AEP - Automatic Exchange of Embedded System Software Parameters

Rita de Cássia Cazu Soldi and Antônio Augusto Medeiros Fröhlich
Federal University of Santa Catarina, Florianópolis, SC, Brazil
rita,guto@lisha.ufsc.br

***Abstract*—The process of debugging embedded system software is a non-trivial task that consumes a lot of time, once it needs a thorough inspection of the entire source code to make sure that there is no behavior beyond expectations.**

Coding and testing embedded systems software, timing and hardware is even more defiant, once developers need to find out how to optimize the use of the scarce resources since the test itself will compete with the application under test by the scarce system resources. Also, both run in proper platforms, that depends on operating systems, architecture, vendors, debugging tool, etc. This makes embedded systems more susceptible to errors as well as specification failures.

This paper presents AEP, a tool to help developers in the process of debugging embedded systems. The main idea of this tool is emulating various possible system configuration to try to find errors in the application. An XML file contains all required information to perform automated compilation, emulation and debugging , and there is no need of human interference.

The evaluation of AEP was in terms of memory consumption and time to perform debugging. The obtained results indicate that even with no previous information this tool can produce helpful data for developers to find and fix bugs.

I. Introduction

An embedded system can be presented as a combination of software and hardware designed to perform a specific task. Applications involving environmental monitoring and analysis, smart cities, and precision farming are only a sample of a set of possible applications.

These systems were designed to monitor and process data related to the physical environment into which they were embedded. Then, the purpose of these systems was extrapolated to interact with the actors of the modification of this environment, humans. Now, embedded systems are widely attached to numerous electronic devices, and their activities are becoming more popular and intrinsic to humans daily life [3].

When the task of the embedded systems does not have a direct interaction with humans, the price for a failure in these systems was more focused on the financial loss such as loss of market share, client information, people time, etc. [15]. Material losses are inconvenient, but endanger human life is an unacceptable risk. With a focus on direct interaction with humans, we must make sure that the behavior of the systems is in accordance with the specification.

The test area has evolved considerably, but testing is still one of the most time-consuming development process. Mainly because it requires a thorough inspection of the source code to find out if the software specification is fully satisfied, and this is a non-trivial process [9].

Software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item [5]. When it comes to embedded systems software, testing should also consider the order of the hardware and the correct integration between hardware and software.

Since the test is not part of the software behavior, it should never interfere in the flow of activities of the software under test. In general-purpose systems, it is usually possible to achieve this premise without much effort, but some special computer systems have some restrictions, such as low memory, low processing power, limited battery time or a deadline to perform a certain activity. In this case, the developer still needs to find strategies ensure that the application under test and eh test itself will not compete for resources.

The debug is a part of the testing process and in this paper, we propose the automatic exchange of configuration parameters (AEP) as automation of the debugging process for embedded systems' application. The exchanging of configuration parameters is performed by a shell script, that works on application-oriented systems modeled in a parametric and feature-based models.

The setting of AEP script can be configured in an XML specification file according to the desired granularity. All configuration directly interferes in time and effectiveness of the tests. After all tests, the script returns to the user/developer a report with all exchange performed and whether there were any errors during compilation or simulation testing.

Since part of AEP script involves emulation and debugging applications without any human interaction during these processes, it is also presented an introduction on how to develop a fully configurable environment for testing embedded systems according to hardware/software requirements.

II. Embedded Systems Debugging Environment

This section presents details of the debugging process, simulation and how to integrate both in order to create a better environment for developing and testing embedded applications.

Regardless of the technique to be used, debugging can be accomplished in two ways: locally and remotely. Local debugging is when the application runs on the same machine as the debugger. As a result, the latency of application and debugger communication is lower. However, the application interferes in the debug process, e.g. if the application under

test crashes, the debugger will need to halt or restart to seek the cause.

In remote debug, this influence does not happen since application and debugger run over separate machines. The tests are performed into an isolated box over a network connection. Despite of having some latency issues, from the debugging point of view, the rest of the process can be viewed as a local debug with two screens connected in only one system.

In order to provide the most number of possibilities for the developer, the emulator used to debug applications must provide both ways to perform this activity. Also, for a useful debug, developers must consider other concepts involved in debugging, such as, how to configure the code execution mode, to observe the application outputs, watch some environment's variables, log the tasks performed and other configurations.

The integration of emulator and debugger is particular for each host/target machine; thus, some steps presented here must be tailored depending on the target architecture. Figure 1 presents the activities required to perform remote debugging. These steps and additional explanation of which techniques and tools are used in this process are listed bellow:

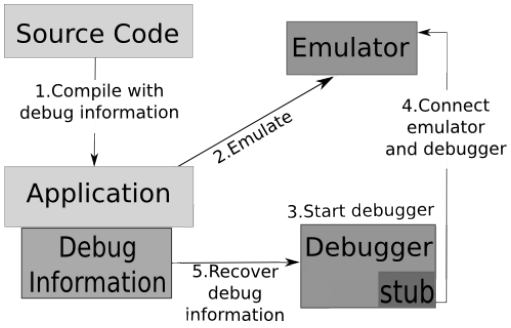

Figure 1: Steps to integrate emulator and debugger

1) **Compile the source code** is the most important step. The source code is the input and the output is the compiled application with debug information. Debug information contains a table that maps instructions in the compiled binary program to their corresponding variable, function, or line in the source code.
2) **Emulate** is a necessary step to execute the application in the correct target architecture. The developer must initiate emulator accordingly and enable a stub to open communication between emulator and debugger.
3) **Connect emulator with debugger** after start the debugger session. The connection must specify the host address and port of the target (emulator).
4) **Recover debug information** is an important step to help finding errors, once its possible to use auto-complete to recovery the all name contained in the symbols table.

III. Automatic Exchange of Embedded System Software Parameters

The algorithm of automatic exchange of parameters is independent of the operating system and platform. However it works on the premise that the system followed an application-oriented system design (AOSD) [16] to guarantee that the operating system will match exactly the requirements of a specific application.

This design methodology uses software engineering techniques to separate abstractions from scenarios hardware and environment dependencies. This way it is possible to configure Abstractions independently from a scenario execution. For AEP it is desirable that each Abstraction of the operating systems can be configured as desired using traits template parameters [14].

The algorithm 1 shows the steps from the choice of a test application until developer receives the report with all attempts. It flows in order to find the desired trait, exchange it for a predetermined amount (or random), run the new application and collect the feedback that the application returns.

Algorithm 1 Algorithm Exchange Configuration Parameters

```
Require: file (Test configuration)
Ensure: Report of tries
  traits⇐ GetTraitFile(file);
  application⇐ GetApplicationFile(file);
  if the file has any configuration value then
    for all configuration in file do
      line⇐ GetTheConfiguration(configuration, traits);
      for all value in configuration values do
        newTrait⇐ ExchangeValue(line, traits);
        newApp⇐ Compile(application, newTrait);
        report⇐ report + Emulate(newApp);
      end for
    end for
  else
    if the file has one maximum number of tries then
      maxNumberTries⇐ GetMaxSize(file);
    else
      maxNumberTries⇐ GetRandomNumber();
    end if
    while tries < maxNumberTries do
      line⇐ GetRandomNumber();
      newTrait⇐ ExchangeValue(line, traits);
      newApp⇐ Compile(application, newTrait);
      report⇐ report + Emulate(newApp);
    end while
  end if
  return report;
```

The configuration file (input) has the only two mandatory information for the beginning of the algorithm: the path to application and its trait. Although it can also contain information such as which configuration must be exchanged, what value should be tested, etc.

A. Configuration file

To run the script with a specific configuration it was necessary to manually fill the information before each new round of tests.

To improve the usability of the script, it is possible to define a configuration file with the information needed to run the test. We chose XML to set test settings because it can set all necessary rules to run the script in a human-readable way and furthermore is also easily interpreted by the computer.

```
<test>
   <application name="dmec_app"></application>
      <configuration>
         < trait >
            <id>ARCH</id>
            <value>IA32</value>
            <value>AVR8</value>
         </ trait >

         <debug>
            <path>"/home/breakpoints.txt"<path>
         </debug>
   </configuration>
<test>
```

Figure 2: Example of AEP configuration file.

Figure 2 brings one example of the AEP configuration file for philosophers dinner application.

The configuration will be set up only once, but even if it requires some manual changes it will not add difficulties, since this file can be read almost as a text: there is a <test> of the *philosopher_dinner_app* <application> with two <configuration>: one <trait> and one <breakpoint>. The <trait> identified as *ARCH* can be *IA32* or *AVR8*. Debugger will use the <breakpoint> file in "*/home/breakpoint_philosopher.txt*" <path>.

At this time, the initial configuration of the testing is still manual, but this automation is being evaluated. For the choice of initial entry, the options are inserting an initial set of entries from a database of cases or cases generation tools. The evolution of the results obtained will be held with optimization tools, artificial intelligence techniques, feature-based models or any other technique that can be automated.

AEP current implementation works on the Embedded Parallel Operating System (EPOS) [4] since it adds a great deal of configurability of the system, which is very suitable for evaluating an exchange configuration script. For a better understanding of the implementation of AEP, we briefly explain how to configure abstractions on EPOS.

B. EPOS

EPOS is a component-based framework that provides all traditional abstractions of operating systems and services like memory management, communication and time management. Furthermore, several academic research and industrial projects uses EPOS as base[1].

This operating system is instantiated only with the basic support for its dedicated application. It is important to highlight that an individual member of a trait is a characteristic of the system and all features of a component must be set appropriately for a better performance of the system. In this context, the automated exchange of these parameters can be used both to discovery a failure in the program by wrong characterization of components, or to improve the performance for the application by selecting a better configuration.

In EPOS, each application has its own trait to define their behavior. The figure 3 shows one excerpt of this configuration

[1]http://www.lisha.ufsc.br/pub/index.php?key=EPOS

```
template <> struct Traits<Build>
{
    enum {LIBRARY};
    static const unsigned int MODE = LIBRARY;

    enum {IA32};
    static const unsigned int ARCH = IA32;

    enum {PC};
    static const unsigned int MACH = PC;
};
```

Figure 3: Excerpt of trait from philosophers dinner application.

for an application that simulates the dining philosophers. This excerpt point how to build the application, which in this example runs in library mode, for an IA-32 (Intel Architecture, 32-bit), using a PC (Personal Computer) machine for generation of the system.

C. Granularity of configuration

AEP provides three different types of test configuration: determined, partially random and random. They should be chosen according to the purpose of the test and the application characteristics, since all configuration directly interferes in time and effectiveness of the tests.

In determined testing, both configuration and value are informed previously in configuration file. This granularity is important when configuration to be tested and its possible values are already known. It is interesting in the sense of optimizing a configuration.

Partially random tests are used to verify configurations and it is an ally to find the optimal configuration of the application. As the value information will not be reported, it will have to be changed by the algorithm. During the election of a value for replacement, maybe the value has already been used previously, making repeated testing and decreasing the efficiency of AEP.

Random test was developed as the worst case. It should be used only when there are no tips or ideas about where could be the error. It may found some wrong values of configuration and help developers to debug smaller applications, but there is no guarantee of coverage tests.

IV. AEP IN A REAL-WORLD APPLICATION

The Distributed Motion Estimation Component (DMEC) performs motion estimation by exploiting similarity between adjacent images in a video sequence. This estimation allows images to be coded differentially, increasing the compression ratio of the generated bitstream. Motion Estimation is a significant stage for H.264 encoding since it consumes around 90% of the total time of the encoding process [7].

Figure 4 presents the interaction between the threads. The `Coordinator` is responsible for defining the partitioning of picture, provide the image to be processed, and return results generated to encoder while `Workers` must calculate motion cost and motion vectors.

The DMEC performance was tested using the AEP integrated environment. The script generated integer random values for the configuration `NUM_WORKERS` and all application could

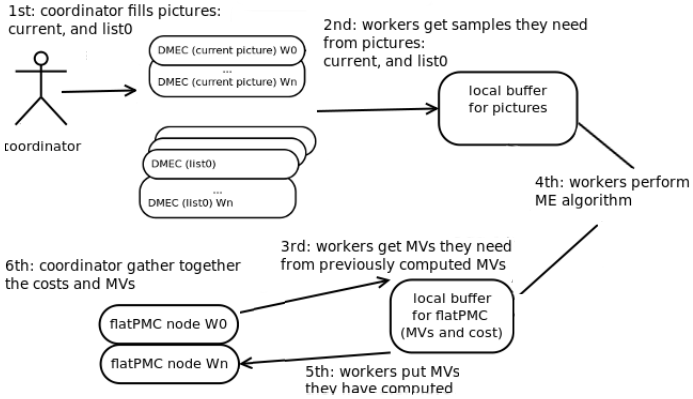

Figure 4: Interaction between the `Coordinator` and `Workers` threads [7]

be compiled and debugged, but only some of them could be emulated by QEMU. AEP performed a reachability test to discovery the cause of these errors. Then DMEC was configured with breakpoints in all functions, specially the main function, see Figure 5. The expected was to reach 5 breakpoints, but after the 3rd breakpoint, "continuing" is the last line that appears in the execution before it crashes.

```
(gdb) target remote :1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb) file app/dmec_app
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from /home/tinha/SVN/trunk/app/dmec_app...done.
(gdb) b main
Breakpoint 1 at 0x8274: file dmec_app.cc, line 47.
(gdb) b testPack
testPack10()  testPack20()
(gdb) b testPack10()
Breakpoint 2 at 0x82f1: file dmec_app.cc, line 66.
(gdb) b testPack20()
Breakpoint 3 at 0x8315: file dmec_app.cc, line 73.
(gdb) continue
Continuing.
```

Figure 5: DMEC debug with GDB execution with `NUM_WORKERS` = 60.

All this steps are automated, and developer will receive a log with information about each configuration that could not even reach the main function. This log could tip the developer about this application fail with a large number of threads and, in this case, emulation and breakpoints were crucial to determine an error in DMEC.

V. Evaluation

The evaluation was performed with two applications: philosopher dinner and DMEC. Both were generated from a x86_64 Linux system, with EPOS 1.1 and compiled with GNU 4.4.4 (IA32 architecture) or GNU 4.0.2 (AVR8 architecture). The integrated environment is composed by GDB 7.5 and QEMU 1.4.0.

For a better analysis of the effectiveness of the tool, test attempts were classified into: compiled, relevant and repeated.. Compilable are those that could be compiled without errors and generated a valid image for the system. The relevant beyond those that are compilable are useful for testing the system, i.e. a configuration modified to alter a system's behavior. Repeated attempts are retries of relevant test. With this classification, the effectiveness will be a compromise between the attempts and the number of relevant tests (minus repeated attempts).

A. Philosopher dinner

The dining philosophers application is the example used to explain the AEP implementation and setup. In this section, it is also base of the analysis of the effectiveness of changes made. Figure 6 presents the classification of data from random test and configured test as described in Section III-A. It is possible to note that the effectiveness of the random test (7%) is lower than that of the partially random test (100%).

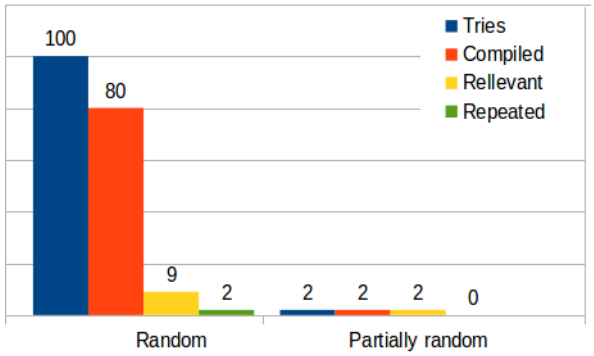

Figure 6: Classification of Philosopher dinner generated configurations

It was expected that a test with some prior information was more effective than one with no information, but the difference was too big. This happened because the application of the dining philosophers is simple and requires little configuration. If we compare the total trait versus the amount of settings that affect the application, this reason is a little more than 10%.

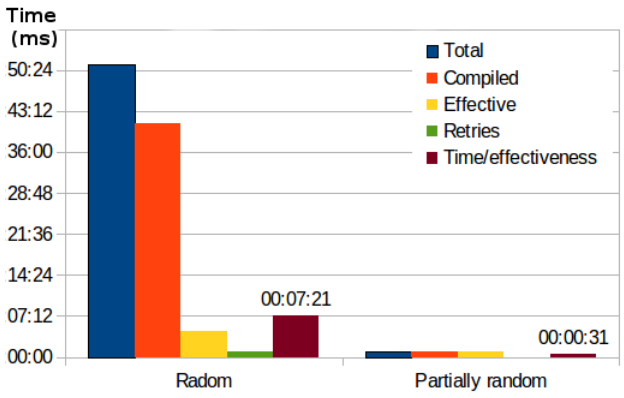

Figure 7: Classification of Philosopher dinner execution time

Time results are in Figure 7. Time/effectiveness is the compromise between the total time taken to execute the complete script divided by the amount of results that tested some configuration. This ratio should be as close as possible to 31 milliseconds because it is the time to build/simulate this application. In the case of totally random testing, this value is

far from desirable, and it would take almost 7 seconds to find a valid configuration to change the parameters.

B. DMEC

DMEC evaluation considered data from random and partially random tests: random test is the one that has no prior information on the application and partially random test has some tips about relevant settings and valid configurations.

The configuration for DMEC partially random test is the same presented in Section IV. It is considered a mix between test with configuration and random test because although there is a setting on the range of values to be tested, the value itself is only defined by AEP. Figure 8 presents data classification:

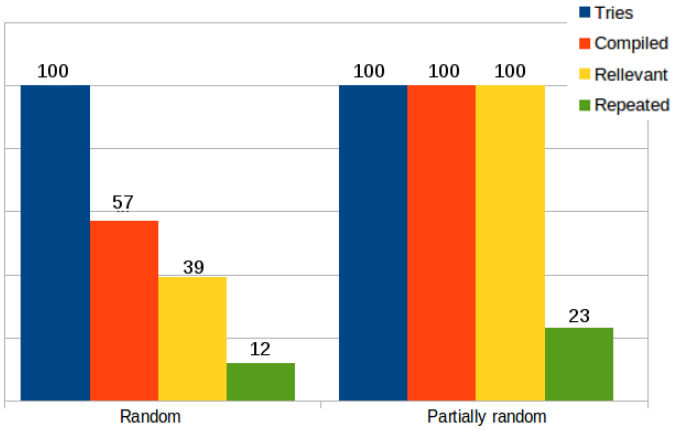

Figure 8: Classification of DMEC generated configurations

In the case of DMEC there were several traits to be modified, but there was a lower rate of compilable applications than the dining philosophers. This was due to the complexity of the system and the dependency between application settings, which are not provided for in AEP.

The complexity of the application also increased the average time to build the system for 46 milliseconds, as shown in Figure 9.

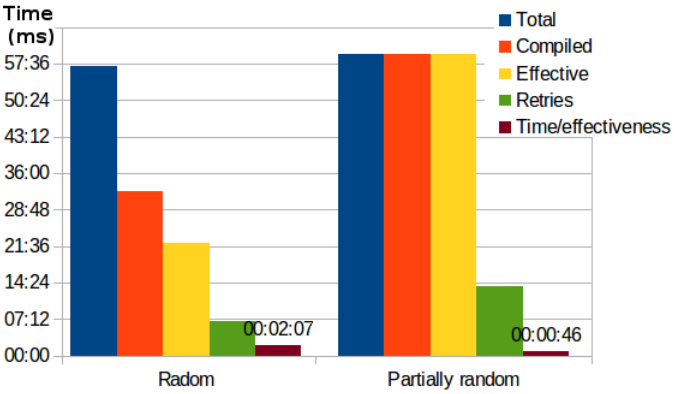

Figure 9: Classification of DMEC execution time

The time difference between the random test and partially random test was much smaller than the difference shown in the Dining Philosophers. This was because there was at least one limitation of the amount offered for the exchange of parameters in DMEC.

In addition to the results and benefits of the AEP, it is also important to inform the burden of testing with the automatic exchange of parameters presents. After all, the emulation of the algorithm based on special debug information, which are provided during the compilation of the application.

In terms of memory consumption, none of example applications have a big image, but when we add the extra information at compile time, the size overhead is considerable. In the case of a simple application such as philosophers dinner the increase was more than 826%. The time it takes to compile information to be also caused an overhead that in both applications was around 63%.

C. Discussion of configuration test types

The above results show the random test as having smaller gain in terms of time spent to perform the AEP and the quantity/quality of information returned. However if variability of data is more important than the effectiveness of this would be the best option.

Partially random tests are interesting to verify configurations and can it is an ally to find the optimal configuration of the application. On the other hand, this configuration test is a mix between random and deterministic, so it has a medium quantity/quality rate and the need of some value limitations for AEP configuration.

Determined tests are what have the highest rate of effectiveness in return is what most needs attention and commitment to the configuration file of the test setup. This type is useful to make modifications to any component of the application, but also want to ensure that the functionality will be equivalent.

VI. Related Work

The automated debugging area has a vast literature that inspired this proposal, most of the approaches are focused on general purposing systems, and both parameters exchange script and test environment for debugging embedded systems applications were designed based on them.

Seo et al. [13] proposes an interface technique that identify and classify interfaces between embedded system layers. They created a model based tool that generates and executes test cases to analyze these interface layers. They also proposed the emulation test technique that integrates monitoring and debugging embedded systems. Despite the similarity, since we also emulate the target board environment and monitor the behavior of environment variables, Seo et al. focus on testing interfaces and layers, while AEP addresses the testing of components and their integration.

ATEMES [6] is a tool for automatic random tests, including coverage testing, unit testing, performance testing and race condition testing. ATEMES supports instrumentation of source code, generation of tests cases and generation of primitive input data for multi-core embedded software. This system is similar to ours since we also automatically run random tests under cross-testing environment to support embedded software

testing. However, the idea of this present work is to integrate the exchange of parameters directly in the operating system in order to test the application and also optimize the choice of the configuration parameters.

Statistical Debugging techniques [19], [18], [10] are capable of isolating a bug by automatic running an application several times and using generated statistical data to analyze these executions information. This statistical analysis can reduce the bug search area by pinpointing a suspiciousness ranking. This technique can not be easily incorporated into an embedded system, due to the need of large data set to accomplish this statistic and the necessity to have a big data storage to keep information of all executions. However, a ranking pointing out possible errors is a breakthrough in the developer's work. Therefore, the AEP uses a debugging environment was build on a machine with more storage and processing than the embedded system.

In program slicing [12], [17], [1], the main idea is to divide the code into parts, then test and remove paths that do not lead to errors. This technique has two approaches for reducing the path that lead to error: static slicing and dynamic slicing. Static slicing has faster reduction of application path, since the final set of paths leading to the error are an approximation of the real set. In dynamic slicing, the initial entry has a great influence on how to slice the code, allowing a greater precision for final error paths. This technique is interesting because it needs only one error path to simplify the group of inputs to be examined. AEP intends to support both types of slicing through configuration files that can address the whole system or just a part of the application.

In capture and replay [2], [11], [8] the program is executed until it reaches the end and all operations performed are stored in a log. Burger and Zeller developed a JINSI tool that can capture and replay interactions between inter/intra-components. So all relevant operations are observed and run step by step, considering all communications between two components until find the bug. Besides being the most widely used, this technique needs to perform all possible paths from one object to another, making this technique time consuming. An AEP script runs in a highly configurable operating system, on which it is possible to plug a single component with different implementations. The design of AEP absorbs the idea of debug focusing on the components that compose the application.

VII. Conclusion

In this paper, we introduce the automatic exchange of configuration parameters and show how to set up a development environment for embedded applications based on specific hardware/software requirements.

The example application results show a considerable difference between the two testing approaches and confirm that the effectiveness of the algorithm is closely tied to the effectiveness of the configuration received. In DMEC real-world application, both random and partial random configuration pointed that the most complex applications may have high dependency between application settings.

The integrated development environment provides independence of the physical target platform for development and test. Also, developers no longer need to spend time understanding a new development platform whenever some characteristic of the embedded system changes. It is an important step since some embedded systems may not be able to store the extra data needed to support debug. The evaluation of the impact of enable debug information in code size of the real-world application was more than 500% and the overhead for the execution time is about 60%.

References

[1] C. Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer*, 13(3):223–246, 2011.

[2] M. Burger and A. Zeller. Replaying and isolating failing multi-object interactions. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ISSTA 2008*.

[3] L. Carro and F. R. Wagner. Sistemas computacionais embarcados. *Jornadas de atualização em informática. Campinas: UNICAMP*, 2003.

[4] A. A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[5] IEEE. NSI/IEEE Standard 1008-1987, IEEE Standard for Software Unit Testing. No. 1986, 1987.

[6] C.-S. Koong, C. Shih, P.-A. Hsiung, H.-J. Lai, C.-H. Chang, W. C. Chu, N.-L. Hsueh, and C.-T. Yang. Automatic testing environment for multi-core embedded software-atemes. *J. Syst. Softw.*, 85(1):43–60, Jan. 2012.

[7] M. Ludwich and A. Frohlich. Interfacing hardware devices to embedded java. In *Computing System Engineering (SBESC), 2011 Brazilian Symposium on*, pages 176 –181, nov. 2011.

[8] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[9] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 199–209. ACM.

[10] S. Parsa, M. Asadi-Aghbolaghi, and M. Vahidi-Asl. Statistical debugging using a hierarchical model of correlated predicates. *Artificial Intelligence and Computational Intelligence*, pages 251–256, 2011.

[11] D. Qi, M. Ngo, T. Sun, and A. Roychoudhury. Locating failure-inducing environment changes. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. ACM, 2011.

[12] N. Sasirekha, A. Robert, and D. Hemalatha. Program slicing techniques and its applications. *Arxiv preprint arXiv:1108.1352*, 2011.

[13] J. Seo, A. Sung, B. Choi, and S. Kang. Automating embedded software testing on an emulated target board. In H. Zhu, W. E. Wong, and A. M. Paradkar, editors, *AST*, pages 44–50. IEEE, 2007.

[14] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[15] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, pages 02–3, 2002.

[16] G. Tondello and A. Frohlich. On the automatic configuration of application-oriented operating systems. In *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on*, pages 120–, 2005.

[17] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

[18] Z. Zhang, W. Chan, T. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009.

[19] A. Zheng, M. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112. ACM, 2006.