An Attempt to Design and Implement Contiki and Cooja Regression Test Suites using Combinatorial Testing

Abhinandan H Patil
BITS Pilani, Goa
Zuari Nagar, Goa
Goa
919886406214
Abhinandan_patil_1414@yahoo.com

Krishnan Rangarajan
Dayanand Sagar College
Bangalore
Karnataka
Krishnanr1234@gmail.com

Neena@goa.bits-pilani.ac.in

ABSTRACT
Although several studies have been conducted on regression test selection, augmentation, prioritization, and pruning, regression test suite creation is performed in an ad hoc manner for most software systems by selecting a few crucial parameters and their combinations. Regression test suite creation can be rigorously approached by using the combinatorial testing (CT) method. Although this approach is advantageous, it has been used in only a few cases. This paper presents a practical approach for applying CT and augmenting the Contiki and Cooja base regression test suites with the National Institute of Standards and Technology tools. We discuss the results of using CT for the Contiki and Cooja regression test suites. In our study, we examined the inadequacies of existing regression test suites. We then incorporated additional tests to re-engineer the existing test suites by using the Automated Combinatorial Testing for Software tool. The test suite was written from scratch for the Cooja simulator of the Contiki operating system. In each case, we measured the coverage of the Contiki simulator. We observed a marginal increase in the simulator code coverage for the re-engineered test suite. However, a substantial increase was observed in the simulator coverage for the test suite written with the simulator acting as a system under test. In this study, we also examine the automation of test case generation for Contiki and Cooja.

Key words: ACTS, CT, NIST

1. INTRODUCTION
Software systems developed and maintained by teams must have a robust regression testing mechanism in place. This mechanism is required because the changes in the codebase must be tested, and the new code must be incorporated almost overnight. In most cases, a regression test suite is created in an adhoc manner by selecting a few crucial parameters and their combinations. This results in a small test suite that can be executed rapidly; however, the obtained test suite neglects crucial test cases and does not satisfy the testing requirements.

In this study, we used the Automated Combinatorial Testing (CT) for Software (ACTS) tool of the National Institute of Standards and Technology (NIST) [1] to generate combinatorial test cases from the Contiki and Cooja regression test suites [2]. We used the code coverage as a criterion for measuring the testing adequacy. Contiki is an operating system, whereas Cooja is a simulator. Contiki and Cooja are primarily used in embedded devices [3]. They have a large code with several input parameters. We can build a practical and reliable regression test suite by augmenting an existing regression test suite through additional tests generated with a combinatorial approach.

2. BACKGROUND
Contiki is a widely accepted Internet of Things (IoT) operating system and is suitable for memory- and resource-constrained devices. Contiki is an open-source software with a substantial user community. Contiki software includes Instant Contiki, which is a user-friendly environment for testing. By using VMWare, Instant Contiki can be launched in a desktop environment. The Ubuntu-based environment includes toolchain dependencies that help in making incremental changes to the operating system easy. Contiki can be built for various target platforms by tweaking the make file. Moreover, Contiki supports several hardware platforms. Instant Contiki is built in a Java Simulator tool called Cooja, which interacts with Contiki through the Java Native Interface.

Cooja has a standard regression test suite in the regression test folder. The test suite comprises XML files with the csc extension. Cooja can understand these csc files. The XML files have information regarding the configuration and arrangement of the mote type as well as scenario-specific JavaScript embedded in them. Although Contiki supports various hardware platforms, the regression test suite does not reflect it. The test cases are concentrated around limited mote types.

In a preliminary investigation, we noticed a scope for improving the test suite. Two additional test suites were planned. These additional test suites were named the re-engineered and Cooja test suites. Details regarding the test suites are provided in Table I.

Table I. Test suites and their description

Test suite	System under test	Additional comments
Base test suite	Mainly Contiki	This can be found in a regression folder. This suite is referred to as “Test Suite A” in this paper.
Re-engineered test suite	Primarily Contiki	This suite was created by selecting additional test cases from the ACTS tool. This test suite is referred to as “Test Suite B.”
Cooja test suite	Primarily Cooja simulator	This test suite was created from scratch. This test suite is

		referred to as "Test Suite C."

We aimed to quantify the effectiveness of the test suites by collecting the coverage data in the operating system and its simulator for the cases A and B and the coverage data in the simulator for case C. However, no open-source or proprietary code coverage tools of C-language software are available for the target platforms supported by Contiki. Furthermore, Contiki is a hard real-time operating system. C code coverage tools add to the overhead in the form of probes or traces, which makes test cases fail. We explored the J Test Pro [4] and G Cover software [5]. These software did not satisfy the testing requirement because they used proprietary compilers and their supported hardware platforms were different from those of Contiki. Therefore, we indirectly measured the coverage of the simulator for Test Suites A and B. For Test Suite C, the coverage of the simulator written in Java was measured directly because the system to be tested was a simulator.

2.1 Existing Regression Test Suites

Unlike other IoT operating systems, such as RIOT [6] and Tiny OS [7], the Contiki operating system includes a standard regression test suite, which is referred to as a base regression test suite. The base regression test suite contains 64 test cases, which are used for the regression testing of Contiki. An initial analysis of the regression test suite indicated that the test cases were unevenly distributed across the hardware platforms supported.

The lack of testing across hardware platforms is a serious problem because several applications are planned on the Contiki operating system. The absence of an appropriately tested code results in applications that are not functioning because of problems in the operating system.

No specification or test design documents are available for the Contiki operating system. Thus, the existing regression test suite is a starting point for understanding the functionalities supported by the Contiki operating system.

2.2 ACTS Tool for Generating Combinatorial Test Design

A report by the NIST indicates that inadequate testing results in a cost of $59.5 billion per year to the US economy [8]. A similar cost is possible for software developed in other parts of the world. Various testing methodologies are actively studied and proposed. The NIST proposed a method called CT in the field of testing. The NIST provides users with two tools as a part of CT, namely the ACTS and Combinatorial Coverage Measurement (CCM) tools.

A survey report by Nie indicates that CT is widely accepted [9]. The NIST published a manual for practical CT [10]. The books by Ammann and Beizer serve as reference material for testing [11,12]. The ACTS tool is suitable for cases in which testing must be performed for more than two parameters [13]. CT is applied to the software of large organizations [14]. The CCM tool can be used to complement the ACTS tool [15]. To demonstrate that the ACTS tool can be used to test real-time software, CT was applied to the ACTS tool [16].

In this study, we used the ACTS tool of the NIST for generating the test design. The ACTS tool uses various algorithms to generate the test cases [6]. We applied the ACTS tool to generate the test cases for Cooja. Our preliminary analysis indicated that the test cases generated by the ACTS tool were evenly distributed around the hardware configurations. The input that must be supplied to the ACTS tool is provided in Appendix A. The input includes the parameters and parameter values of Cooja as well as the constraints that must be supplied to the ACTS tool. The ACTS tool generates its output in a series of rows, where each row represents a test case. The test cases are mapped to the functional test cases and can then be executed in the test environment.

2.3 Code Coverage Using OpenClover

The code coverage is a quantitative measure of how appropriately the test cases test the software. Currently, various Java coverage tools are available. Coverage tools employ either source code or bytecode instrumentation for collecting coverage data. Instrumentation is a process in which a tool inserts additional hooks into the codebase. The tool later uses these hooks for collecting data. The coverage was meant for both the Contiki operating system and its simulator Cooja. In this study, we examine the coverage data on Cooja. For Cooja, the build.xml file is already available. Therefore, the CodeCover and Clover tools were ideal for our study [17].

The regression test cases were written such that the Java Virtual Machine terminated each test case. The coverage tool CodeCover generated the coverage log files (CLFs) for every session of the run. Thus, for 100 test cases, 100 CLFs were generated. To obtain the consolidated view at a regression test suite level, the sessions of the run must be merged. CodeCover is ineffective for merging hundreds of sessions. However, the Clover tool can effectively merge the sessions. Clover is an open-source software (effective since April 2017). Clover automatically augments the database files of various sessions. Thus, merging the sessions manually or through shell scripts is not required. We used the Clover Java code coverage tool for collecting coverage data. The build.xml file was suitably modified for code coverage data collection.

3. RE-ENGINEERING THE BASE TEST SUITE

Two scenarios exist for the software under study:

- The software includes the standard regression test suite.
- The software does not include the base regression test suite.

In the first case, the missing test cases can be generated using the CCM tool. Another approach involves designing the test suite by using the ACTS tool and finding the missing test cases from the regression test suite. In the second case, the test design uses the ACTS tool and selects an appropriate number of test cases according to the coverage requirements.

In this section, we examine how the inadequacy of the coverage data can be addressed by re-engineering the regression test suite.

3.1 Test Design Using the ACTS Tool

We generated the test design by using the ACTS tool of the NIST. The ACTS tool evenly distributed the test cases around the mote types. The test cases of Contiki and Cooja were in the *.csc format, which is an XML file that Cooja can understand. The *.csc files were scenario-specific and typically a few hundreds of lines in length. In our previous research, we generated the design by using the ACTS tool. The test design exhibited an even distribution of the test cases around the MICAz, ESB, Wismote, Z1, Sky, and Contiki mote types.

Figure 1 depicts the process of collecting the benchmark code coverage data. The code coverage data of the baseline regression test suite was compared with that of the re-engineered test suite.

In a previous study, we described the design of the ACTS tool by considering all the parameters at a time [18]. The implementation of such test cases was impractical, and therefore two parameters were considered at a time.

The test design was implemented using the ACTS tool of the NIST. The input parameters were selected after examining the base regression test suite. The ACTS tool indicated the existence of 289 test cases. This shows that ACTS tool was inefficient in generation of required test cases.

3.2 Functional Test Case Generation

The functional test cases built the firmware from the *.c files in the examples directory and copied the files to the motes for a given scenario.

However, the build of the firmware failed for a few target types.

For example:

1) The file example-runicast.c in the directory /home/user/contiki-2.7/examples/rime was successful in building for all the target types, namely the Sky, ESB, exp5438, Z1, Wismote, and MICAz motes.

2) The aforementioned behavior was expected from the example-trickle.c in the directory /home/user/contiki-2.7/examples/rime. However, the build (i.e., "make command") was successful for the Sky, ESB, Z1, and Wismote motes but failed for the exp5438 and MICAz motes.

The behavior in 1 and 2 was external to the test cases that were implemented. Thus, the test cases depended on the successful build for all the target types. Therefore, all the test cases could not be implemented using the ACTS tool. Furthermore, we wanted to restrict the number of test cases to a reasonable count (say 100) because if all the 289 suggested test cases were implemented and included in the regression, it would result in the following:

• A test all approach.

• Very long execution cycles (considering the time taken to execute the test cases in Contiki and Cooja).

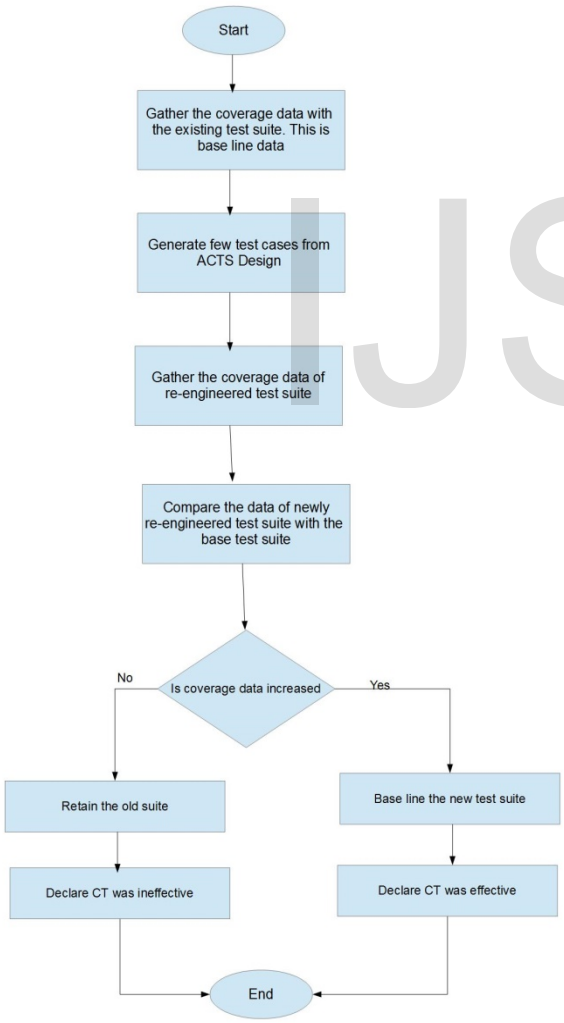

Figure 1. Process of collecting the code coverage in CT

4. TEST DESIGN BY USING THE ACTS TOOL ON THE RE-ENGINEERED TEST SUITE

The test cases suggested by the ACTS tool formed a super set of the base regression test suite. Because the base regression test suite already had 64 test cases, an additional 35 test cases from the ACTS design were implemented using autogeneration. We essentially had two test suites:

• The base regression test suite of Contiki, which had 64 test cases.

• The modified regression test suite with 99 test cases from the ACTS design. Among the 99 test cases, 35 were new and the remaining 64 were the same as the test cases of the base regression test suite.

5. AUTOGENERATION OF TEST CASES

Thirty-five additional test cases were introduced to the base regression test suite. The functional test cases exceeded 100 lines of XML code. Thus, the addition of 35 test cases translated into the addition of more than 3000 lines of XML code. Because the effort involved was substantial, the test cases were autogenerated.

5.1 Autogeneration of Functional Test Cases

The process involved using the human-readable text files to autogenerate functional test cases. The functional test cases included mote-arrangement-specific information as well as scenario-specific JavaScript. The mote and mote arrangement information was autogenerated using the developed tool. The scenario-specific JavaScripts were introduced manually.

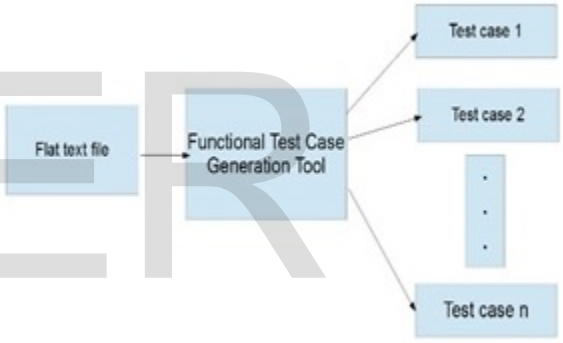

Figure 2. Tool for the autogeneration of functional test cases

Figure 3 illustrates the process used for autogenerating the test cases. The output of each stage acted as the input of the subsequent stage. We developed a tool comprising 897 lines of code. The code was written in Java, and the Cooja code was reused at several locations for the generic engine. The generic engine was coded first and required drivers. The RegEx package was used to parse the input text file. The parsed information was used to populate the internal data structures of the tool. The driver then used this data to drive the generic engine. In summary, the input text file contained all the configuration information of the mote types embedded in it. The output files were csc XML files that included the configuration information required for the test cases. The scenario-specific JavaScript was then manually embedded in the test case for completion. The XML line count for the activity was more than 3500.

Figure 4 displays a sample input text file accepted by the tool. The parser was written such that any number of csc files could be generated in one run. The logical blocks for the individual test cases are called records, and each line within the record is a field. The engine is generic because it works for any number of mote types and motes. The code written for this study can be found in GitHub [19]

Figure 5 displays the sample output XML file generated using the developed tool. The generated XML file was complete in all aspects except the scenario-specific JavaScript. Because the generation of

scenario-specific information could not be automated, the scenario-specific JavaScript was inserted manually.

In this study, test cases were generated through a two-pass mechanism. The skeletons of the XMLs were generated in the first pass, and the JavaScript was inserted in the second pass. Thus, complete test cases were generated, which were ready for execution in Cooja. The advantages of automation include the elimination of manual work and human errors when coding individual lines of XML.

6. AN ATTEMPT TO GENERATE THE COOJA TEST SUITE USING THE ACTS TOOL

Appendix B gives the test design that was used to generate the test cases using the ACTS tool. The tool generated impractical number of test cases. The solution was to run the test cases manually with the aid of ACTS.

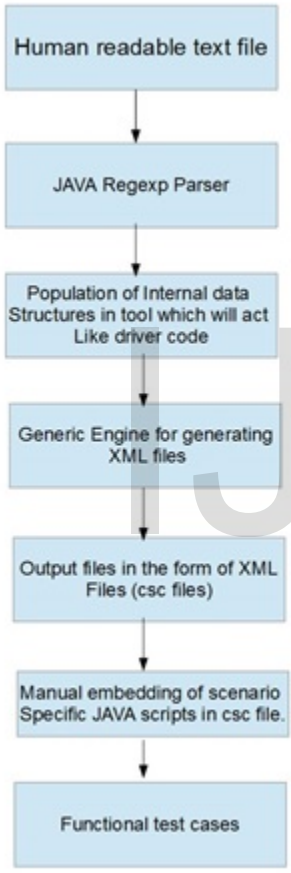

Figure 3. Autogeneration process for the functional test cases

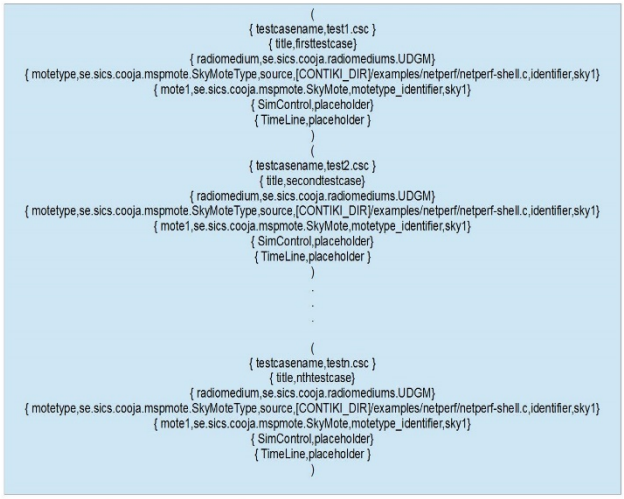

Figure 4. Sample text input file

Figure 5. Sample XML output file

7. PROCESS OF COLLECTING CODE COVERAGE DATA

Figure 1 depicts the process flow for collecting coverage data prior to and post CT. Figure 1 indicates the comparison that must be performed. The coverage data of the base regression suite is the reference point. Figure 6 illustrates how the Clover tool interacts with Cooja to generate the output files for inference. Figure 7 illustrates the changes that must be made to the test environment for obtaining the required code coverage data. The build.xml file must be appropriately modified for incorporating the Clover tool in the Cooja environment.

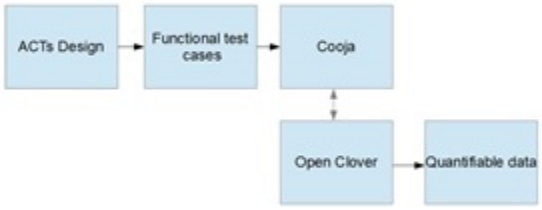

Figure 6. Interaction between Cooja and the OpenClover tool

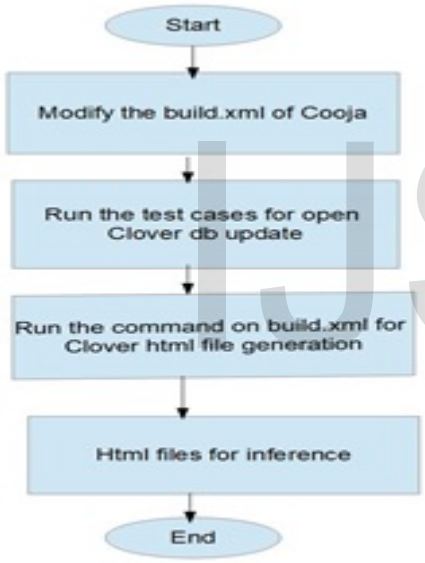

Figure 7. Changes in the test environment for collecting Clover data

8. RESULTS

Table II provides a comparison of the code coverage in the simulator for various Java packages. The total percentage of coverage (TPC) at the simulator level depends on the internal calculation of the coverage tool employed for collecting the coverage data. Section 9 describes how the TPC is calculated in the Clover tool.

Table II. Comparison of code coverage in simulator for three test suites

Java package	Test Suite A	Test Suite B	Test Suite C
Cooja.plugins.analyzers	0%	0%	77.8%
Cooja.plugins.skin	0%	0%	72.5%
Cooja.positioners	0%	0%	87.4%
contikimote.interfaces	3%	3%	52%
cooja.util	6.50%	6.50%	53.7%
cooja.motes	6.60%	6.60%	58.6%
cooja.plugins	6.60%	6.90%	72.5%
cooja.dialogs	7.30%	7.70%	69.8%
cooja.contikimotes	17.90%	19.70%	64.2%
cooja.interfaces	20.80%	20.80%	69.3%
se.sics.cooja	33%	34.60%	77%
cooja.radiomediums	43.60%	44%	60.7%
cooja.emulatedmotes	1.70%	53.80%	54.7%
TPC	13.6%	14.7%	70.5%

Figure 8 illustrates the comparison of the code coverage in the simulator in the form of bar charts. The figure compares the coverage at the package level of the simulator for the three test suites.

The coverage at the package level alone does not provide sufficient information. Figure 10 indicates the class coverage distribution of the three test suites (A, B, and C). In an ideal test output, all the classes are represented in the extreme right bar of the chart. The bars on the right side of the chart for Test Suite C are taller than those on the left side, which indicates that the testing in the simulator was superior for Test Suite C than for Test Suites A and B.

Figure 11 displays a tree map of the coverage in the simulator for the three test suites. The following convention is used in tree maps:

- Deep red: No coverage
- Pale green: Full coverage
- Yellow: Lies between red and green
- Square size: Indicates the complexity of the code

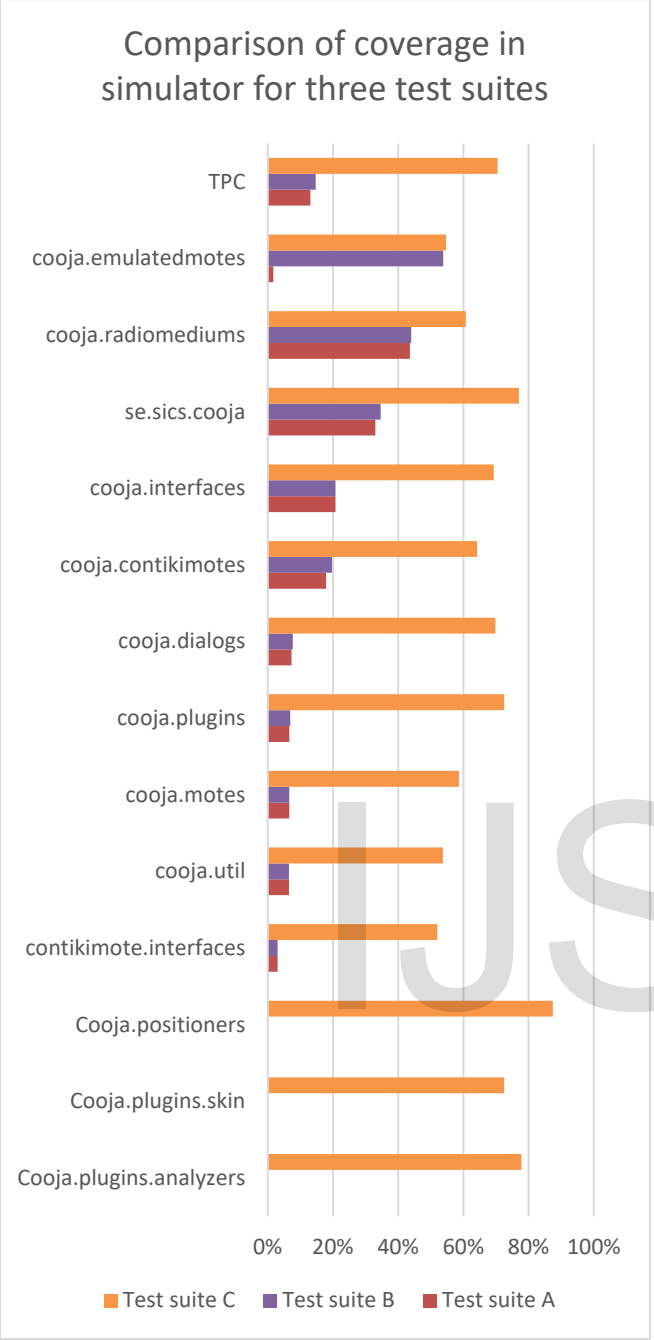

Figure 8. Comparison of the code coverage in the simulator for the three test suites

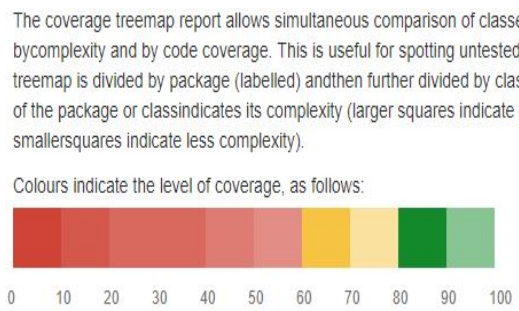

Figure 9. Reading the tree map

Figure 10. Class coverage distribution for Test Suites A (top), B (middle), and C (bottom)

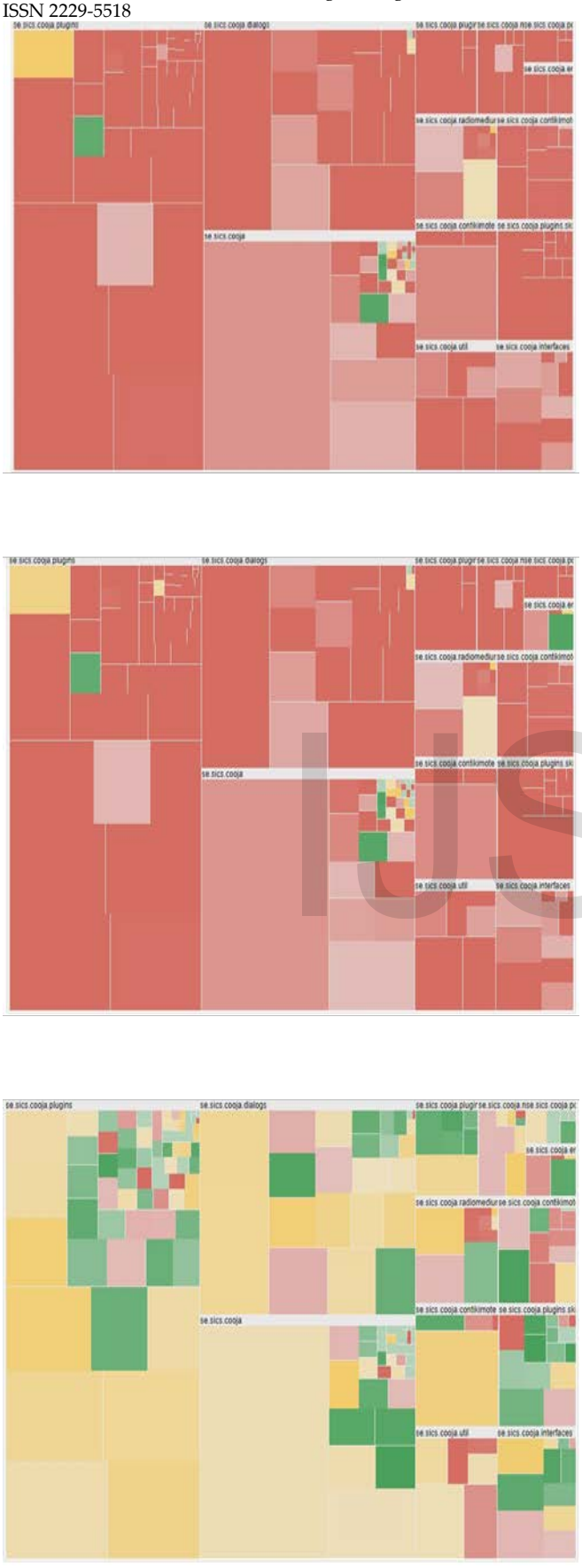

Figure 11. Tree maps of the code coverage in the simulator for Test Suites A (top), B (middle), and C (bottom)

9. ANALYSIS

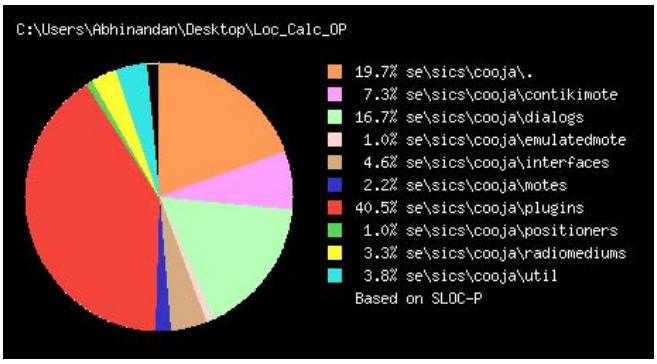

Figure 12. Source code distribution in the simulator with LOC metrics

Code metrics

Branches: 5,292 **Statements:** 15,608 **Methods:** 2,156
Classes: 269 **Files:** 155 **Packages:** 13 **LOC:** 45,272
NCLOC: 29,755 **Total complexity:** 5,399
Complexity density: 0.35 **Statements/Method:** 7.24
Methods/Class: 8.01 **Classes/Package:** 20.69
Average method complexity: 2.5

Figure 13. Code metrics for the Cooja code base

Test Suites A and B exhibited a TPC of approximately 13.6% and 14.7%, respectively. We analyzed the code of Cooja to determine the causes for low coverage percentage. The major packages of the code are plugins, the Cooja package, and dialogs. Cooja supports two modes of execution, namely the graphical user interface (GUI) mode and non-GUI mode. The regression test suite environment was written such that Cooja ran in the non-GUI mode. The plugins, Cooja package, and dialogs included a significant code meant for the GUI mode. In this case, achieving a high TPC with Test Suites A and B was impossible.

CT was applied to Test Suite B for testing various configuration combinations. The Cooja code was written such that the firmware file was built externally (The *.C files in the examples directory were compiled.) and loaded into the mote types. This logic is primarily present in the files se.sics.cooja.Contikimote.ContikiMoteType and se.sics.cooja.Simulation. Therefore, despite using the CT test cases for testing various hardware configurations, no increase was observed in the code coverage of the simulator.

9.1 Coverage Analysis of Test Suites

We used three test suites:

1) A base test suite with 64 test cases (Test Suite A).

2) A re-engineered test suite with 64 base and 35 ACTS test cases (Test Suite B).

3) The Cooja test suite designed from scratch (Test Suite C).

The TPC for Test Suite B was 1.1% higher than that for Test Suite A. The TPC is calculated as follows in the Clover tool:

$$TPC = (BT + BF + SC + MC)/(2 \times B + S + M) \times 100\%$$

where

BT: Branches that evaluated to “true” at least once
BF: Branches that evaluated to “false” at least once
SC: Statements covered
MC: Methods entered

B: Total number of branches

S: Total number of statements
M: Total number of methods

B, S, and M were obtained from the Clover output. Thus, the total increase in (BT + BF + SC + MC) was 312 for the 35 added test cases.

Each package responded differently to the 35 added test cases. For example, the emulated motes package registered a 52.1% increase in (BT + BF + SC + MC) for the additional 35 test cases of Test Suite B.

The low TPC is attributed to the manner in which Cooja runs in the regression mode. During regression, Cooja runs in the non-GUI mode. However, a significant portion of the code in Cooja is meant for the GUI mode.

In the code of Cooja, the firmware is built externally and loaded into the mote types for various emulated configurations of CT. The remainder of the code is common for the various mote types. The TPC increase is low with this type of code. However, in this case, the TPC increase is not a direct measure of the effectiveness of CT for Test Suite B.

For Test Suite C, the simulator (i.e., Cooja) is the system under test. In this mode, the intention is to thoroughly test the simulator. Test Suite C runs the simulator in both the GUI and non-GUI modes. For Test Suite C, we primarily concentrated on the success path test cases. Moreover, some critical failure path test cases were executed. A quick examination of the Cooja code revealed 237 catch blocks of Java code. These corresponded to 237 failure scenarios. We did not hit all the failure paths. We concluded the refining of input parameter modeling at 70% coverage.

10. SUPPLEMENTARY INFORMATION

The Clover output of the test executions are maintained in a repository that can be accessed online [20].

11. CONCLUSION

We present a test suite design approach in this research by using CT. The base regression test suite was redesigned using the CT approach. The test environment of Contiki has difficult-to-use constraints for the ACTS generated test cases. We had to explicitly select the test cases that were runnable in Contiki and Cooja environments from the ACTS generated set. Functional test cases were autogenerated and added to the base regression test suite. The increase in the coverage of the simulator was marginal for the re-engineered test suite because of the execution mode of the simulator and simulator code structure. We designed the test cases from scratch for the Cooja system under test. This designed suite exhibited a substantial increase in the simulator coverage. In this study, we investigated the use of CT on Contiki and Cooja.

ACKNOWLEDGMENTS

We thank the developers of the open-source and free software used in this research.

REFERENCES

[1] 'NIST', http://csrc.nist.gov/groups/SNS/acts/index.html.

[2] 'Contiki Operating System', http://www.contiki-os.org.

[3] 'Contiki supported hardware platforms', http://www.contikios.org/hardware.html.

[4] ‘J Test Pro’, https://www.segger.com/products/debug-probes/j-trace/technology/real-time-code-coverage

[5] ‘G Cover’, https://www.ghs.com/products/safety_critical/gcover.html

[6] 'RIOT Operating System', https://riot-os.org

[7] 'Tiny Operating System', https://github.com/tinyos/tinyos-main, accessed 21 September 2017

[8] D. Richard Kuhn, Raghu N. Kacker and Yu Lei.: 'Introduction to combinatorial testing', 2013, Text book.

[9] C Nie.:, 'A survey of combinatorial testing', ACM Computing Surveys, Vol. 43, No. 2, Article 11, Publication date: January 2011.

[10] D. Richard Kuhn, Raghu N. Kacker and Yu Lei.:,'Practical combinatorial testing manual',2013,NIST special publications 800-142.

[11] P. Ammann, J. Offutt.:, 'Introduction to Software Testing', Cambridge University Press, New York, 2008.

[12] B. Beizer.:,'Software Testing Techniques', Van Nostrand Reinhold, New York, 2nd edition, 1990.

[13] J. Bach, P. Shroeder.:, 'Pairwise Testing - A Best Practice That Isn't',Proceedings of 22nd Pacific Northwest Software Quality Conference, 2004, pp. 180-196

[14] 'Introducing Combinatorial Testing in Large Organizations', ASTQB, Mar 2014

[15] 'Combinatorial coverage measurement', NASA IV&V Workshop, Sept 11-13, 2012.

[16] M. Nouroz Borazjany, L. Yu, Y. Lei, R.N. Kacker and D.R. Kuhn.:, 'Combinatorial Testing of ACTS: A Case Study'

[17] 'Open Clover', http://openclover.org/, accessed 11 September 2017

[18] Abhinandan H. Patil, Neena Goveas and Krishnan Rangarajan.:, 'Re-architecture of Contiki and Cooja Regression Test Suites using Combinatorial Testing Approach', ACM SIGSOFT SEN, 2015, Volume 40 Issue 2, pp 1-3,doi:10.1145/2735399.2735413

[19] ' Test case autogeneration code Git hub repository', https://github.com/Abhinandan1414/CoojaTestCaseGeneration

[20] ‘Test logs repository’, https://drive.google.com/drive/folders/0B2vHzPHgs0nVZWxtSE5sVVdGUmc

APPENDIX A: ACTS TEST DESIGN FOR THE RE-ENGINEERED TEST SUITE

Parameters:

Platform	[Exp5438, z1, wismote, micaz, sky, jcreate, sentilla-usb, esb, native, cooja]
base	[Multithreading, coffee, checkpointng, none]
Rime	[collect, rucb, deluge, runicast, trickle, mesh, none]
NetPerformance	[NetPerf, NetPerf-lpp, NetPerf-cxmac, none]
collect	[shell-collect, shell-collect-lossy, none]
ipv4	[telnet-ping, webserver, none]
ipv6	[ipv6-udp, udp-fragmentation, unicast-fragmentation, ipv6-rpl-collect, none]
RPL	[up-root, root-reboot, large-network, upanddownroots, temporaryrootloss, randomrearrangement, rpl-dao, none]
ipv6apps	[servreg-hack, coap, none]

Relations:

Constraints :

(base != "none") => (Rime == "none")
(base != "none") => (NetPerformance == "none")
(base != "none") => (collect == "none")
(base != "none") => (ipv4=="none")
(base != "none") => (ipv6=="none")
(base != "none") => (RPL =="none")
(base != "none") => (ipv6apps == "none")
(Rime != "none") => (base=="none")
(Rime != "none") => (NetPerformance == "none")
(Rime != "none") => (collect == "none")
(Rime != "none") => (ipv4 == "none")
(Rime != "none") => (ipv6 == "none")
(Rime != "none") => (RPL == "none")
(Rime != "none") => (ipv6apps == "none")
(NetPerformance != "none") => (base == "none")
(NetPerformance != "none") => (Rime == "none")
(NetPerformance != "none") => (collect == "none")
(NetPerformance != "none") => (ipv4 == "none")
(NetPerformance != "none") => (ipv6 == "none")
(NetPerformance != "none") => (RPL == "none")
(NetPerformance != "none") => (ipv6apps == "none")
(collect != "none") => (base == "none")
(collect != "none") => (Rime == "none")

(collect != "none") => (NetPerformance == "none")
(collect != "none") => (ipv4 == "none")
(collect != "none") => (ipv6 == "none")
(collect != "none") => (RPL == "none")
(collect != "none") => (ipv6apps == "none")
(ipv4 != "none") => (base == "none")
(ipv4 != "none") => (Rime == "none")
(ipv4 != "none") => (NetPerformance == "none")
(ipv4 != "none") => (collect == "none")
(ipv4 != "none") => (ipv6 == "none")
(ipv4 != "none") => (RPL == "none")
(ipv4 != "none") => (ipv6apps == "none")
(ipv6 != "none") => (base == "none")
(ipv6 != "none") => (Rime == "none")
(ipv6 != "none") => (NetPerformance == "none")
(ipv6 != "none") => (collect == "none")
(ipv6 != "none") => (ipv4 == "none")
(ipv6 != "none") => (RPL == "none")
(ipv6 != "none") => (ipv6apps == "none")
(RPL != "none") => (base == "none")
(RPL != "none") => (Rime == "none")
(RPL != "none") => (NetPerformance == "none")
(RPL != "none") => (collect == "none")
(RPL != "none") => (ipv4 == "none")
(RPL != "none") => (ipv6 == "none")
(RPL != "none") => (ipv6apps == "none")
(ipv6apps != "none") => (base == "none")
(ipv6apps != "none") => (Rime == "none")
(ipv6apps != "none") => (NetPerformance == "none")
(ipv6apps != "none") => (collect == "none")
(ipv6apps != "none") => (ipv4 == "none")
(ipv6apps != "none") => (ipv6 == "none")
(ipv6apps != "none") => (RPL == "none")
(base != "none") || (Rime != "none") || (NetPerformance != "none") || (collect != "none") || (ipv4 != "none") ||
(ipv6 != "none") || (RPL != "none") || (ipv6apps != "none")

APPENDIX B: ACTS TEST DESIGN ATTEMPT FOR THE COOJA TEST SUITE

Input Parameter Model 1:

Parameters:

FileOperation	[NewSimulation, OpenSimulation, CloseSimulation, SaveSimulation, ExportSimulation, Exit]
Simulation	[StartSimulation, ReloadSimulation, ControlPanel, Simulation, Null]
Motes	[AddMotes, MoteTypes, RemoveAllMotes, Null]

Relations:

[2,(Simulation, Motes)]

Constraints :

(FileOperation = "CloseSimulation") => (Simulation == "Null")

(FileOperation = "CloseSimulation") => (Motes == "Null")

(FileOperation = "Exit") => (Simulation == "Null")

(FileOperation = "Exit") => (Motes == "Null")

Input Parameter Model 2:

Parameters:

FileOperation	[NewSimulation, OpenSimulation, CloseSimulation, SaveSimulation, ExportSimulation, Exit]
Simulation	[StartSimulation, ReloadSimulation, Null]
Tools	[Network, MoteOutPut, TimeLine, BreakPoints, RadioMessages, SimulationScriptEditor, Notes, BufferView, MoteRadioDutyCycle, MoteInformstion, MoteInterfaceViewer, VariableWatcher, MSPCli, MSPCodeWatcher, MSPStackWatcher, MSPCycleWatcher, SerialSocket, CollectView, Null]

Relations:

[2,(Simulation, Tools)]

Constraints :

(FileOperation = "CloseSimulation") => (Simulation=="Null")

(FileOperation = "CloseSimulation") => (Tools == "Null")

(FileOperation = "Exit") => (Simulation == "Null")

(FileOperation = "Exit") => (Tools == "Null")

(FileOperation = "SaveSimulation") => (Simulation=="Null")

(FileOperation = "SaveSimulation") => (Tools == "Null")

(FileOperation = "ExportSimulation")=>(Simulation=="Null")

(FileOperation = "ExportSimulation") => (Tools == "Null")

Input Parameter Model 3:

Parameters:

FileOperation	[NewSimulation]

RadioMedium	[UDGM_DistanceLoss, UDGMConstantloss, DirectedGraphRadioMedium, NoRadioTraffic, MutiPathRayTraceMedium]
CreateNewMoteType	[DisturberMote, ImportJavaMote, CoojaMote, MicazMote, SkyMote, Exp430F5438Mote, Wismote, Z1Mote]
Tools	[Network, MoteOutput, TimeLine, BreakPoints, RadioMessages, SimulationScriptEditor, BufferView, MoteRadioDutyCycle, MoteInformation, MoteInterfaceViewer, VariableWatcher, MSPCli, MSPCodeWatcher, MSPStackWatcher, SerialClientSocket, SerialServerSocket, CollectView]

Relations:
[3,(RadioMedium, CreateNewMoteType, Tools)]

Input Parameter Model 4:

Parameters:

FileOperation	[OpenSimulation]
IOTScenario	[RplUdp, RplUdpPowerTrace, SkyWebSense, UnicastExample, BroadCastExample, RplCollectTreeDenseNoloss, RplCollectTreeSparseLossy, UdpStream, TrickleLibrary, RimeCollect, RimeBroadCast, HelloWorld, Netdb, NetPerfSky, ServerClient, ServerOnly, CoapServerClientExample, RestServerExample]
Tools	[Network, MoteOutPut, TimeLine, BreakPoints, RadioMessages, SimulationScriptEditor, BufferView, MoteRadioDutyCycle, MoteInformation, MoteInterfaceViewer, VariableWatcher, MSPCli, MSPCodeWatcher, MSPStackWatcher, SerialClientSocket, SerialServerSocket, Collectview]

Relations: