Supporting Tool for Embedded Software Testing

Chorng-Shuh Koong¹, Hung-Jui Lai¹, Chih-Hung Chang², William C. Chu³, Nien-Lin Hsueh⁴, Pao-Ann Hsiung⁵, Chihhsiong Shih², Chao-Tung Yang³

¹Dept. of Computer and Information Science, National Taichung University, Taichung, Taiwan
²Dept. of Information Management, Hsiuping Institute of Technology, Taichung, Taiwan
³Dept. of Computer Science and Information Engineering, Tunghai University, Taichung, Taiwan
⁴Dept. of Information Science and Information Engineering, Feng Chia University, Taichung, Taiwan
⁵Dept. of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan

csko@mail.ntcu.edu.tw, bcs097108@ms3.ntcu.edu.tw

Abstract—Embedded software testing is a complex and difficult task. During the testing process, testing tasks often require additional hardware devices to assist. This paper, in the perspective of software solution, presents a supporting tool for testing embedded software. This tool can automatically generate test cases and test drivers, and supports unit test and coverage test which are based on cross testing technology and multiple rounds mechanism. The test results can be visually represented to facilitate the observation. By using our constructed testing environment in the embedded platform to do testing experiments, the experiment results show that our constructed testing environment is workable.

Key words - automatic testing; embedded software testing; coverage testing; unit testing; cross-testing; testing tool; test case generating; object testing

I. INTRODUCTION

As computer software technology has evolved to more sophistication, vast scale of embedded software application and growing customized demands have highlighted the importance of embedded software quality control [1]. Among the most discussed issues of today’s software engineer, how software quality could be improved is the major one raised [2] extensive attention. Software testing is one of the leading approaches [3] to ensure software quality since it is the key to enhance software quality and reliability. However, mass manpower and efforts are consumed during software development. In general, testing time span and efforts can be economized through automatic or semi-automatic approaches and tools.

Random testing [4] is currently the most popular testing approach to evaluate software quality and reliability. Software testing based on random testing method is firstly to randomly generate mass test cases [5] from all possible input collection, and then through the technology of code coverage test to quantify software testing quality. Studies indicates that when testing coverage is increased, software reliability is strengthened accordingly [6]. Relatively speaking, embedded system confronted with software and hardware limitations are more than those general-purpose computers faced with. More efforts are needed for testing engineer to conduct testing and improving embedded software quality during the process of testing embedded software [7]. Traditional unit testing tool is mainly focusing on workstation platform. Test case and test input data are generated from manual or automatic input.

Automation as it is, approaches still need to be improved in generating test case [3]. Most unit testing tools, specifically, are only capable of either automatically generating program framework of test case, or merely supporting primitive type. Test engineers are obliged to manually write testing program/code segment and input test input data under the generated program framework, or generating test case manually [9]. The nature of testing demands numerous repetitions, which leads to enormous workloads. In addition, testing conducted through manually input test data is neither efficient, precise, nor capable of increasing test coverage [10]. In terms of testing performance, how embedded software execute on the Target platform can hardly be monitored by test engineer, making it even difficult to improve testing performance during the testing process [11].

To reduce executing overhead from implementing testing functions on embedded platform and decrease efforts for test engineer, we have developed an automatic testing tool to support cross-testing. Automation embedded software testing can be realized in the host-side and target-side testing environment of our system. Host-side is in control of the preprocessing action, including parsing source code, automatically generating instrument code segments and instrument to the target program, automatically generating test case and test driver based on cppunit library. Target-side is the actual target environment to implement embedded software testing. When testing is in process, the module will collect related testing data and results and transfer to host-side for further analysis and presentation of test results as reference for next round of automatically generated testing data. With the host-target mechanism, testing does not need to be implemented solely on embedded system platform. Hence, testing efficiency can be increased, embedded hardware resources can be utilized more fully, and efforts of test engineer can also be curtailed.

Further, a random testing approach is developed to test simple and complicated data types, including primitive type, structure type and object type, so that automatic unit test can be free from the restrictions of testing input data type. With the approach, test case and test input data is generated automatically aiming at these types. Further, line coverage and branch coverage can be multi-round tested automatically. During runtime, performance of how program is executed on target-side can be monitored. Each round of coverage test
results, executed time spent can be presented in visualized fashion.

II. RELATED WORK

M. E. Delamaro [12] developed a coverage testing tool for mobile device software. The tool, named JaBUTi/ME, can support Java source code and can solve restrictions of performance and storage of mobile device. Testing conducted on mobile device is difficult since elements such as memory limitations, persistent storage, and network connection availability have to be taken into consideration.

The tool allows the testing of mobile applications not only on emulators, but also on the desired target mobile device. For test engineer, testing mobile devices can be performed simply on a desktop computer where test session is generated and instrument class. Communication between desktop and mobile devices can be exchanged by test server. During instrumented code is being executed on mobile devices, trace data is transferred through network link to monitor the testing. This approach can help reducing workloads from testing mobile devices. Presenting test result on visualized interface keeps test engineer informed of how many lines are yet to be executed. Automation such as generating test input data and test case have not been emphasized, making it more time-consuming for test engineer to edit test source code and manually generate test input data.

Jooyoung Seo [13] proposed a tool Justitia to test embedded software. Debugging can be executed for test engineer to set the breakpoints on the interface. However, more times are needed to conduct testing if test engineers are not quite experienced in testing or they do not know architecture of the entire embedded system.

The focal function of Justitia is automatically detecting errors on interface of program. The existing monitoring and debugging technology of emulator is combined to test embedded software. By defining embedded software interface pattern, an automated scheme is created to find location of source code interface. The testing tool can automatically generate test case (interface test feature, location of interface, symbol to be monitored at the interface, input data, and expected output) and execute test case using emulation testing technology. Further, the system can support memory test and interrupt test. After completing testing, the result of test coverage and interface error can be presented on visualized interface. The main focus of the system is on single testing. Extra time and efforts are needed to increase test coverage testing because its lack of multi-round automatic testing.

Yongyun Cho [14] proposed a Multi-paradigm views embedded software testing tool. The tool is based on client/server model of host-target architecture. It is a performance testing tool independent from extra hardware support. On host-side, this tool provide graphic user interface for test engineer and software testing is executed on target-side of embedded system platform which include memory test, code coverage test, and function performance test. In addition, Yongyun Cho has designed a xml-based DTDs (Document type definitions) to increase test script usability and reusability. On the host-side, with the help of test script wizard and test driver wizard, test engineer can generate test script, test suite, and test driver by inputting data manually. Visualized options are provided to present test result. Target-side is the embedded software test platform where target program is executed on HRP-SC3410 (Ami) of embedded Linux. The tool developed by Yongyun Cho makes it easy for test engineer to generate test script, test suite, and test driver. However, test script and test driver can not be generated automatically and there's a shortage of multi-round automatic testing. Test engineer would need to take extra time to write test script and test driver.

III. EMBEDDED SOFTWARE AUTOMATIC TESTING TOOL

The Automatic Testing Environment for Multi-core Embedded Software (ATEMES) is composed of four parts: Pre-Processing Module (PRPM), Host-Side Auto-Testing Module (HSATM), Target-Side Auto-Testing Module (TSATM), and Post-Processing Module (POPM). Testing functions provided by ATEMES system include coverage testing, unit testing, performance testing and race condition testing. In this paper, we focus on coverage testing and unit testing. System module is shown as Figure 1. System architecture layers are shown as Figure 2. Functions of respective modules are detailed as follows sections.

A. ATEMES System Module Description

1) System architecture layers

ATEMES system is divided into 5 layers shown as Figure 2. The bottom layer is hardware platform layer. On host-side, we have used X36 Platforms. Platforms of target-side can be X36 or ARM11 MPCore. The 3rd layer is OS layer. On both sides, we have used Linux operation system. The 3rd layer is the library layer we adopted, including GCOV [15], Cppunit [16], and JFreechart [17]. The 4th layer is application layer constructed by ATEMES system modules. The 5th layer is the graphic user interface layer of host-side.

Figure 1. System modules

Source code Preprocessing Module Host-side Auto Testing Module Post-processing Module Target-side Auto Testing Module

Testing Engine Test result report

Analysis test information

Preprocessing information

Instrumented code result

Trace data

Database

Test result report

Analysis test information

Instrumented code result

Trace data

Database

Input data

Expected output

Execute Program

Analysis test information

Figure 2. System architecture layers

A1. ATEMES System Module Description

1) System architecture layers

ATEMES system is divided into 5 layers shown as Figure 2. The bottom layer is hardware platform layer. On host-side, we have used X36 Platforms. Platforms of target-side can be X36 or ARM11 MPCore. The 3rd layer is OS layer. On both sides, we have used Linux operation system. The 3rd layer is the library layer we adopted, including GCOV [15], Cppunit [16], and JFreechart [17]. The 4th layer is application layer constructed by ATEMES system modules. The 5th layer is the graphic user interface layer of host-side.
2) Pre-processing testing module

PRPM module is in control of the pre-processing testing tasks, including parsing source code, automatically generating test case and test input data, and automatically instrument source code so that data can be collected during the execution of target program and report to host-side during runtime. System module is shown as Figure 3. Figure 4 shows the example of test case framework automatically generated.

3) Host-side and target-side automatic testing module

Tasks of automatic testing are completed by two modules, which are the HSATM module on the host-side, and the TSATM module on the target-side “Figure 5”. The main function of HSATM module is to automatically generating test driver based on cppunit library, and automatically compiling test driver and instrumented source code to executable files. The executable files and test input data are then delivered to target-side for execution. The major task of TSATM module is to trigger test driver to execute or terminate testing tasks, and can monitor profiler to collect testing data. It can return collected testing information to host-side during runtime. Figure 6 shows the example of test driver automatically generated.
4) Testing post-processing module

POPM module takes charge of the post-processing testing tasks, mainly by parsing the collected data as reference for the next round generated testing data. To make it easy to get across the test result, outcome of the test result is presented in visualized way. Graphical interface is adopted to present test result during runtime or the end of testing. The modules are composed of 2 parts, test log analyzer and test result presentation module. The system module is shown as Figure 7.

**Figure 6. Test driver example**

**4) Testing post-processing module**

POPM module takes charge of the post-processing testing tasks, mainly by parsing the collected data as reference for the next round of automatically generated testing data. To make it easy to get across the test result, outcome of the test result is presented in visualized way. Graphical interface is adopted to present test result during runtime or the end of testing. The modules are composed of 2 parts, test log analyzer and test result presentation module. The system module is shown as Figure 7.

**Figure 7. POPM module**

**B. Multi-Round Testing Scenario**

UML sequence diagram “Figure 8” will be used in this section to illustrate system operation mechanism. Multi-round automatic testing scenario is selected to describe how the system operates. The scenario of multi-round automatic testing is parsing source code, generating the intended data including test case, test input data, instrument code segment and test driver, automatically executing testing, collecting and parsing test log file, automatically executing the next round testing task based on the parsed outcome until testing termination conditions are met. Actions are as follows:

**Step 1:** PRPM read source code.

**Step 2:** With the aid of information provided by requirement specification files, PRPM conducts parsing source code which has been read and extracts program function name and parameter, internal structure of program, and other information.

**Step 3:** PRPM analyzes the extracted information based on the parsed data by step 2. Test case is generated automatically after analysis, or new test case (test input data) is generated based on the test result feedback by POPM during testing.

**Step 4:** PRPM processes the extracted information based on the parsed data by step 3. Analysis is conducted according to the intended testing items. Instrumented source code is automatically generated after analysis.

**Step 5:** HSATM reads test case data generated from PRPM.

**Step 6:** HSATM automatically generates test drivers based on the data collected by PRPM.

**Step 7:** HSATM reads the instrumented source code generated from PRPM.

**Step 8:** HSATM compiles the instrumented source code and test driver to target-side executable files and uploads to TSATM.

**Step 9:** TSATM is triggered to execute automatic testing. It collects the test result logs and returns to HSATM.

**Step 10:** HSATM receives the test result logs executed by TSATM and passes to POPM module to analysis and statistics.

**Step 11:** Repeat step 3 to step 10 until testing termination conditions are met.

**Step 12:** POPM presents test result to testing engineer.

**Figure 8. The sequence diagram of multi-round testing**
C. Coverage Testing Mechanism

1) Coverage testing

The operation of coverage testing is based on the functions of testing coverage tool of GNU. By using automatic testing mechanism, incorporating functions of multi-round testing and cross-testing, implement coverage testing. Each round of testing coverage result is summed up to increased total coverage of target testing program. During each round testing, visualization result of testing coverage can be presented dynamically on host-side and can be continuously added up to illustrate coverage test report. Process of multi-round coverage testing is shown as “Figure 9”. Algorithm of multi-round coverage testing will be introduced in next section.

2) Multi-round testing algorithm

In order to implement coverage testing functionality, GCov flag will be automatically added when HSATM is compiling source code of target testing program so that executable files of target testing program and *.gcno files can be generated and are delivered to the embedded platform for testing. After each round of testing, *.gcda will be automatically generated and are transferred back to host-side. In the meantime, the POPM module will analyze coverage testing log data (*.gcda and *.gcov files) and calculates line coverage and branch coverage. Algorithm is shown as Table 1.

IV. EXPERIMENT

In order to demonstrate the functionalities of ATEMES, we have selected some programs of data structures as our target testing programs. The testing tasks of ATEMES are fulfilled on the ARM11 multi-core platform. Automatic multi-round testing mechanism is incorporated to conduct coverage testing experiment and unit testing experiment. Test experiment result is presented in visualized way. Software and hardware platforms for the experiments are detailed as follows:

- **Host-side hardware platform**: Intel® Core™ 2 Duo CPU P8400.
- **Host-side OS platform**: Linux UBUNTU 9.10.
- **Target-side hardware platform**: The platform baseboard for ARM 11 MPCore with 4 ARM11 MPCore CPUs.
- **Target-side OS platform**: Linux Kernel 2.6.24.

A. Coverage Testing Experiment

1) Experiment of automatic coverage testing

13 programs of data structures are selected as the target testing program, among which most of input data types are array of structure type. Random testing method is applied as the testing environment to generate test input data. For each function, each round generates 3 sets of input data, and each function is planned to execute 20 rounds of testing. During runtime, testing engineer will be able to observe coverage testing result of each round “Figure 10” and the final testing result. The system can record history data for each round, including line coverage, branch coverage, executing time and information of failed test case.

2) Result of experiment

As shown in “Table 2”, “Table 3” and “Figure 10”, for insertion_sort(), 100% line coverage is covered up from the 1st round to the 5th round; 100% branch coverage is covered up from the 1st round to the 5th round. For other functions, similar situation is likely to happen to reach up to 100% coverage.

![Figure 9. Coverage testing process](image-url)
Nevertheless, the coverage of some functions is hardly reached to 100%. For fractionCalculator(), line coverage and branch coverage in the 4th round reached to 92% and 65%, and then coverage is hardly increased beyond that. The bottleneck lies in that the algorithm we have adopted is random testing method, higher coverage is can be foreseen if other algorithms were adopted.

![Figure 10. The line and branch coverage testing result](image)

As results of experiment 1 and experiment 2 have indicated, our system ATEMES showed to be workable. Efforts from testing engineer can be greatly reduced by the functionalities of automatically and semi-automatically generating test input data. Moreover, the automatically generating test cases programs and test drivers, the mechanism of automatically multi-round cross-testing and visualized presentation of multi-round test result have provided another solution for testing engineer to save his effort as well.

### 1) Result of experiment

As shown in “Figure 11”, on the left side of the figure presents test result of all function in text, including the information of success, failure and error test case. On the right side of the figure, unit test result is shown by pie chart. Summing up statistics from the experiment, total success test cases are 67% (20 test cases), failure test case is 16.5% (5 test cases), and error test case is 16.5% (5 test cases).

![Figure 11. The pie chart of unit test result](image)

### Table 2. Line coverage

<table>
<thead>
<tr>
<th>Function</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
<th>8th</th>
<th>9th</th>
<th>10th</th>
<th>20th</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>92%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>binarySearch</td>
<td>99%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>insertionSort</td>
<td>93%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
</tr>
<tr>
<td>merge</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>mergePass</td>
<td>92%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>quickSort</td>
<td>92%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>quickSort</td>
<td>92%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>qsort</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>randNumSum</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>FractionCalculator</td>
<td>83%</td>
<td>83%</td>
<td>83%</td>
<td>83%</td>
<td>83%</td>
<td>83%</td>
<td>83%</td>
<td>83%</td>
<td>83%</td>
<td>83%</td>
<td>83%</td>
</tr>
<tr>
<td>average</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
</tr>
</tbody>
</table>

### Table 3. Branch coverage

<table>
<thead>
<tr>
<th>Function</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
<th>8th</th>
<th>9th</th>
<th>10th</th>
<th>20th</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>82%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>binarySearch</td>
<td>68%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
</tr>
<tr>
<td>insertionSort</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
</tr>
<tr>
<td>merge</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>mergePass</td>
<td>98%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>quickSort</td>
<td>94%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>quickSort</td>
<td>94%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>runRandGame</td>
<td>64%</td>
<td>82%</td>
<td>82%</td>
<td>82%</td>
<td>82%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
</tr>
<tr>
<td>quickSort</td>
<td>29%</td>
<td>54%</td>
<td>54%</td>
<td>54%</td>
<td>54%</td>
<td>54%</td>
<td>54%</td>
<td>54%</td>
<td>54%</td>
<td>54%</td>
<td>54%</td>
</tr>
<tr>
<td>average</td>
<td>95%</td>
<td>95%</td>
<td>95%</td>
<td>95%</td>
<td>95%</td>
<td>95%</td>
<td>95%</td>
<td>95%</td>
<td>95%</td>
<td>95%</td>
<td>95%</td>
</tr>
</tbody>
</table>

### Table 4. Unit test result information

<table>
<thead>
<tr>
<th>Function name</th>
<th>Test case</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSubString</td>
<td>success</td>
<td>success</td>
<td>exception</td>
<td>out of range</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>electricTraffic</td>
<td>success</td>
<td>assert</td>
<td>failed</td>
<td>success</td>
<td>assert</td>
<td>failed</td>
</tr>
<tr>
<td>factorial</td>
<td>success</td>
<td>success</td>
<td>assert</td>
<td>failed</td>
<td>assert</td>
<td>failed</td>
</tr>
<tr>
<td>insertSort</td>
<td>success</td>
<td>success</td>
<td>segment</td>
<td>fail</td>
<td>segment</td>
<td>fail</td>
</tr>
<tr>
<td>sequence</td>
<td>success</td>
<td>success</td>
<td>success</td>
<td>success</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>FractionCalculator</td>
<td>success</td>
<td>assert</td>
<td>failed</td>
<td>success</td>
<td>success</td>
<td>success</td>
</tr>
</tbody>
</table>

As shown in “Figure 11”, on the left side of the figure presents test result of all function in text, including the information of success, failure and error test case. On the right side of the figure, unit test result is shown by pie chart. Summing up statistics from the experiment, total success test cases are 67% (20 test cases), failure test case is 16.5% (5 test cases), and error test case is 16.5% (5 test cases).
V. Conclusion

During the process of developing embedded software, testing demands mass efforts. Limitations caused by embedded system hardware resource makes it even more difficult to conduct embedded software testing. To alleviate these problems, this study has developed an automatic cross-testing environment to support embedded software testing. Basing on the parsed source code, the system can automatically instrument source code, generate test case and test driver, and support generating primitive, structure and object types of test input data. Other functions such as supporting unit test and coverage test, visualizing test results, and automatically performing multi-round testing are developed to both reduce the burden of test engineer and enhance efficiency when embedded software testing is in process. Furthermore, we have selected some programs of data structures examples as target testing programs. By using our constructed testing environment in the ARM11 multi-core platform to do testing experiments, the test results show that our constructed testing environment is workable.

This research adopted random testing method in our experiment. It is our goal to develop a more precise algorithm to automatically generate input test data, test case and test driver. Further, testing functions, aiming at multi-core embedded software, such as runtime efficiency and multithread race condition testing, will be our challenge for expansion.

References


