
  jumpToRelatedCPar(it);
  handleBlockAfterCond(file, it);
}

void Prettifier::handleElse(SrcFile *file, vector<Token*>::iterator &it)
{
  vector<Token*>::iterator backup_it = it;
  jumpOverWhiteSpace(it);
  if((*it)->_type == Telse) {
    it++;
    handleBlockAfterCond(file, it);
  } else {
    it = backup_it;
    file->insertToken(it, Telse, "else");
    file->insertToken(it, TOBrace, "{");
    file->insertToken(it, TCBrace, "}");
  }
}

void Prettifier::handleBlockAfterCond(SrcFile *file,
                                      vector<Token*>::iterator &it)
{
  jumpOverWhiteSpace(it);
  if((*it)->_type == TOBrace) { // the block/stmt is covered in braces
    it++;
    jumpOverBlock(file, it);
  } else { // insert braces
    file->insertToken(it, TOBrace, "{");
    jumpOverStatement(file, it);
    file->insertToken(it, TCBrace, "}");
  }
}

void Prettifier::jumpOverStatement(SrcFile *file,
                                   vector<Token*>::iterator &it)
{
  while((*it)->_type != TPtVirg ) {
    if((*it)->_type == Tif) {
      it++;
      handleCondStmt(file, it);
      handleElse(file, it);
      return;
    }
    else if((*it)->_type == Twhile ||
            (*it)->_type == Tfor) {
      it++;
      handleCondStmt(file, it);
      return;
    }
    it++;
  }
  it++;
}

void Prettifier::jumpOverBlock(SrcFile *file, vector<Token*>::iterator &it)
{
  while((*it)->_type != TCBrace) {
    condStmtSwitch(file, it);
  }
  it++;
}
```

Bibliography

[1] Hiralal Agrawal. Dominators, super blocks, and program coverage. In *Conference Record of the 21st Symposium on Principles of Programming Languages*, pages 25–34. ACM, 1994.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.

[5] Michael Barr and Anthony Massa. *Programming embedded systems 2nd Edition-with C and GNU development*. O'Reilly, 2006.

[6] Benjamin M. Brosgol. Non-intrusive code coverage for safety-critical software. `http://embedded-computing.com/articles/non-intrusive-code-coverage-safety-critical-software/`, November 2011. Accessed: 2014-04-25.

[7] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9:193–200(7), 1994.

[8] Yong-Yoon Cho, Jong-Bae Moon, and Young-Chul Kim. A system for performance evaluation of embedded software. In *International Conference on Computational Intelligence*, ICCI 2004, pages 69–72. International Computational Intelligence Society, 2004.

[9] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the 8th International Conference on Software Engineering*, ICSE, pages 244–251. IEEE Computer Society, 1985.

[10] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, 1989.

[11] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2, 1995.

[12] Intel Corporation. *Intel Itanium 2 processor reference manual: For software development and optimization*. Intel Corporation, May 2004.

[13] Edsger W. Dijkstra. *Notes on Structured Programming*. Technological University Eindhoven Netherlands, 1970.

[14] Mathias Ekman and Henrik Thane. Dynamic patching of embedded software. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 337–346. IEEE Computer Society, 2007.

[15] International Organization for Standardization. ISO C Standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[16] Robert L. Glass. Loyal opposition - frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):112–111, 2001.

[17] Richard G. Hamlet. Theoretical comparison of testing methods. In *Symposium on Testing, Analysis, and Verification*, pages 28–37. ACM, 1989.

[18] Steve Heath. *Embedded systems design*. Newnes, 2002.

[19] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, 1975.

[20] Susanne Kandl and Sandeep Chandrashekar. Reasonability of MC/DCfor safety-relevant software implemented in programming languages with short-circuit evaluation. In *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*. IEEE Proceedings, 2013.

[21] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[22] Donald E. Knuth and Francis R. Stevenson. Optimal measurement points for program frequency counts. *BIT Numerical Mathematics*, 13(3):313–322, 1973.

[23] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: Efficient static binary instrumentation for linux. In *IEEE International Symposium*

on Performance Analysis of Systems and Software, ISPASS'10, pages 175–183. IEEE Computer Society, 2010.

[24] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[25] J. Jenny Li, David M. Weiss, and Howell Yee. An automatically-generated run-time instrumenter to reduce coverage testing overhead. In *Proceedings of the 3rd International Workshop on Automation of Software Test*, AST '08, pages 49–56. ACM, 2008.

[26] ARM Limited. *Embedded Trace Macrocell ETMv1.0 to ETMv3.5, Architecture Specification*. ARM Limited, 2011.

[27] BCC Research LLC. Embedded systems: Technologies and markets, report ift016d. `http://www.bccresearch.com/report/IFT016D.html`, 2012. Accessed: 2014-04-25.

[28] Michael R. Lyu et al. *Handbook of software reliability engineering*, volume 3. IEEE Computer Society, 1996.

[29] Michael R. Lyu, Zubin Huang, Sam K. S. Sze, and Xia Cai. An empirical study on testing and fault tolerance for software reliability engineering. In *14th International Symposium on Software Reliability Engineering*, ISSRE, pages 119–132. IEEE Computer Society, 2003.

[30] Brian Marick. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*, pages 16–18, 1999.

[31] Aditya P. Mathur. *Foundations of software testing: Fundamental Algorithms and Techniques*. Pearson Education India, 2013.

[32] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63, 1963.

[33] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing, 3rd Edition*. John Wiley & Sons, 2011.

[34] Tammy Noergaard. *Embedded systems architecture, 2nd Edition - a comprehensive guide for engineers and programmers*. Newnes, 2012.

[35] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988.

[36] Robert L. Probert. Optimal insertion of software probes in well-delimited programs. *IEEE Transactions on Software Engineering*, 8(1):34–42, 1982.

[37] C. V. Ramamoorthy, K. H. Kim, and W. T. Chen. Optimal placement of software monitors aiding systematic testing. *IEEE Transactions on Software Engineering*, 1(4):403–411, 1975.

[38] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM.

[39] Alex Shye, Matthew Iyer, Vijay Janapa Reddi, and Daniel A. Connors. Code coverage testing using hardware performance monitoring support. In *Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging (AADEBUG)*, pages 159–163. ACM, 2005.

[40] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2002, pages 86–96. ACM, 2002.

[41] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient online computation of statement coverage. *Journal of Systems and Software*, 78(2):146–165, 2005.

[42] Zhonglei Wang, Antonio Sanchez, and Andreas Herkersdorf. Scisim: A software performance estimation framework using source code instrumentation. In *Proceedings of the 7th International Workshop on Software and Performance (WOSP)*, WOSP '08, pages 33–42. ACM, 2008.

[43] Elecia White. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly, 2011.

[44] W. Eric Wong, Sharath Rao, John Linn, and James Overturf. Coverage testing embedded software on symbian/omap. In *Proceedings of the Eighteenth International Conference on Software Engineering*, SEKE'06, pages 473–478, 2006.

[45] Xianming Wu, J. Jenny Li, David M. Weiss, and Yann-Hang Lee. Coverage-based testing on embedded systems. In *Proceedings of the 2nd International Workshop on Automation of Software Test*, AST '07, pages 31–36. IEEE Computer Society, 2007.

[46] Hong Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255, 1996.