Test Suite Coverage Measurement and Reporting for Testing an Operating System without Instrumentation

Hermann Felbinger
Virtual Vehicle Research Center
Graz, Austria
hermann.felbinger@v2c2.at

Joel Sherrill
OAR Corporation
Huntsville, AL
joel.sherrill@oarcorp.com

Gedare Bloom
Dept. of Computer Science
Howard University
Washington, DC
gedare.bloom@howard.edu

Franz Wotawa
Institute for Software Technology
Graz University of Technology
Graz, Austria
wotawa@ist.tugraz.at

Abstract

Measuring the coverage of a test suite provides common metrics to assess the quality of a test suite. In safety-critical applications, as in the domains of avionics and automotive, complete coverage is required for certification. Usual approaches to measure the coverage require instrumentation of the source code or the object code of the system under test to obtain processable execution traces. However, instrumentation might change the behavior of the system under test. In this paper we show an approach to measure the coverage of a test suite and to generate human-readable reports without instrumentation of the system under test. As a system under test we use an operating system. Our approach is based on the execution traces obtained from an instrumented QEMU CPU emulator. We use this emulator to execute the operating system and the test cases. From the execution of the test cases we obtain execution traces. We provide a framework to map these execution traces back to the source code and to generate a detailed report exposing execution and branching (taken/not taken) information at the assembly language level and source code level.
To evaluate our approach we generate coverage reports for the RTEMS real time operating system. We provide detailed coverage results for RTEMS running on different CPUs in this paper. Coverage of a test suite can be used by operating system developers to assess test suite quality and guide test case creation. Our approach is due to the lack of instrumentation of source code and object code broadly applicable for development of embedded systems applications.

1 Introduction

Different industry and country specific standards specifying functional safety requirements for software-based systems exist. The ECSS-E-ST-40C [3] Standard defines the principles and requirements applicable to space software engineering, ISO 26262 [5] is considered as state-of-the-art in automotive electric and electronic systems, and DO-178C [4] is the document by which certification authorities approve all software-based avionics systems. These are three examples of standards which software of systems in the respective domain have to satisfy to obtain certification. All of these standards have in common that testing the software is quantified in code coverage where the standards require different coverage metrics depending on the criticality level of the system under test (SUT).
Code coverage is the percentage of the software artifacts that have been executed (covered) by the test suite [3]. Software artifacts are e.g. statements, conditions, and decisions within the source code, or instructions and branches within the object code.
Three different methods exist to obtain code coverage:

1. **Instrument Source Code** to compile the SUT including instrumentation code that is used to generate traces during execution.
2. **Instrument Object Code** by a compiler to generate traces during execution.
3. **Obtain execution traces from the execution platform** without instrumentation.

The differences of the three methods are visualized in Figure 1. The methods instrumenting the source and object code are labeled as conventional approach. In this work we introduce a tool chain based on the third method labeled as approach by virtualization in Figure 1. As execution platform we utilize the CPU emulator QEMU [7] which virtualizes the target CPU and dynamically translates object code into native host instructions.
Different versions of QEMU [6] and also other CPU emulators, e.g. TSIM [8], Skyeye [9], etc. exist which produce a debug log of the executed instructions when an executable is running. This debug log is the trace information which is analyzed to identify branch instructions and to determine whether the branch was taken or not taken. In this work we use an extended version of QEMU [7] that produces execution traces which are analyzed to obtain coverage information. This QEMU version was extended within the COUVERTURE project [2] and therefore named Couverture QEMU.

The tool we introduce here, to process coverage information and generate the coverage reports, was named *covoar*. *Covoar* was designed to analyze code coverage as automated as possible. Because *covoar* supports performing coverage analysis using a set of different CPU emulators, *covoar* has to solve issues caused by different formats of execution information. Each source producing execution information, e.g. emulator, hardware debugger, etc. may produce the information in a different format. *Covoar* converts the execution information into an internal representation where currently formats produced by TSIM, Skyeye, QEMU, and Couverture QEMU emulators are supported. From the internal representation *covoar* merges execution information for a set of methods of interest.
The output produced by *covoar* is actually a set of HTML and simple ASCII files that give a developer the necessary information to quickly determine the current status of the code coverage and enough information to determine the location of uncovered code. The location of uncovered code is determined by using the source to object code mapping information extracted from the debugging information contained in the SUT. The resolution of uncovered code does not simply translate into additions to the test suite. Often the resolution points to improvements or changes to the analyzed code. *Covoar* is invoked once the execution of the test suite is complete.

Since the aforementioned standards demand that all requirement-based test cases must be executed on the target platform for coverage purposes a final verification on the target platform is required. Applying the approach by virtualization for coverage analysis, this final verification can be simplified, because no instrumentation was added to the executables. Therefore the final verification entails rerunning the test cases and showing that the results are the same as on the emulator.

Covoar is an open source tool and freely available, developed within the RTEMS project [1]. In this work we use the RTEMS operating system (OS) as SUT and provide two coverage reports resulting from executing the test suite, provided by the RTEMS project, within the Couverture QEMU emulator. As target platforms for these two examples we decided to use Intel 80386 and SPARC. The reports currently contain information about instruction and branch coverage.

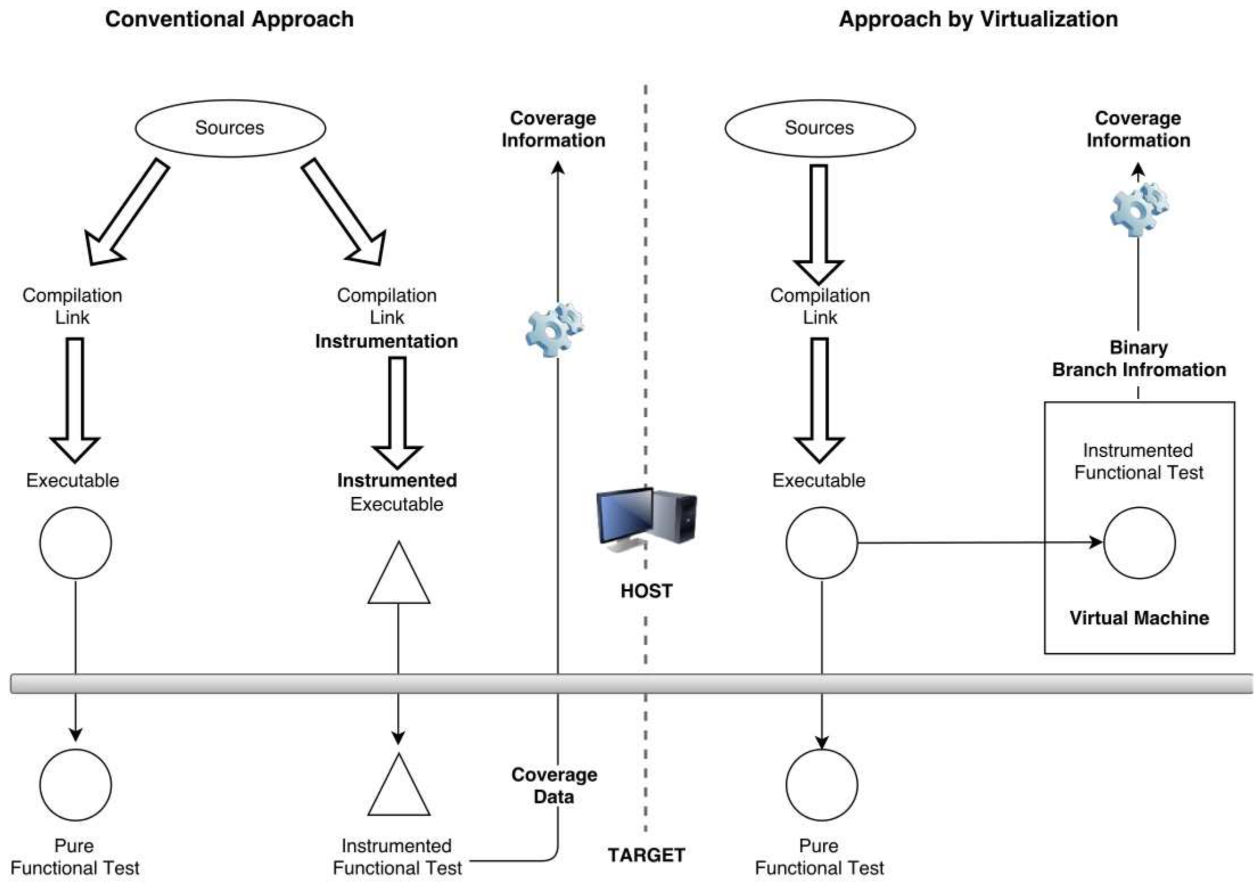


Figure 1: Approaches to extract code coverage information [2].

2 Preliminaries

In this work we use a CPU emulator to obtain execution traces from which we generate a code coverage report. We obtain these execution traces from the emulator without instrumenting our SUT and analyze their coverage regarding different coverage metrics.

2.1 CPU emulator QEMU

QEMU is an open source CPU emulator. As an emulator QEMU enables a host computer system to behave like another computer, called guest. Therefore QEMU enables a host computer to run software which was designed for the guest system.
All versions of QEMU can be configured to produce a debug log of the instructions executed while an executable is running. An extended version of QEMU is Couverture QEMU that produces a trace log which is denser than the debug log and represents a superset of the execution information contained in the debug log. This information can be analyzed to identify branch instructions and to determine whether the branch was taken or not taken.

The extensions for Couverture QEMU were developed within the COUVERTURE project. The COUVERTURE project had the objectives to produce a coverage analysis tool-set with the ability to generate artifacts that allow the tools to be used for safety-critical software projects. Beyond the production of a tool-set an important goal was to raise awareness and interest about safety-critical and certification issues. In the COUVERTURE project the front-end, that is used to analyze execution information and generating coverage reports including source to object code mapping, is only available for programs written in the programming language Ada [10].

The QEMU emulator and therefore also Couverture QEMU dynamically translates object code into native host instructions. As a result test suites typically execute faster than on the actual target hardware. To perform validation campaigns within the

space industry emulators must be qualified before they can be used for credit. To our knowledge TSIM, a commercial product from Gaisler Research, is such a qualified emulator.
Despite the non-intrusive extraction of execution information where the SUT is not instrumented, an emulator enables levels of debugging and testing that are not readily available on real hardware. An emulator can offer [11]:

- **Greater availability:** Early production hardware is often available in very limited quantities or is expensive. The emulator allows a greater number of embedded software developers to access the tools they need to write and test software.
- **More control:** The emulator can be easily stopped, reconfigured, and rerun. The system state can be saved and restored. This is not easy on real hardware.
- **Increased debugging ability:** The emulator provides a debugging environment that is not possible to create on real hardware. The emulator can show internal state of devices revealing information that cannot be captured with a logic probe. The emulator also provides more debugging control (with the use of breakpoints) that is not possible on real hardware.
- **Increased stability:** Early prototype hardware may have bugs or be unstable. The emulator enables comparative testing on early prototype hardware so that hardware bugs are easier to identify.
- **Facilitate Testautomation:** An emulator allows transferring the SUT into a certain state before the execution starts which facilitates automatically execution of tests.

2.2 Code Coverage Metrics

Code coverage or also structural coverage is used as a measure to analyze which parts of an SUT are tested. Depending on the level of the language in which the SUT is represented we divide structural coverage into two different measures. First we use object code coverage for a lower level language (object code), second we use source code coverage for higher level languages (C, C++, etc.). As shown in [12] object code coverage and source code coverage are not equivalent. Therefore, since source code coverage is required for certifications in the aforementioned domains of automotive, avionics, etc., we treat both of them in this work.

Object Code Coverage includes two measures: first measure is the proportion of executed instructions to existing instructions, second is the proportion of executed branches to existing branches within an SUT.

Source Code Coverage includes three simple measures, and combinations of them subsuming the simple ones. The simple measures are the proportion of executed statements to existing statements, the proportion of executed conditions to existing conditions, and the proportion of executed decisions to existing decisions within an SUT. The only combination we use in this work is the modified condition / decision coverage (MC/DC). MC/DC confirms that every point of entry and exit was invoked at least once, every condition in a decision has been taken on all possible outcomes at least once, and each condition has been shown to independently affect the overall decision outcome.

Detailed introductions of source and object code coverage can be found in [12]. In the aforementioned standards we found inconsistencies in the use of branch and decision coverage. Where DO-178C and ECSS-E-ST-40C require decision coverage, ISO 26262 requires branch coverage. Due to the definition of *decision* in the position paper [13] we assume branch coverage used in ISO 26262 to be equivalent to decision coverage.

3 Covoar

The tool *covoar* has been developed as part of the RTEMS project to analyze the coverage of the test suite created to test the RTEMS OS. Traditionally, code coverage analysis has been performed by instrumenting the source code or object code or by using special hardware to monitor the instructions executed. The guidelines for the RTEMS code coverage effort were to use existing tools and to avoid altering the code to be analyzed. This was accomplished by using a CPU emulator that provides execution information. This information is processed by *covoar* to determine which instructions are executed.

Covoar is **not** restricted to be used only in combination with the RTEMS operating system. *Covoar* is broadly applicable for development of embedded systems applications.

Covoar takes the execution information, provided as execution traces, from a CPU emulator, marks the corresponding symbols, which are extracted from the execution traces in the SUTs symbol table, as executed, and generates a coverage report. This report also contains information of executed and not executed source code which is created by using the debug information in DWARF format [16] within the SUT. The symbol table from the SUT can be used to analyze coverage when executing a test suite because a set of object code to be analyzed is the same in all tests and linked to the same address range. *Covoar* is a next generation program which takes a list of symbols and accounts for them being at different addresses in different tests. It knows the sizes and offsets of instructions are the same, so it can merge the coverage information of multiple programs using the same method.

Covoar requires a CPU emulator which captures execution information and has a board support package, which, e.g., for RTEMS, can be found in [17]. Currently *covoar* supports processing the execution information formats from the CPU emulators QEMU, Couverture QEMU, TSIM, and Skyeye.
Further *covoar* requires installation of the GNU Binutils [18]. From the GNU Binutils *covoar* uses *nm* to obtain symbol tables, *objdump* to disassemble the SUT, and *addr2line* to associate addresses from executables to filename and line number of source code.

3.1 Execution Traces

In this work we process execution traces produced by Couverture QEMU. The execution traces appear in a binary output file in a format specified within the COUVERTURE project. Two kinds of execution traces are relevant in this work:

Summary traces: The output identifies the address ranges of the instructions that were executed, and for conditional branches, which branch(es) was (were) taken. The output data has bounded size (actually linear with respect to object program size), as it only reveals which instructions/branches were executed and not the entire execution history.

Full historical traces for specified address ranges: In addition to indicating the instructions that were executed, the output shows which branch was taken at each evaluation of the relevant conditional expressions. The size of the output data depends on the execution history.

3.2 Coverage Report Generator

The output of *covoar* is a detailed report consisting of HTML and ASCII files. Currently these reports contain information about branch and instruction coverage. Further the reports expose execution and branching (taken/not taken) information at the assembly language level and high level language level. The execution and branching information is shown as highlighted code which is directly linked within the report.
The report contains an overview summarizing the output of the coverage analysis, a detailed coverage report containing links from symbols to source code, a branch report containing links from symbols to source code, a direct link to the annotated assembly, an overview of analyzed symbols containing the sizes in byte, and a list of explanations that were not found for this coverage analysis. Explanations allow senior developers to analyze coverage gaps and write guidance for addressing the gap.

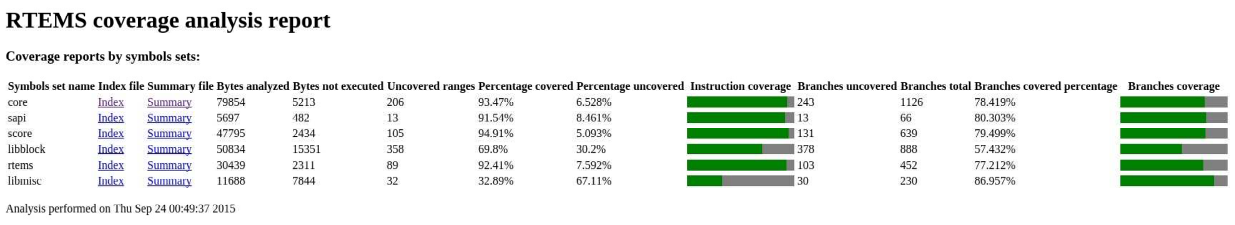
4 Experimental Results

Here we show two coverage reports obtained from executing the test suites for the RTEMS OS within Couverture QEMU. We built RTEMS 4.11 with an equivalent configuration for a PC386 and a LEON2 CPU. The executed test suite contained 421 test cases.

4.1 Coverage Report for PC386

To execute the test cases we built RTEMS and the board support packages (BSP) for a PC386 CPU. After a single execution of all 421 test cases, where we set a timout of 180 seconds, 399 test cases passed, 1 failed, and 21 timed out. The coverage report, generated after the execution of the test cases, presents the results as shown in Figure 2. Figure 2 provides a summary of the instruction and branch coverage for six analyzed libraries from RTEMS.

In Figure 3 a list of uncovered symbols from the *core* library is shown. These symbols are already mapped to the source files which are annotated with *executed/not executed* and *taken/not taken* statements as shown in Figure 4.



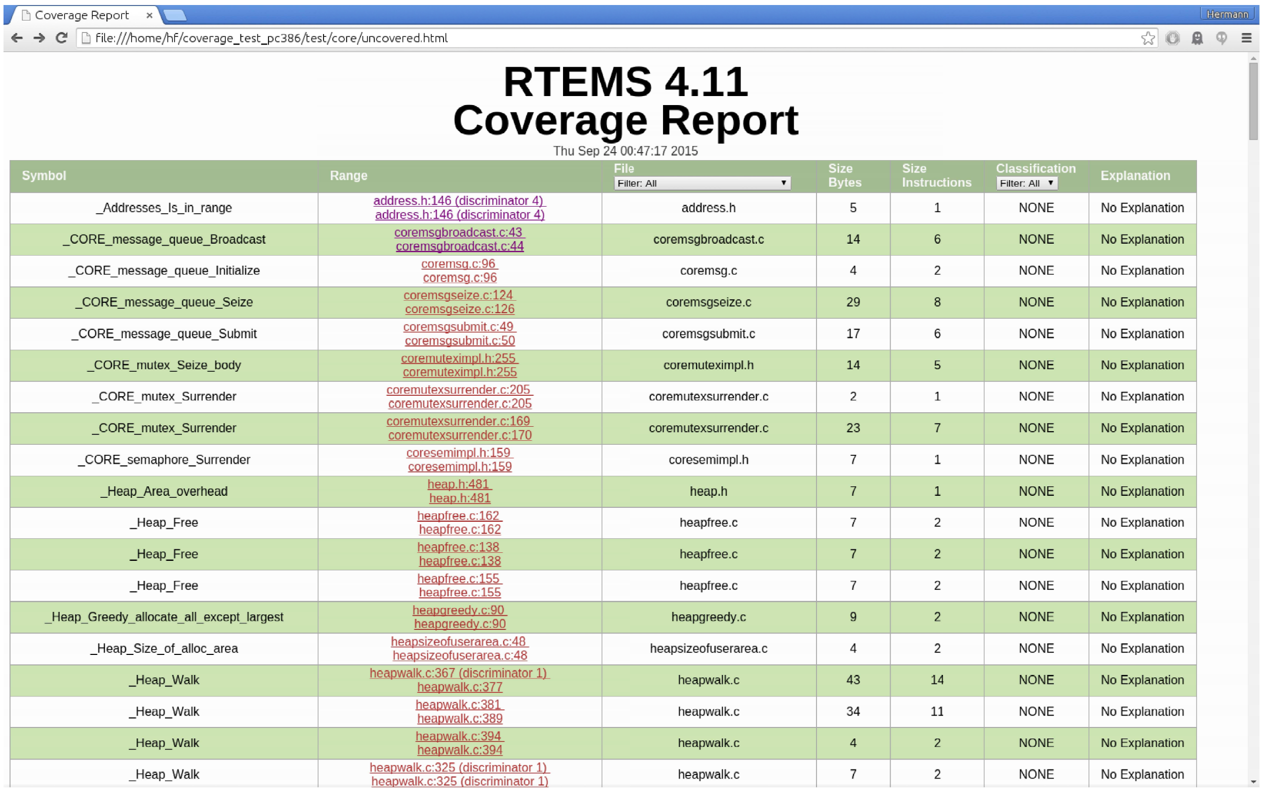
RTEMS coverage analysis report

Coverage reports by symbols sets:

Symbols set name	Index file	Summary file	Bytes analyzed	Bytes not executed	Uncovered ranges	Percentage covered	Percentage uncovered	Instruction coverage	Branches uncovered	Branches total	Branches covered percentage	Branches coverage
core	Index	Summary	79854	5213	206	93.47%	6.528%	🟩	243	1126	78.419%	🟩
sapi	Index	Summary	5697	482	13	91.54%	8.461%	🟩	13	66	80.303%	🟩
score	Index	Summary	47795	2434	105	94.91%	5.093%	🟩	131	639	79.499%	🟩
libblock	Index	Summary	50834	15351	358	69.8%	30.2%	🟩	378	888	57.432%	🟩
rtems	Index	Summary	30439	2311	89	92.41%	7.592%	🟩	103	452	77.212%	🟩
libmisc	Index	Summary	11688	7844	32	32.89%	67.11%	🟩	30	230	86.957%	🟩

Analysis performed on Thu Sep 24 00:49:37 2015

Figure 2: Overview of coverage results for PC386 CPU.



Coverage Report
Hermann
file:///home/hf/coverage_test_pc386/test/core/uncovered.html

RTEMS 4.11
Coverage Report

Thu Sep 24 00:47:17 2015

Symbol	Range	File Filter: All	Size Bytes	Size Instructions	Classification Filter: All	Explanation
_Addresses_Is_in_range	address.h:146 (discriminator 4) address.h:146 (discriminator 4)	address.h	5	1	NONE	No Explanation
_CORE_message_queue_Broadcast	coremsgbroadcast.c:43 coremsgbroadcast.c:44	coremsgbroadcast.c	14	6	NONE	No Explanation
_CORE_message_queue_Initialize	coremsg.c:96 coremsg.c:96	coremsg.c	4	2	NONE	No Explanation
_CORE_message_queue_Seize	coremsgseize.c:124 coremsgseize.c:126	coremsgseize.c	29	8	NONE	No Explanation
_CORE_message_queue_Submit	coremsgsubmit.c:49 coremsgsubmit.c:50	coremsgsubmit.c	17	6	NONE	No Explanation
_CORE_mutex_Seize_body	coremuteximpl.h:255 coremuteximpl.h:255	coremuteximpl.h	14	5	NONE	No Explanation
_CORE_mutex_Surrender	coremutexsurrender.c:205 coremutexsurrender.c:205	coremutexsurrender.c	2	1	NONE	No Explanation
_CORE_mutex_Surrender	coremutexsurrender.c:169 coremutexsurrender.c:170	coremutexsurrender.c	23	7	NONE	No Explanation
_CORE_semaphore_Surrender	coresemimpl.h:159 coresemimpl.h:159	coresemimpl.h	7	1	NONE	No Explanation
_Heap_Area_overhead	heap.h:481 heap.h:481	heap.h	7	1	NONE	No Explanation
_Heap_Free	heapfree.c:162 heapfree.c:162	heapfree.c	7	2	NONE	No Explanation
_Heap_Free	heapfree.c:138 heapfree.c:138	heapfree.c	7	2	NONE	No Explanation
_Heap_Free	heapfree.c:155 heapfree.c:155	heapfree.c	7	2	NONE	No Explanation
_Heap_Greedy_allocate_all_except_largest	heapgreedy.c:90 heapgreedy.c:90	heapgreedy.c	9	2	NONE	No Explanation
_Heap_Size_of_alloc_area	heapsizeofuserarea.c:48 heapsizeofuserarea.c:48	heapsizeofuserarea.c	4	2	NONE	No Explanation
_Heap_Walk	heapwalk.c:367 (discriminator 1) heapwalk.c:377	heapwalk.c	43	14	NONE	No Explanation
_Heap_Walk	heapwalk.c:381 heapwalk.c:389	heapwalk.c	34	11	NONE	No Explanation
_Heap_Walk	heapwalk.c:394 heapwalk.c:394	heapwalk.c	4	2	NONE	No Explanation
_Heap_Walk	heapwalk.c:325 (discriminator 1) heapwalk.c:325 (discriminator 1)	heapwalk.c	7	2	NONE	No Explanation

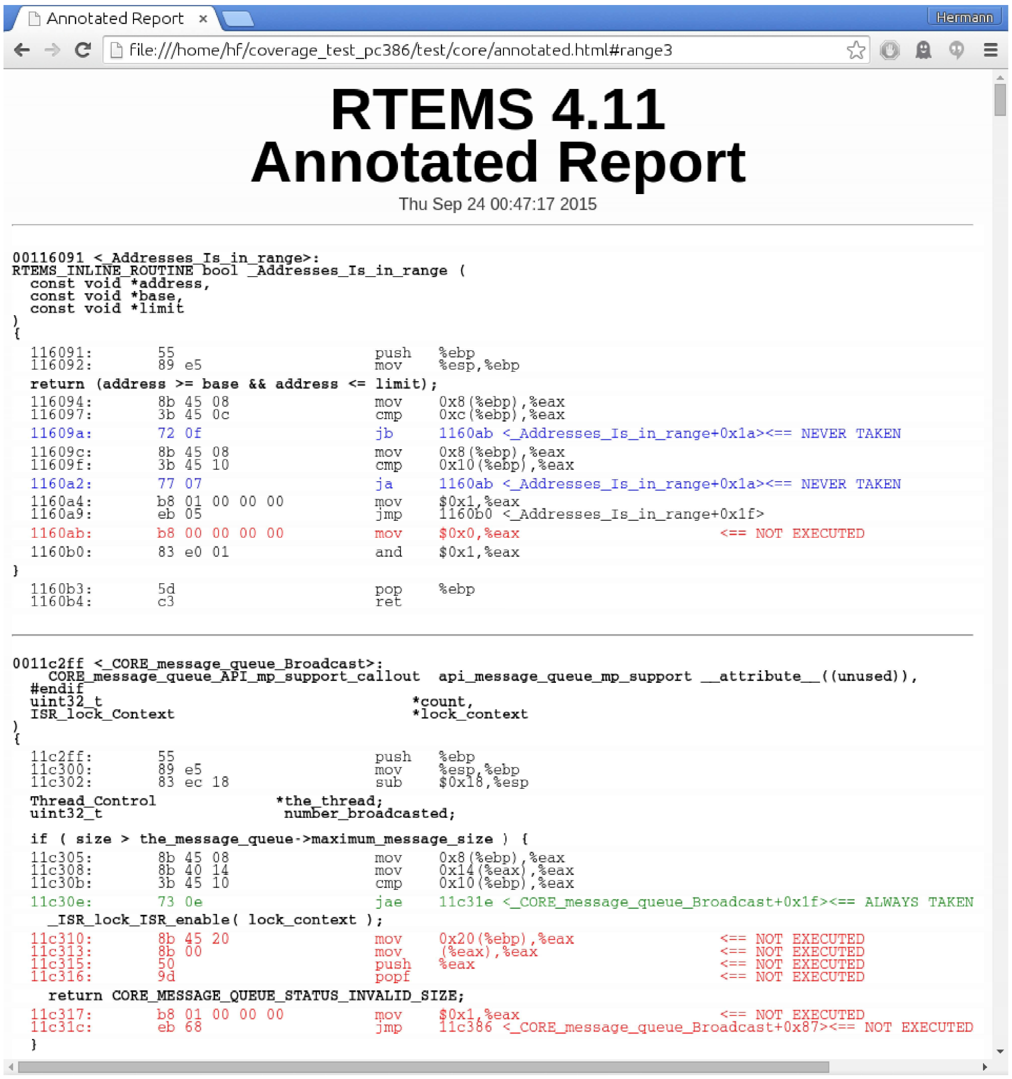
Figure 3: Uncovered symbols with links to the annotated source code.

4.2 Coverage Report for LEON2

To execute the test cases we built RTEMS and the board support packages (BSP) for a LEON2 CPU. After a single execution of all 421 test cases, where we set a timout of 180 seconds, 409 test cases passed, 10 failed, and 2 timed out. The coverage report, generated after the execution of the test cases, presents the results as shown in Figure 5. Figure 5 provides a summary of the instruction and branch coverage for six analyzed libraries from RTEMS.

5 Related Work

In this section we provide a selection of tools and tool suites related to the tool chain introduced in this work. In [12] the authors introduce a tool chain very similar to the tool chain from this work. The authors of [12] are developers of the extended QEMU, namely Couverture QEMU. The main difference of their work to ours is that they use as a front end a coverage analysis tool supporting code coverage for Ada whereas we support C and C++. Actually our approach is language independent since it relies on nm, addr2line, and objdump. Those handle the language specifics. In this work we focus on C and C++. Also within the COUVERTURE project the authors of [14] provided support for Objective CAML. In [19] the authors describe the application of an open-source tool chain including a tool to analyze coverage. They use Couverture QEMU to produce execution traces and a front end for C code, named XCOV. Unfortunately no further documen-


Annotated Report
file:///home/hf/coverage_test_pc386/test/core/annotated.html#range3
Hermann

RTEMS 4.11
Annotated Report
Thu Sep 24 00:47:17 2015

```
00116091 <_Addresses_Is_in_range>:
RTEMS_INLINE_ROUTINE bool _Addresses_Is_in_range (
  const void *address,
  const void *base,
  const void *limit
)
{
  116091:       55                      push   %ebp
  116092:       89 e5                   mov    %esp,%ebp
  return (address >= base && address <= limit);
  116094:       8b 45 08                mov    0x8(%ebp),%eax
  116097:       3b 45 0c                cmp    0xc(%ebp),%eax
  11609a:       72 0f                   jb     1160ab <_Addresses_Is_in_range+0x1a><== NEVER TAKEN
  11609c:       8b 45 08                mov    0x8(%ebp),%eax
  11609f:       3b 45 10                cmp    0x10(%ebp),%eax
  1160a2:       77 07                   ja     1160ab <_Addresses_Is_in_range+0x1a><== NEVER TAKEN
  1160a4:       b8 01 00 00 00          mov    $0x1,%eax
  1160a9:       eb 05                   jmp    1160b0 <_Addresses_Is_in_range+0x1f>
  1160ab:       b8 00 00 00 00          mov    $0x0,%eax                  <== NOT EXECUTED
  1160b0:       83 e0 01                and    $0x1,%eax
}
  1160b3:       5d                      pop    %ebp
  1160b4:       c3                      ret
```

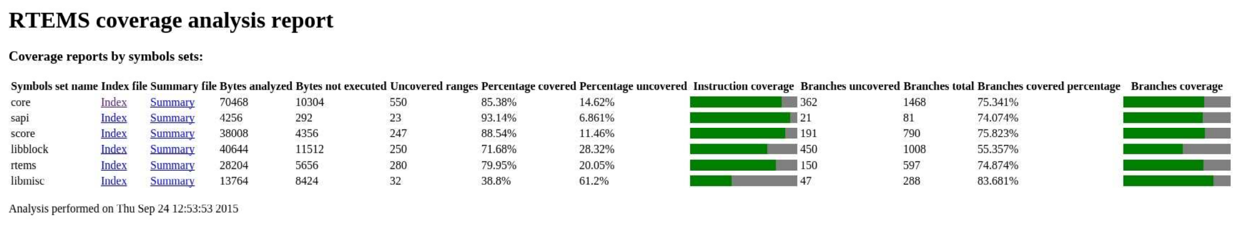
```
0011c2ff <_CORE_message_queue_Broadcast>:
    CORE_message_queue_API_mp_support_callout  api_message_queue_mp_support __attribute__((unused)),
  #endif
  uint32_t                                   *count,
  ISR_lock_Context                           *lock_context
)
{
  11c2ff:       55                      push   %ebp
  11c300:       89 e5                   mov    %esp,%ebp
  11c302:       83 ec 18                sub    $0x18,%esp
  Thread_Control              *the_thread;
  uint32_t                     number_broadcasted;

  if ( size > the_message_queue->maximum_message_size ) {
  11c305:       8b 45 08                mov    0x8(%ebp),%eax
  11c308:       8b 40 14                mov    0x14(%eax),%eax
  11c30b:       3b 45 10                cmp    0x10(%ebp),%eax
  11c30e:       73 0e                   jae    11c31e <_CORE_message_queue_Broadcast+0x1f><== ALWAYS TAKEN
    _ISR_lock_ISR_enable( lock_context );
  11c310:       8b 45 20                mov    0x20(%ebp),%eax            <== NOT EXECUTED
  11c313:       8b 00                   mov    (%eax),%eax                <== NOT EXECUTED
  11c315:       50                      push   %eax                       <== NOT EXECUTED
  11c316:       9d                      popf                              <== NOT EXECUTED
    return CORE_MESSAGE_QUEUE_STATUS_INVALID_SIZE;
  11c317:       b8 01 00 00 00          mov    $0x1,%eax                  <== NOT EXECUTED
  11c31c:       eb 68                   jmp    11c386 <_CORE_message_queue_Broadcast+0x87><== NOT EXECUTED
  }
```

Figure 4: Example for annotated source code.

tation for XCOV is publicly available.
An approach how to perform coverage analysis using on-chip debugging and obtaining execution information via the JTAG interface can be found in [20]. They also used the RTEMS OS to evaluate their approach.
Several commercial, intrusive tools for coverage analysis which instrument source or object code of programs written in C or C++ exist. These are certified tools, e.g., Bullseye [15], SCADE Test Model Coverage [21], or LDRAcover [23] which are used in industry to obtain the required coverage reports.

6 Conclusions

In this paper we introduced a tool chain to measure the code coverage of a test suite. Code coverage represents a measure to assess the quality of a test suite. Since in safety-critical applications, as in the domain of avionics and automotive, complete coverage is required for certification, several different approaches to measure code coverage and tools implementing these approaches already exist. Most of the existing tools are intrusive, which means, that these tools instrument either the source code before compilation or the object code during compilation.
Applying our tool chain does not need any changes within the SUT. In this work we explained the CPU emulator Couverture QEMU which executes an SUT and produces the execution information as execution traces. As an SUT we use a real time operating



RTEMS coverage analysis report

Coverage reports by symbols sets:

Symbols set name	Index file	Summary file	Bytes analyzed	Bytes not executed	Uncovered ranges	Percentage covered	Percentage uncovered	Instruction coverage	Branches uncovered	Branches total	Branches covered percentage	Branches coverage
core	Index	Summary	70468	10304	550	85.38%	14.62%	🟩	362	1468	75.341%	🟩
sapi	Index	Summary	4256	292	23	93.14%	6.861%	🟩	21	81	74.074%	🟩
score	Index	Summary	38008	4356	247	88.54%	11.46%	🟩	191	790	75.823%	🟩
libblock	Index	Summary	40644	11512	250	71.68%	28.32%	🟩	450	1008	55.357%	🟩
rtems	Index	Summary	28204	5656	280	79.95%	20.05%	🟩	150	597	74.874%	🟩
libmisc	Index	Summary	13764	8424	32	38.8%	61.2%	🟩	47	288	83.681%	🟩

Analysis performed on Thu Sep 24 12:53:53 2015

Figure 5: Overview of coverage results for LEON2 CPU.

system RTEMS for which we present coverage reports as experimental results. We introduce the tool *covoar* which supports processing of different execution trace formats and produces the coverage reports as HTML and ASCII files.

6.1 What was discovered?

The RTEMS code coverage effort with *covoar* began between the 4.8 and 4.9 (2008) release series. There was no objective coverage measure before this point. Some of our initial observations were interesting. First, we were a little surprised at the incompleteness of the test suite. We knew that there were some areas of the RTEMS code that were not tested at all, but we also found that areas we thought were tested were only partially tested. We also observed some interesting things about the code we were analyzing. We noticed that the use of inlining sometimes caused significant branch explosion. This generated a lot of uncovered ranges that really mapped back to the same source code. We also found that some defensive coding habits and coding style idioms could generate unreachable object code. Also, the use of a case statement that includes all values of an enumerated type instead of an if statement sometimes lead to unreachable code. Also we generally improved code via refactoring and simplification, changes to coding style and addition of test cases. None alone was sufficient.

6.2 Future Work

Currently we are working on integrating *gcov* [22] into our tool chain such that we can validate the obtained coverage results of *covoar* and gcov mutually. Further we will revise the report generator to modularize the output considering the analyzed executable, library or object file. To analyze source code coverage metrics as decision coverage and MC/DC we will map the branches within full historical execution traces to the source code and check whether decision coverage or MC/DC is satisfied.

References

[1] *RTEMS: Real-Time executive for multiprocessor systems.* `http://www.rtems.com`

[2] Bordin M., Comar C., Gingold T., Guitton J., Hainque O., Quinot T., Delange J., Hugues J., Pautet L.;*Couverture: an Innovative Open Framework for Coverage Analysis of Safety Critical Applications.* In Ada User Journal, Vol. 30, Issue 4, 2009.

[3] *ECSS standard ECSS-E-ST-40C.* European Cooperation for Space Standardization (ECSS). Space engineering - software. ESA-ESTEC, Requirements & Standards Division, Mar 2009.

[4] *RTCA/DO-178C Software Consideration in Airborne Systems and Equipment Certification.* RTCA Inc., 2011.

[5] *ISO 26262.* Road vehicles - Functional safety. Nov 2011.

[6] *http://wiki.qemu.org*

[7] *https://forge.open-do.org/projects/couverture-qemu*

[8] *http://www.gaisler.com/index.php/products/simulators/tsim*

[9] *http://skyeye.sourceforge.net*

[10] *http://ada-auth.org*

[11] *http://www.esa.int/TEC/Software_engineering_and_standardisation/SEMHYAXIPIF_0.html*

[12] Bordin M., Comar C., Gingold T., Guitton J., Hainque O., and Quinot T.; *Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework.* In Embedded Real Time Software and Systems (ERTSS) 2010.

[13] *CAST-10, What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?* Position Paper. Certification Authorities Software Team. 2002.

[14] Wang P., Jonquet A., Chailloux E.; *Non-Intrusive Structural Coverage for Objective Caml.* In Electronic Notes in Theoretical Computer Science, Vol. 264, Issue 4, 2011.

[15] `http://www.bullseye.com`

[16] `http://www.dwarfstd.org`

[17] `https://devel.rtems.org/wiki/TBR/Website/Board_Support_Packages`

[18] `http://www.gnu.org/software/binutils/`

[19] Delange J., Perrotin M.; *On integration of open-source tools for system validation, example with the TASTE tool-chain.* In 13th Real-Time Linux Workshop, 2011.

[20] Cunha J. C., Barbosa R.,Rodrigues G.; *On the Use of Boundary Scan for Code Coverage of Critical Embedded Software.* In IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE), 2012.

[21] `http://www.esterel-technologies.com/products/scade-test/test-creation-host-execution/scade-test-model-coverage/`

[22] `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

[23] `http://www.ldra.com/en/ldracover`