Survey on Testing Embedded Systems

Árpád Beszédes† Tamás Gergely† István Papp‡
Vladimir Marinkovic‡ Vladimir Zlokolica‡ *

†Department of Software Engineering, University of Szeged
‡Faculty of Technical Sciences, University of Novi Sad

Abstract

Embedded systems are widely used in everyday life, thus the quality assurance of such systems are important. One of the quality assurance methods is software testing. Different software testing methods have different applicability in this special environment of embedded systems, which sometimes require specific solutions for testing. The Department of Software Engineering, University of Szeged and Faculty of Technical Sciences, University of Novi Sad have started a joint project whose main topic is embedded systems software testing. The goal of the project is the combination of white-box and black-box testing methods to improve the quality of the tests (and, transitively, the quality of the software) in digital multimedia environment. The goal of this survey is to overview existing, documented solutions for embedded system testing, concentrating on (but not limited to) the combination of structural and functional tests.

Preface

The University of Szeged (USZ), University of Novi Sad (UNS) and Vojvodina ICT Cluster (VOICT) have started a joint project called CIRENE. The project is financed by the European Union, and its main goal is to establish a working cross-border cooperation between the parties. As a proof of concept, the project included a joint research and development activity on embedded systems testing. The Faculty of Technical Sciences on UNS (FTN) has a long-time experience in testing of multimedia embedded systems. Their main profile is black-box testing of digital multimedia devices (digital TVs, set-top-boxes, etc.). The Department of Software Engineering on SZTE (DSE) has been working on improving testing quality using white-box testing methods. The goal of this R&D activity is to exchange knowledge and jointly develop a method or methods specific to embedded systems in which white-box testing methods support black-box methods, resulting in an improved quality of the tests implying higher quality of the products.

This survey serves as a base of this R&D activity. The goals of this survey are

*Additional authors of the paper: Gergő Balogh†, Szabolcs Bognár†, Ivan Kastelan‡, Jelena Kovacevic‡, Kornél Muhi†, Csaba Nagy†, Miroslav Popovic‡, Róbert Rácz†, István Siket†, Péter Varga†

- to search for previous works that utilizes black-box or white-box testing techniques or their combination in embedded system environment;
- to evaluate and classify these works by some defined evaluation and classification criteria, which helps selecting those ones that can be a base of the to be defined methodologies of the R&D activity;
- compare different works by their applicability and potential in using them in embedded systems environment.

The paper assesses the state of the art and enumerates a number of possibly applicable methods and solutions. Later on the project the general and specialized methodologies will be created using this document as the source of knowledge.

1 Introduction

In this survey we try to assess the state of the art of embedded systems software testing. Testing is an important task in software development, and different circumstances entitles for different problems and different solutions. Embedded systems are special types of systems with special attributes (e.g. the software and hardware has more influence on eachother and cannot be entirely separated), thus general testing methodologies can only be applied by limitations. This survey collects and evaluates a number of existing testing methods and tools that could be applied to test embedded systems.

In the rest of this section some background on software testing and embedded systems testing is given. In Section 2 we describe the search methodology we applied when assessing the state of the art. In Section 3 the criteria used to evaluate and compare different solutions are given. In Section 4 the methods, solutions, and tools that contribute to embedded systems testing are listed and evaluated according to the criteria. In sections 4.1 and 4.2 black-box and white-box methods are assessed. In Section 4.3 methods that combine black-box and white-box elements are described and evaluated. In Section 4.4 some tools that provides support for the above methods are listed. In Section 5 a comparison of the different methods and/or tools is given. Finally, in Section 6 we draw conclusions.

1.1 About Testing

Software testing is a very important risk management task of the software development project. With testing, the risk of a residing bug in the software can be reduced, and by reacting on the revealed defects, the quality of the software can be improved. During testing different functionalities, behavior, or quality attributes of the software can be checked and assessed.

Tests can be categorized by many point of view. Using static testing any written workproduct (including source code) of the development process can be examined without executing the software. Dynamic techniques examine the software itself by executing it. Amongst many, there are two basic types of dynamic test design techniques: black-box and white-box techniques.

1.1.1 Black-box testing

The black-box test design technique concentrates on testing the functionalities and requirements of the software without having any knowledge on the structure of the program. The techniques take the software as a black box, examine "what" the program does and do not intrerested in the "how?" question. The black-box techniques test the software against some specification. The input and preconditions of the test cases are determined from some specifications of the program, and whether the test case is executed successfully or not depends on the similarity between the expected output and postconditions of the test case and the actual output and postconditions of the test case execution.

Black-box Testing is one approach for automated functional testing in TV and multimedia technology. It contains both software and hardware components offering a wide range of possibilities for testing of integrated DTV systems, digital satellite and terrestrial receivers (set-top-box - STB), DVD and blu-ray players. It can be used for testing of video and audio quality, measurement of electrical values characteristic for AV signals, automated navigation through menus, for providing signal feeds, performing capturing and displaying of video and audio content, for storage of test results in various formats in a file system or database, generating test reports, etc.

As it is intended for functional testing, it ignores internal mechanisms of the system or component and focuses specifically on the outputs generated as the system response to specific inputs and conditions of test execution.

Execution of tests can be manual, semi-automatic and automatic tests, and tests can be carried out in reference systems (SUT against golden reference system) and in systems without a reference device (comparison against previously captured referent AV files).

In this approach, different types of input devices (generators), one or more SUTs (System Under Test), and audio/video grabber devices are used. Flexible concept is needed to expand the functionality of devices through expansion and modification of devices parameters and commands.

For this purpose, available equipment which user possesses in-house can be used, such as: AV signal generators (Fluke, Quantum, AudioPrecision, and other supported devices), acquisition devices (grabber cards), RC (Remote Controller) emulators (RedRat), instrumentation for electrical measurements, and power supplies (Agilent, Hameg, Tektronix, etc.).

Software part of Black Box Testing is a PC based application for control, development and execution of automated tests. The application is installed on a PC and can be connected with all the generators through interfaces they support (RS232, LAN, USB, GPIB, etc.). The application allows sending of specific commands to adjust parameters of the generated signal. The application can also send commands to the SUT (over RC emulator, RS-232, LAN, etc.) bringing it into a desired state, required by a test scenario (e.g. quality of image brightness on CVBS input), followed by acquisition of video signal by the dedicated grabber device. Later on, the test continues with analysis of the captured SUT output against previously defined audio or picture references („golden reference"), grabbed from the referent device, using defined algorithms for video or audio quality assessment. Thus, the results of the test are obtained based on a defined limit of deviation of the grabbed sequence compared to the reference.

Types of testing in Black Box Testing:

- Manual testing
- Semi-automatic testing
- Automated testing

Manual testing requires that all steps of the test are carried out manually by tester, in accordance with the description given in the test scenario. Application in a step by step manner displays messages with description of each step that needs to be carried out; upon the step execution the tester resumes the test until all test steps are accomplished. At the end the application prompt window pops up with a question on the test result, including a field where the tester can enter a comment. Evaluation of the results is performed post-run by a professional based on visual observations. The major differences between semi-automatic and automated testing are that at the former the tester decides on the result of the testing (like in manual tests) and the system performs automatic control and management of deployed devices, whereas at the latter algorithms built into test system makes decision on the test results. In the case of automated tests the criteria for decision making (PASS, FAIL and others) are set by the test requirements. The criteria are forwarded to the test management mechanism built into the control application as a parameter used to settle on whether the test passed or failed. Automated testing of integrated DTV systems presumes functional testing of supported interfaces. Devices generating video and audio content intended for testing of each specific interface are connected to SUT, which performs post-processing of the content. After the actions of the predefined test scenario are accomplished the resultant SUT output is grabbed from the TV motherboard and its content is verified against the reference. Using additional analogue and digital generators RF functionality test can also be covered. Control emulators fitted for the specific DTV producer enables automatic navigation and setting of TV menu options (brightness, color, sharpness, volume, etc.).

Three different oracles:

- Golden reference testing - at this type of testing, referent AV content (golden reference) used to compare grabbed images and audio against, is known in advance. Referent AV content is usually obtained by recording of AV output from the referent device which had been approved to operate reliably. Another option for creating of referent AV content is by using image and audio editors. Upon the tests' execution, grabbed files are compared against the references from the device considered to be the referent one, based on which pass/fail test criteria had been set.
- Golden device testing - at this type of testing, during the testing itself, SUT outputs are compared against outputs from the device declared as "golden device". AV outputs from both devices are captured "live" (at test run time) and compared by an algorithm which decides on the test success (pass/fail).

- Testing without reference - when the testing is performed without a referent device or previously recorded referent files, this technique can be used. It is based on algorithms for image and audio processing for real time detection of MPEG like artifacts and artifacts caused by signal broadcast. Most commonly detected artifacts are blocking, blurring, ringing, and field loss for video, and signal absence and discontinuities for audio signals.

1.1.2 White-box

The difference between white-box testing and black-box testing is that while black-box testing concentrates on the question "What does the program do?", and has no information about the structure of the software, white-box testing examines the "How does the program do that?" question, and tries to exhaustively examine the code from some aspects. This exhaustive examination is given by a so-called coverage criterion. The code gets executed during testing of the program to measure coverage.

There are two main types of white-box coverage criteria:

- Instruction coverage defines that program points should be executed during the tests. What a program point means is dependent on many factors like granularity (it can be sole instructions, basic blocks, methods, classes, modules, etc.).

- Branch coverage defines how different program paths should be executed or different decisions should be exercised during the tests. Of course, it is dependent on the definition of program point: on instruction level we can examine decisions, or even parts of the decisions (e.g. condition coverage); while on method level the call graph paths can be examined.

The coverage information somehow should be extracted from the test execution. There are many possibilities to do this:

- Trace generation is an important part of the white box testing. It means the code parts that are reached during the execution of a test case. To calculate traceability and coverage we need to follow the run of the program. Instrumentation and debugging can provide this following by inserted feedback points.

- Code instrumentation is inserting instructions that output some information about the interesting points of the executed code. The information content and the interesting points are vary depending on the coverage level and criterion. For example, a simple method coverage requires only a binary "I was executed" information at the beginning of each methods, while condition coverage requires to output the value of all elementary condition of an executed decision, and the code providing this information needs to be inserted into all decisions (thus all decision points needs to be instrumented). This instrumentation can be made in source code or in binary code.

- Instrumenting the middleware can be a good solution if we use one middleware for many programs, and we want to get information from all the programs. The middleware lies between the hardware and the operating system, and it is built up from libraries and drivers. If we insert methods into this middleware which send back information from the execution, than we can collect some kind of information.

- Modifying execution framework (virtual machine) by extend the code of the framework. This is a software layer between the executable binary code and the operating systems. It is an environment in which special binary can be executed. Special binary is an intermediate language which is typically compiled from simple source code. We can use call trace which consists of information of called method.

- Debugging can be made in hardware level, and we need to have debug port in the hardware or a debugger device, which can communicate with the hardware in common ports. The debugger can read the code in the hardware and can insert breakpoints into it and can store additional code or contact to other devices which stores additional code. When the trap instruction is encountered, a software interrupt is generated. The additional instrumentation code may then be executed. After it, the original instruction content is restored. Debuggers provide very detailed information on the program execution.

These coverage information can be used to manipulate the executed test set: we can select from the test cases to reach a special aim, or we can prioritize them to reach a chosen coverage on the code in a shorter time. Other usage of the coverage information is to calculate other property of the test cases or the code.

Traceability is the ability to link product documentation requirements back to stakeholders' rationales and forward to corresponding design artifacts, code, and test cases. Traceability can be computed based on the connection between the functionalities, the test cases and the coverage information.

Reliability provides an estimation of the level of business risk and the likelihood of potential application failures and defects that the application will experience when placed in operation. We can calculate the reliability from the possible locations of the faults, which can be ascertained from the coverage and traceability information.

1.2 Difficulties of Embedded Systems Testing

In this section the most experienced difficulties in embedded systems testing are depicted.

A primary characteristic of embedded systems is the variety of available platforms for developers, like the different CPU architectures, their vendors, operating systems and their variants. These systems are not general-purpose designs, by definition. Typically, they are designed for a specific task, so the platform is specifically chosen to optimize that kind of application. Having this, the consequences are more difficulties for embedded system developers, harder debugging and testing, since different debugging tools are required for different platforms [1].

The development of embedded systems is more focused on testing and system evaluation than desk-top systems. In embedded systems, errors and failing behaviour can stay unnoticed for quite a while, only until things like service failure or a device which is not responding appear in embedded systems. Of course, these errors and failures can be corrected on time, so that no problems in systems occur. In order to achieve this behaviour, or to at least improve a certain systems behaviour, it is necessary to follow through with system monitoring and to analyze the system post-mortem [48].

Embedded systems have become widely spread and popular, controlling a vast variety of devices. For functional and error correctness validation of these systems, the most commonly used method is software testing. Effective testing techniques could be helpful in improving dependability of embedded systems, and therefore developing such testing techniques can be a challenge [60].

Embedded systems consist of software layers. Application layers utilize services provided by underlying system service and hardware support layers, while a typical embedded application consists of multiple user tasks. System failures in field applications can be caused by two different kinds of interactions, those that occur between application and lower layers, and those that occur between various user tasks initiated by the application layer. In embedded systems, a particularly difficult problem in the testing domain can be the "Oracle problem". Oracle automation is complicated by the uncertain determination of expected outputs, for given inputs. This can occur due to the multiple tasks which could have a non-deterministic output.

There are many different classes of real-time embedded systems. For example, hard real-time embedded systems have strict temporal requirements, and include safety critical systems such as those found in avionics and automotive systems. Soft real-time embedded systems, in contrast, occur in a wide range of popular but less safety-critical systems such as consumer electronic devices, and tend not to have such rigorous temporal constraints.

Since embedded systems are usually real-time systems as well, its correctness of execution is not only characterized by its logical correctness, but by moment when the result is produced as well, especially in the case of hard real-time systems. Thus, not only when the expected result is missing, but also when the expecting result is produced but outside the period defined by timing constraints, the system is considered as failing.

As the failing behaviour is not acceptable for many embedded systems, specifically safety-critical systems, the testing of meeting timing constraints is equally important as testing functional behaviour of these systems.

Some characteristics of embedded world are making the testing process of embedded systems slightly different than testing systems use in other fields:

- Platforms for execution and running application are usually separately developed
- Wide spectre of development architectures
- Cross-development environments impacted by existence of a number of execution platforms
- Limited resources and tight space for timing constraints on the platform
- Implementation paradigms can be diametrically different
- Frequently unclear design models

- New quality and certification standards

Testability and measurability of an embedded system is often affected by these issues, what is the main reason for testing such systems to be so difficult and thus considered as the weakest point of development process. Having this in mind, it is natural that more than 50 percent of total development effort is spent in testing embedded systems, especially the systems which development is months behind the expected schedule, which is also more than 50 percent [21].

Having complex embedded designs with frequently changed requirements, the testing of real-time embedded systems is particularly difficult. They usually require a number of rigorous white-box (structural) and black-box (functional) testing modules, as well as the integration testing before releasing them to market. The functional testing is usually more important than structural, and similarly, the integration testing is more challenging task than module testing, and even more, functional integration testing requires separate test scripts generated based on the system requirements [58].

2 Search methodology

To collect valuable information for this survey paper we searched for previous works (papers and tools) connecting black-box and white-box techniques (so called gray-box testing), or applying such techniques in embedded systems environment.

As a first step, we collected in-house knowledge: created a list of relevant papers and tools that had been reviewed or applied in previous research and development activities. Next, Google and Google Scholar searchers were used to find scientific articles and case studies. We started to search with basic terms as "graybox", "gray box", "gray-box", "graybox testing", "graybox process", "graybox testing process". Unfortunately, the found articles showed that these terms are widely used but note not only those techniques we are interested in. As a result, very few relevant papers were found. The next terms we were searching for were "whitebox helped blackbox testing" and "whitebox aided blackbox testing". These searches also resulted in a huge number of hits. By filtering out irrelevant ones, many papers were left. Unfortunately, after processing these papers we had to realize that most of them concentrated on the results of applying such methods and not on the elaboration or explanation of the testing processes they used.

At this point we narrowed search by making the search terms more specific to the R&D activity we wish to perform. As code coverage is decided to be used in the project, we started to search for "coverage aided blackbox testing" and "coverage aided testing". There were much less hits than with the previous more general search terms, but finally these papers are found to be very relevant ones.

After trying to find complex papers and solutions that fit to our goals, we started to collect relevant information one by one to the following terms: "white-box testing", "code coverage", "instrumentation", "model-based testing" and "embedded system testing". With these term we found a huge amount of articles, papers, reports, tools and case studies. The first selection were based on the abstracts on the papers. The introduction and conclusion sections of the selected papers were read, and those papers that were not proved to be interesting were filtered out.

Later, as the goal of the R&D activity became clearer, we added "test generation" as a search term, which resulted in works mostly concerning model based testing, random testing, automative test generation, symbolic execution, and some processes that use them.

As testing and debugging are close to each other (although they are different activities, both debugging and white-box testing are based on the program code and deals with execution data), "embedded system debug" terms were also searched and a few relevant papers were found.

We also processed the reference lists of the relevant papers we found. These referred to articles usually describing the basics of some techniques, or to different tools that utilize the described technique.

As the last step, tools supporting automatic test generation and/or test execution are searched and processed. Search terms that were included for this purpose were "automated test generation tool", "automated test execution tool", and "integrated test generation and test execution tool". Large amount of tools were found using these terms, and later filtered by selecting some of them according to given short descriptions and specifications. In order to get more precise information and to improve assessment of selected tools, more detailed documents addressing these tools (e.g. specifications, tutorials, etc.) were searched and processed.

At the last phase of tools assessment, still missing information of key importance for later evaluation and comparison of examined existing solutions, were searched by combining terms describing the information with the tool name. For some tools, when none of described method gave us the information, the tool was tried out using free (academic) licences or versions that are free for evaluation in the case they existed.

3 Classification and evaluation criteria

To evaluate and classify, and especially to compare the previous works, we had to set up some criteria.

At the beginning, we started to evaluate the articles without any fixed points of view. After processing some relevant papers, we compared their content and tried to list similarities and differences. This list were the base of setting up the classification and evaluation criteria.

Classification of methods/evaluation criteria:

input type Gives the input of the evaluated method. It can be a model, the source code, the binary, or various other representations of the system under test.

output/result Gives the output and/or result of the evaluated method. It can be a set of new or selected test cases, prioritization of test cases, coverage information, test execution results, or many other things, depending on the type of the method.

programming language This criterion denotes whether the evaluated method is specific to some programming languages or language families, or it is general in the means that could be (even if actually it is not) applied to any programming languages.

implemented/tool support This indicates whether the method is implemented fully or partially, or there are tools that supports this method.

applied in real environment An important property of a method is whether it is purely theoretical and works only for "toy" programs/environments, or it has been applied and its applicability has been proven in real scenarios.

specific to embedded systems Whether the evaluated method is specific to embedded systems environment, or it is general and can be effectively used not only in embedded systems.

use some coverage measure Indicates whether the method uses some kind of coverage values (e.g. code or functional coverage) as input.

computes some coverage measure Indicates whether the method computes some kind of coverage values (e.g. code or functional coverage) as output.

instrumentation technique If instrumentation is used in the method, this point gives the instrumentation technique (e.g. source code, binary, etc.)

requires source code Indicates if the method requires the source code of the system under test, or works from some other test basis.

BB testing method(s) This point indicates the general black-box testing methods that are specialized in the evaluated solution.

makes prioritization/selection This indicates whether the method includes some test case prioritization and/or selection functionalities, and shows what kind of selection / prioritization is used.

prioritization/selection based on Shows the base measure or data of the used test case prioritization/selection techniques (e.g. extent of code covered, time required for execution, etc.).

4 Assessment

In this section a detailed assessment of relevant papers and tools can be found. We separately evaluate black-box, white-box, and gray-box techniques and tools. At the end of the free-format evaluation of a paper, we give the answers to the classification and evaluation criterion in a table.

4.1 Black-box

In this section papers describing some black-box testing methods/activities are assessed. The focus is on those techniques that are more frequently or can more probably be used in embedded system testing.

Graph Transformations for Model-based Testing [13]

This paper presents an extended heuristic and a generic implementation of the classification tree. It uses the classification-tree transformer (CTT) tool to accomplish.

The classification-tree method is an instance of partition testing where the input domain of the test object is split up under different aspects, usually corresponding to different input data sources. The different partitions, called classifications, are subdivided into (input data) equivalence classes. Finally different combinations of input data classes are selected and arranged into test sequences.

The CTT tool needs the raw classification tree (as a model made in Matlab/Simulink/Stateflow; raw classification trees are automatically created by the model extractor) as input from the model-based development, and provide a complete classification tree. Then this complete tree can be used to generate model-based test scenarios by exploration. This extension is a tree-transformation with class definitions to partition the value space of input signals. For this test design step a number of heuristics have been developed which led to further automation steps:

- Data type related heuristics: e.g. the classification of a Boolean signal is set up by two classes true and false or enumeration types are classified by setting up a class for each enumeration value.
- Problem-specific partitioning heuristics: e.g. there is an interval of a variable's values, but there is a distinguished range of it, and a functionality can't be launch if the actual value is out of this range.
- General testing heuristics.

Besides, further tree transformations may be applied for structure refinement to simplify the tree. The transformation rules must be collected in sets to build up a library of test heuristics which can provide tree extension rules for specific application domains or different projects.

This paper mentioned that if we use some proper form of coverage we can generate more sensible inputs for the tests, but did not elaborate on details.

This approach is common in the embedded system development.

input type	Matlab/Simulink/Stateflow model
output/result	complete classification-tree
programming language	PROGRES
implemented/tool support	yes, the CTT tool supports it
applied in real environment	yes
specific to embedded systems	yes
use some coverage measure	no
computes some coverage measure	no
instrumentation technique	no
requires source code	no
BB testing method(s)	model-based
Makes prioritization/selection	no

Model based software testing [20]

This article shows and explains the main streams of the model-based testing.

Useful models in software testing:

- Finite-state machines:
  Finite state machines are applicable to any model that can be accurately described with a finite number (usually quite small) of specific states.
  A common scenario: the tester selects an input from a set depending on the prior results. At any given time, a tester has a specific set of inputs to choose from. This set of inputs varies depending on the exact "state" of the software. This characteristic of software makes state-based models a logical fit for software testing.

- State charts:
  State charts are specifically address modeling of complex or real-time systems. They provide a framework for specifying state machines in a hierarchy, where a single state can be "expanded" into another "lower-level" state machine. It involves external conditions that affect whether a transition takes place from a particular state, which in many situations can reduce the size of the model being created. State charts are probably easier to read than finite state machines, but they are also nontrivial to work with.

- UML:
  The unified modeling language models replace the graphical-style representation of state machines with the power of a structured language. It can describe very complicated behavior and can also include other types of models within it.

- Markov chains:
  Markov chains are stochastic models and they are structurally similar to finite state machines and can be thought of as probabilistic automaton. Their primary worth is generating tests and also gathering and analyzing failure data to estimate such measures as reliability and mean time to failure.

- Grammars:
  Different classes of grammars are equivalent to different forms of state machines. Sometimes, they are much easier and more compact representation for modeling certain systems such as parsers. They are generally easy to write, review, and maintain.

- Other: see in [16]

It gives proper terminology and examples, make a review of the MBT's role. It's aim to give an approach to the reader about the model-based testing methods and its functionality.

This paper not deals with coverage criteria, but tells some form of coverage that can be reached by MBT. The methodology not needs the source code to work. It needs only some kind of model. It can be applied widely in software development.

input type	some kind of model
output/result	usually test cases, scenarios
programming language	none
implemented/tool support	many tools supports
applied in real environment	yes
specific to embedded systems	no, but also used there
use some coverage measure	not common
computes some coverage measure	model, path coverage
instrumentation technique	no
requires source code	no
BB testing method(s)	model-based, sometimes with random inputs
Makes prioritization/selection	no

MaTeLo: Automated Testing Suite for Software Validation [29]

This paper presents the MaTeLo software, a model-based functional testing device, and its advantages, options and objectives. The developers not meant to make a device that fully tests a system, but to test a system to make it usable in the future without defects.

This device containing the follow issues:

- selecting relevant test cases:
  MaTeLo is generating the Test Suite from the Usage Model. The Test Suite can be analyzed by the MaTeLo system with a report generation, in order to generate a relevant Test Suite.

- giving the acceptance criteria of the testing and definition of a test stopping criteria:
  MaTeLo supports project manager to manage the test campaign. He will use the report's functions of MaTeLo to foresee the end of the project and so the delivery date of the system for customers. For tests, MaTeLo stores the model and computes some coverage criteria to give the satisfying conditions.

- helping the different development strategies:
  The industry is heightened at different stages regarding testing, and the MaTeLo project is committed to promote the use of statistical tools & methods to answer European industries' needs.

- test automation:
  MaTeLo provide support to build the software test plan and generate the usage model, than generate the test suite from it. MaTeLo provides the capability of automatic execution of test suite and stores test results in a database to allow further analysis.

It uses Markov-chains to generate test cases, because these give the proper user behavior models. The states of the Markov-chain represents the states of the system and the transitions in the Markov-chain refers to the user actions, so the state-changes in the system.

The MaTeLo contains many options to enhance model-based functional testing. It can provide the usage model from the specification, generate test cases from it in TTCN-3 or textual formats, and calculate coverage on specification and model level.

input type	some kind of model
output/result	test cases, scenarios
programming language	TTCN-3
implemented/tool support	it is a tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	specification, arc and state
instrumentation technique	no
requires source code	yes
BB testing method(s)	Markov-model-based
Makes prioritization/selection	can make selection

Automatic test case generation from requirements specifications for real-time embedded systems [15]

In this paper, the authors present a method to generate test cases, using the requirements specifications for event-oriented, real-time embedded systems. The requirements documentation and test case generation activities make up the initial steps in their method to realize model-based codesign. This codesign method relies on system models at increasing levels of fidelity in order to explore design alternatives and to evaluate the correctness of these designs. As a result, the tests that we desire should cover all system requirements in order to determine if all requirements have been implemented in the design. The set of generated tests will then be maintained and applied to system models of increasing fidelity and to the system prototype in order to verify the consistency between models and physical realizations.

In this codesign method, test cases are used to validate system models and prototypes against the requirements specification. This ensures coherence between the system models at various levels of detail, the system prototype, and the final system design. Automating the test case generation process provides a means to ensure that the test cases have been derived in a consistent and objective manner and that all system requirements have been covered. The goal is to generate a suite of test cases that provide complete coverage of all documented system requirements.

The paper contains a simple example of a controller for a safety injector of a reactor core. The system monitors pressure and adds coolant if the pressure drops below a given threshold.

The difficulty of this problem has been discussed in this paper and a heuristic algorithm is presented to solve the problem.

input type	requirements specification
output/result	test suite
programming language	C
implemented/tool support	yes
applied in real environment	no
specific to embedded systems	yes
use some coverage measure	yes
computes some coverage measure	no
instrumentation technique	-
requires source code	yes
BB testing method(s)	specification based
Makes prioritization/selection	can make selection
Prioritization/selection based on	specification

Automatic test generation: a use case driven approach [45]

The authors propose a new approach for automating the generation of system test scenarios from use cases in the context of object-oriented embedded software and taking into account traceability problems between high-level views and concrete test case execution. The method they develop is based on a use case model unraveling the many ambiguities of the requirements written in natural language. They build on UML use cases enhanced with contracts (based on use cases pre- and postconditions). The test objectives (paths) generation from the use cases constitutes the first phase of their approach. The second phase aims at generating test scenarios from these test objectives. The test cases are generated in two steps: Use case orderings are deduced from use case contracts; and then use case scenarios are substituted for each use case to produce test cases. While in the first step the use cases model handles high level concerns, in the second step, the data complexity (numerical data, object models, OCL constraints, etc.) is taken into account with the use of use case scenarios. The approach has been evaluated in three case studies by estimating the quality of the test cases generated by their prototype tools.

input type	UML use cases
output/result	test suite
programming language	C++
implemented/tool support	implemented, but no tool
applied in real environment	yes
specific to embedded systems	yes
use some coverage measure	use case coverage
computes some coverage measure	-
instrumentation technique	-
requires source code (yes/no)	no
BB testing method(s)	model-based
Makes prioritization/selection	-
Prioritization/selection based on	-

A Test Generation Method Based On State Diagram [39]

This paper aims to resolve the following research issues:

- minimize size of test cases and test data derived from extended state chart diagram,
- maximize a number of nodes coverage, and
- minimize total time of test case generation from diagrams.

The paper proposes an effective method to prepare and generate both of test cases and test data, called TGfMMD method. The TGfMMD method is developed to verify the state chart diagram before generation and generate both of test cases and test data from extended state chart diagram. The extended state diagrams is a Mealy Machine diagram. The Mealy Machine diagram is extended from the UML state diagram. Both of these diagrams are used to describe the behavior of systems but differ in the sense of Mealy Machine diagram has input and output while normal state diagram does not have.

input type	state diagram
output/result	test cases
programming language	-
implemented/tool support	TGfMMD
applied in real environment	no
specific to embedded systems	no
use some coverage measure	state diagram coverage
computes some coverage measure	diagram
instrumentation technique	no
requires source code	no
BB testing method(s)	method based
Makes prioritization/selection	selection
Prioritization/selection based on	state diagram coverage

A Practical Approach for Automated Test Case Generation using Statecharts [52]

This paper presents an approach for automated test case generation using a software specification modeled in Statecharts. The steps defined in this approach involve: translation of Statecharts modeling into an XML-based language and the PerformCharts tool generates FSMs based on control flow. Statecharts extend state-transition diagrams with notions of hierarchy (depth), orthogonality (parallel activities) and interdependence/synchronization (broadcast communication). Statecharts consist of states, conditions, events, actions and transitions.

These FSMs are the inputs for the Condado tool which generates test cases. A case study was on an implementation of a protocol specified for communication between a scientific experiment and the On-Board Data Handling Computer of a satellite under development at National Institute for Space Research (INPE). The approach was applied on a simulated version of a satellite experiment software. The results were satisfactory.

input type	Statechart
output/result	test cases
programming language	C++
implemented/tool support	implemented, but no tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	no
instrumentation technique	no
requires source code	no
BB testing method(s)	-
Makes prioritization/selection	-
Prioritization/selection based on	-

Testing Concurrent Object-Oriented Systems with Spec Explorer [9]
The basics of the SpecExplorer is the interface automaton [3], which separates the input and the output edges in the nodes and uses FIFO structure to explore the input model. SpecExplorer discovers the specification (high-level or source code) to build the interface model and than explores it to build the model which will be the basis of the test case generation. SpecExplorer can create not only fix scenarios but dynamic or infinite ones as well (e.g. chat servers) and can choose series of method calls which do not violate the system's operation and which are relevant for the users' test inputs.

It uses the next two methods for simplify the infinitive systems:

- grouping statuses: merge the statuses which are indistinguishable in a user define aspect;
- state-dependent parameter generating: defines parameter-intervals which can help us to select the proper input values.

The result graphs can use as oracles. To solve the branches SpecExplorer use Markov-decision logic. With this, it can provide a good path and model coverage.

input type	specification or model
output/result	test scenarios or a graph
programming language	C#, .NET
implemented/tool support	Visual Studio 2010 Ultimate and above, SpecExplorer
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	behavioral, branch
computes some coverage measure	code
instrumentation technique	.NET assembly level, binary inst.
requires source code	yes
BB testing method(s)	Markov-model-based
Makes prioritization/selection	can make both
Prioritization/selection based on	some user-defined aspect

Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution [53]
We can find proper inputs for parametrized unit tests (PUTs) during symbolic execution thus we can reach high model coverage and in some case we can look this PUTs as specification. During symbolic execution we explore the symbolic variables and develop them with proper values. The symbolic variables are mathematical structures that contains every variable from above in the path which the symbolic variable depends on.

PUTs can be provided from existing unit test or we can write brand news from the implementation.

In this paper these tools mentioned as providing symbolic execution:

- Java PathFinder with some extensions,
- .NET XRT.

The next two device was developed by Microsoft Research for automatic unit test generation: UnitMeister and AxiomMeister. These devices can make new PUTs from implementation, parametrize existing UTs and refactor existing PUTs. The symbolic variables are expressions over the input symbols. The symbolic execution builds up a dependency path between the variables thus it can compute the values for all the variables by choosing the proper input values. these dependency paths can contain junctions (so we call them trees more than paths) and the tree-exploration or tree-execution makes as much UTs as the number of the branches.

We can specify the minimal number of test scenarios by define the proper inputs so these scenarios can cover all the paths. A path is inappropriate if we can't find input for it. For example it will newer be chosen or in the branch the value is always false, etc. In this case we can drop this branch even from the system. The symbolic execution unfolds all the loops and recursions, so it can provide infinite number of paths. For prevent this, we can use several techniques. One of these is if we can give a number for limitation for running the loops by analyzing the behavior of the loops and gives a maximum number of the execution of the loop. We can use mock objects for imitate the behavior and functions of the software components. Though the mock objects contains only a slice of the functionalities, if we can generate these automatically, we can have unlimited number of mock objects, each with different functionality.
For this case the symbolic mock objects are the best choices. In these objects the functionalities are specified like the values of the symbolic variables (in dependency trees). We can represent each procedure calls result by mock objects.

input type	any kind of unittests or implementation
output/result	Parametrized Unit tests
programming language	Java, .NET
implemented/tool support	PathFinder, XRT, UnitMeister, AxiomMeister
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	input coverage
computes some coverage measure	model, path coverage
instrumentation technique	no
requires source code	can use the source code also
BB testing method(s)	BB, GB
Makes prioritization/selection	BB, GB
Prioritization/selection based on	BB, GB

Feedback-directed Random Test Generation [47]

This paper presents a technique that improves random test generation by incorporating feedback obtained from executing test inputs as they are constructed. Build inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously-constructed inputs. As soon as an input is built, it is executed and checked against a set of contracts and filters. The result of the execution determines whether the input is redundant, illegal, contract-violating, or useful for generating more inputs. Inputs that create redundant or illegal states are never extended into tests containing more steps. The technique outputs a test suite consisting of unit tests for the classes under test in object-oriented systems. This technique is implemented in RANDOOP, which is a fully automatic system, requires no input from the user (other than the name of a binary for .NET or a class directory for Java), and scales to realistic applications with hundreds of classes. It can be efficiently used in the sparse and global sampling. Inputs created with feedback-directed random generation achieve equal or higher block and predicate coverage than the systematic techniques. Feedback-directed random testing does not require a specialized virtual machine, code instrumentation, or the use of constraint solvers or theorem provers.

The basics if this technique is that an object-oriented unit test consists of a sequence of method calls that set up state (such as creating and mutating objects), and an assertion about the result of the final call. Each method have input arguments, which can be primitive values or reference values returned by previous method calls. The feedback-directed random test generation technique chooses a method randomly from the method list and generating inputs for it. When the input is generated, the method is executed and measured. If the result violates any constraint, the methods is dropped. If not, a new method is chosen from the available set. This set is made up from the methods that are reachable after the run of the previous one. The technique is iterating these steps until the program is terminating. The result is a test sequence from valid method calls and the proper inputs. As soon as a (sub)sequence is built, it is executed to ensure that it creates non-redundant and legal objects, as specified by filters and contracts.

RANDOOP takes all these steps automatically and makes a complete test suite of one library by one run.

input type	model or source code
output/result	test suite+inputs
programming language	.NET, Java
implemented/tool support	RANDOOP
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	method
instrumentation technique	none
requires source code	yes
BB testing method(s)	random
Makes prioritization/selection	no

Path Oriented Random Testing [28]

Test campaigns usually require only a restricted subset of paths in a program to be thoroughly tested, so we face the problem of building a sequence of random test data that execute only a subset of paths in a program based on backward symbolic execution and constraint propagation to generate random test data based on an uniform distribution.
Usual white-box testing approaches require only a subset of paths to be selected to cover all statements, all decisions or other structural criteria.
There are also paths which never will be chosen during the programs operation.

Our approach derives path conditions and computes an over-approximation of their associated sub-domain to find such a uniform sequence. One key advantage of Random Testing over other techniques is that it selects objectively the test data by ignoring the specification or the structure of the Program Under Test. Path testing requires to find a test suite so that every control flow path is traversed at least once. As every feasible path corresponds to a sub-domain of the input domain, path testing consists in selecting at least one test datum from each sub-domain with minimalizing the numbers of rejects in selected inputs. A reject is produced whenever the randomly generated test datum does not satisfy the path conditions.

This paper presents and explains the symbolic execution, the constraint programing, and gives some example algorithms how to calculate path condition and how to generate path-oriented random test data.

input type	control flow
output/result	test suite
programming language	SICStus Prolog, C
implemented/tool support	implemented, but no tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	path coverage
instrumentation technique	no
requires source code	no
BB testing method(s)	model-based, random input
Makes prioritization/selection	no

Adaptive Random Testing [10]

Adaptive random testing seeks to distribute test cases more evenly within the input space. It is based on the intuition that for non-point types of failure patterns, an even spread of test cases is more likely to detect failures using fewer test cases than ordinary random testing.

In recent studies, it has been found that the performance of a partition testing strategy depends not only on the failure rate, but also on the geometric pattern of the failure-causing inputs. This has prompted the authors of this article to investigate whether the performance of random testing can be improved by taking the patterns of failure-causing inputs into considerati.

This study assumes that the random selection of test cases is based on a uniform distribution and without replacement. Elements of an input domain are known as failure-causing inputs, if they produce incorrect outputs. We use the expected number of test cases required to detect the first failure (referred to as the F-measure), as the effectiveness metric. The lower the F-measure the more effective the testing strategy because fewer test cases are required to reveal the first failure. The patterns of failure-causing inputs have classified into three categories: point, strip and block patterns. It conjectures that test cases should be as evenly spread over the entire input domain as possible.

Adaptive random testing makes use of two sets of test cases, namely the executed set and the candidate set which are disjoint. The executed set is the set of distinct test cases that have been executed but without revealing any failure; while the candidate set is a set of test cases that are randomly selected without replacement. The executed set is initially empty and the first test case is randomly chosen from the input domain. The executed set is then incrementally updated with the selected element from the candidate set until a failure is revealed. From the candidate set, an element that is farthest away (Euclidean distance) from all executed test cases, is selected as the next test case. There are also various ways to construct the candidate set.

The authors make an experiment with many kind of open source programs in variety of programming languages but all programs have converted into C++.

The article gives an example algorithm to show how to generate a candidate set and select a test cases.

input type	input domain
output/result	test inputs
programming language	C++
implemented/tool support	it is implemented, but have no tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	no
instrumentation technique	no
requires source code	no
BB testing method	random
Makes prioritization/selection	no

4.2 White-box

In this section, papers that describe methods helping to extract some white-box coverage measures are assessed.

Observability analysis of embedded software for Coverage-Directed validation [14]

In this paper the authors propose a new metric that gives a measure of the instruction coverage in the software portion of the embedded system. Their metric is based on observability, rather than on controllability. Given a set of input vectors, their metric indicates the instructions that had no effect on the output.

The coverage metric being proposed was implemented to handle programs in the C language. The algorithm was implemented in a two step process. In the first step they transform the source program by adding for each statement a call to a function. The parser used was c2c which is a public-domain software program. c2c works by making an Abstract Syntax Tree (AST) of a C program. The AST can then be manipulated in several ways such as adding or deleting nodes in it. Finally, after changing the AST, the c2c tool produces the C program for that new AST.

In their case, the modifications made are, for each statement, adding one of several functions to the code. Several functions will process the information extracted from the statement.

Then, in the second step they compile the transformed program inside a framework that will allow several input vectors to be run and obtain an overall estimate of the observability coverage for these vectors. They show four examples they used to test the observability based metric being proposed. One of the program computes Fibonacci numbers, one matches a stream of characters against a string, one computes the Huffman code and the last one implements the Fast Fourier Transform (FFT). All four were implemented using the C language.

This metric has great potential to be used in embedded software testing. There is significant overhead due to the fact that for each statement, a function call is made.

input type	source code
output/result	percentage of observed statements
programming language	C/C++
implemented/tool support	implemented, but no tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	statement coverage
computes some coverage measure	statement coverage
instrumentation technique	code instrumentation
requires source code	yes
BB testing method(s)	-
Makes prioritization/selection	-
Prioritization/selection based on	-

Flow logic: a multi-paradigmatic approach to static analysis [46]

The flow logic is a formalism of static analysis. It separates when and how: when an estimation of an analysis is acceptable and how to make the analysis. It is based in particular on the conventional use-case analysis, border analysis and abstract interpretation. Definitions in different levels can be specified by the same formalism. It allows us to use the conventional techniques in static analysis. This is the basis of using different paradigms in different parts of the system according to what paradigm gives the best solution.

The specifications of the flow logic are sets of closes. It is necessary to write these closes co-inductively. An estimation of an analysis is acceptable if not violates any of the conditions set by the specification. We can reach a good specification coverage, if selects these kind of analysis.

There are two approaches of the flow logic:

- abstract vs. complex,
- succinct vs. verbose.

The complex specification is syntax-driven, similar to the implementation, while the abstract specification is close to the common semantics. The verbose specification reports all the inner flow information like the use-case and the boarder analysis, while the succinct specification deals only with the top level estimation of an analysis.

input type	source code, implementation, interface
output/result	sets of closes
programming language	none
implemented/tool support	no
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	specification
instrumentation technique	no
requires source code	yes
BB testing method(s)	use-case analysis, border analysis, abstract interpretation
Makes prioritization/selection	no

Boundary Coverage Criteria for Test Generation from Formal Models [40]

This article presents a new area of the model-based coverage criteria, which is based on the formalism of the boundary-testing heuristics. It can be applied in every system working with variables and values. It feasible to measure coverage or to generate test cases. It is implemented in the B-Z-TESTING-TOOLS tool suite, which is able to generate test cases from B, Z or UML/OCL model.

They tried and suggested a number of coverage metric in the early development:

- Transition coverage or transition-pair coverage for transitions represented in state-chart;
- Constraint coverage for abstract state machines' behavior-defining constraints;
- Disjunctive Normal Form coverage for states in state-based models, like B, Z, VDM, where predicates provides the behavior.

Besides, there are different analyzing methods to provide the basis for test generating algorithms, but they aren't used as coverage metrics. One from these is the boundary-analysis. The boundary coverage is independent from the structure, so it can be an extension for it. It's suitable for selecting or extending the test cases generated from structural coverage. This BZ-TT tool suite have special possibilities to efficiently implement the boundary value computing method, and it is commonly used for smart cards and in transport systems. The formal model used by the BZ-TT is assembled from variables and predicates and can be created from any kind of formal specification.

This article gives a formal definition for the boundary values, the boundary coverage, and a test selection algorithm, and gives a particular formal example.

input type	formal model
output/result	boundary coverage value
programming language	B, Z, VDM, UML/OCL
implemented/tool support	implemented in BZ-TESTING-TOOLS
applied in real environment	yes
specific to embedded systems	partly
use some coverage measure	no
computes some coverage measure	boundary
instrumentation technique	no
requires source code	no
BB testing method(s)	no
makes prioritization/selection	can make selection
Prioritization/selection based on	boundary coverage

A Dynamic Binary Instrumentation Engine for the ARM Architecture [31]

Dynamic binary instrumentation (DBI) is a powerful technique for analyzing the runtime behavior of software. There are numerous DBI frameworks for general-purpose architectures, but for embedded architectures are fairly limited.

This paper describes the design, implementation, and applications of the ARM version of Pin.

ARM is an acronym for Advanced RISC Machines. Most implementations of the ARM architecture focus on providing a processor that meets the power and performance requirements of the embedded systems community.

Pin is a dynamic binary rewriting system developed by Intel. It allows a tool to insert function calls at any point in the program and automatically saves and restores registers so the inserted call does not overwrite application registers. At the highest level, Pin consists of a virtual machine (VM), a code cache, and an instrumentation API invoked by Pintools. The VM consists of a just-in-time compiler (JIT), an emulator, and a dispatcher. The JIT compiles and instruments application code, which is then launched by the dispatcher.

Since Pin sits above the operating system, it can only capture user-level code. It uses a code cache to store previously instrumented copies of the application to amortize its overhead. Code traces are used as the basis for instrumentation and code caching.

Pin provides transparency to any application running under its control. All memory and register values, including the PC, will appear to the application as they would had the application been run directly on the hardware.

To ensure that the VMmaintains control of execution at all times, and control never escapes back to the original, not instrumented code, all branches within the cached code are patched and redirected to their transformed targets within the code cache.

From an ISA standpoint, system calls do not present any particular problem in Pin for ARM, since they can be executed directly without further intervention from Pin. However, in order to stay in control of the application under all circumstances, some system calls must be intercepted and emulated instead.

Superblocks (single-entry, multiple-exit regions) are used as the basis for instrumentation and code caching in Pin. Just before the first execution of a basic block, Pin speculatively creates a straight-line trace of instructions that is terminated by either an unconditional branch, or an instruction count limit. One ARM-specific trace selection optimization we explored was to limit trace lengths to a fixed maximum number of basic blocks. This optimization reduces the tail duplication resulting from caching superblocks.

A major challenge in many dynamic instrumentation systems is self-modifying code (SMC). Any time an application modifies its own code region, the instrumentation system must be aware of this change in order to invalidate, regenerate, and re-instrument its cached copy of the modified code. The real problem is the efficient detection. Fortunately, architectures such as ARM contains an explicit instruction that must be used by the software developer in order to correctly implement SMC.

After these, the article shows a performance analysis to Pin for ARM. Finally it lists out the potential applications.

input type	embedded system
output/result	instrumented system
programming language	C
implemented/tool support	implemented in Pin for ARM
applied in real environment	yes
specific to embedded systems	yes
use some coverage measure	no
computes some coverage measure	no
instrumentation technique	binary
requires source code	no
BB testing method(s)	none
Makes prioritization/selection	no

Automated Formal Verification and Testing of C Programs for Embedded Systems [36]

This paper introduces an approach for automated verification and testing of ANSI C programs for embedded systems. Automatically extract an automaton model from the C code of the system under test. This automaton model is used for formal verification of the requirements defined in the system specification, and we can derive test cases from this model by using a model checker, too. This paper specifically shows how to deal with arithmetic expressions in the model checker NuSMV and how to preserve the numerical results in case of modeling the platform-specific semantics of C.

In this paper the verification of the SUT is realized in two important independent steps:

- In the first step the platform-independent semantics of the system can be verified formally by model checking. By verifying all requirements from the specification, it can be shown that the C program conforms to the specification. Verifications are done with X-in-the-loop method.
- The second step is testing the system by execution of test cases on the target platform. It proves whether the platform-specific semantics of the program has the same behavior as the model. Test cases are generated by model checking from the automaton model.

Every step is done in Matlab Simulink.

The model extraction is done in the following steps: (1) The C-source code is parsed and by static analysis, the syntax tree of the program is generated. (2) The syntax tree is used to generate the automaton model by sequentially processing it and interpreting the semantics of the basic statements. (3) The description of the automaton model is given in an automata language.

For the formal verification of the system the properties from the specification have to be translated into temporal logic formulas. These formulas can be verified on the model with a model checker. Some properties from the specification are suitable to be checked directly on the extracted model.

For the test case generation we also use model checking techniques. The main purpose of a model checker is to verify a formal property on a system model. In case that the formal property is invalid on a given model, a model checker provides a counterexample, which describes a concrete path on which the property is violated. This feature of a model checker can be used to generate test cases in a formal and systematic way. For finding suitable test cases the challenge is to find appropriate properties (trap properties), that yield specific paths that can be used as test cases.

input type	specification
output/result	test cases and verification information
programming language	C
implemented/tool support	NuSMV
applied in real environment	yes
specific to embedded systems	yes
use some coverage measure	no
computes some coverage measure	no
instrumentation technique	none
requires source code	no
BB testing method(s)	model checking
Makes prioritization/selection	no

Using Property-Based Oracles when Testing Embedded System Applications [61]

As prior work in this paper an approach for testing embedded systems is presented, focusing on embedded system applications and the tasks that comprise them. This article focuses on a second but equally important aspect of the need to provide observability of embedded system behavior sufficient to allow engineers to detect failures. It presents several property-based oracles that can be instantiated in embedded systems through program analysis and instrumentation, and can detect failures for which simple output-based oracles are inadequate.

The authors presented an approach in this paper to help developers of embedded system applications detect faults that occur as their applications interact with underlying system components. This approach involves two dataflow-based test adequacy criteria. First, we use dataflow analysis to identify inter-layer interactions between application code and lower-level (kernel and hardware-related) components in embedded systems. Second, we use a further dataflow analysis to identify inter-task interactions between tasks that are initiated by the application. Application developers then create and execute test cases targeting these interactions.

The "oracle problem" is a challenging problem in many testing domains, but with embedded systems it can be particularly difficult. Embedded systems employing multiple tasks that can have non-deterministic outputs, which complicates the determination of expected outputs for given inputs. Faults in embedded systems can produce effects on program behavior or state which, in the context of particular test executions, do not propagate to output, but do surface later in the field. Thus, oracles that are strictly "output-based", may fail to detect faults. So several "property-based" oracles are presented that use instrumentation to record various aspects of execution behavior and compare observed behavior to certain intended system properties that can be derived through program analysis. These can be used during testing to help engineers observe specific system behaviors that reveal the presence of faults.

input type	program and test suit
output/result	test results
programming language	C, Java
implemented/tool support	no
applied in real environment	yes
specific to embedded systems	yes
use some coverage measure	trace
computes some coverage measure	trace
instrumentation technique	source code, OS, libraries, runtime systems
requires source code	yes
BB testing method(s)	none
Makes prioritization/selection	no

A Model-Based Regression Test Selection Approach for Embedded Applications [7]

A compound model-based regression test selection technique for embedded programs is proposed in this paper. Also proposed a graph model of the program under test (PUT). The authors mention to select a regression test suite based on slicing this graph model. They also propose a genetic algorithm-based technique to select an optimal subset of test cases from the set of regression test cases after this selection.

The embedded systems' advancement entails the growing complexity of the embedded programs. Object-oriented technologies are being increasingly adopted for development because of the advantages they offer to handle complexity.

Every software product typically undergoes frequent changes in its lifetime to fxing defects, enhancing or modifying existing functionalities, or adapting to newer execution environments. But this means also that the satisfactory testing of the embedded programs has turned out to be a challenging research problem.

For testing, we need a huge set of test cases, which we need to execute for regression testing. To save the resources during regression testing we can select a subset from the regression test set and execute only this subset of test cases. These are mostly the test cases that executes the modified parts of a program. Test cases which tests a part of the program that has been deleted during a modification can also be removed from the regression test set. Unfortunately, many test cases that would detect regression errors are not selected so we need to chose the test selection method wisely.

There are many test selection algorithms, but only few of them are suitable for embedded systems. Moreover, if this system is large, complex and different parts of it are written in different languages, than the traditional source-analyzing methods are useless. The new approach proposed in this paper is the model-based regression testing and test selection. The authors use a graph model that is constructed with program analysis. This model can also be used for prioritizing the regression test cases and selecting an optimal test suite.

Briefly the different steps involved in the approach presented in this article:

- The Intermediate Model Constructor constructs the intermediate model for the original program.

- The Code Instrumenter instruments the original program, and the instrumented code is executed on the initial test suite by the Program Execution module.
- The Model Differencer analyzes the modified source code and identifies the model elements that are modified and tags those elements on the model.
- The Slicer performs a forward slice on the modified marked model to identify the affected model elements that need to be retested.
- The Optimizer analyzes additional information about the program components gathered from the operational profile, and prioritizes the test cases based on the criteria used in the operational profile module.
- Subset of test cases than selected.

In the next section this paper shows the inadequacy of existing graphical models to embedded systems and shows an extended one from them that is suitable for embedded program's regression test selection. The article shows the additional features of the model in detailes. These features are the representation of the control flow, exception handling and information representation from design models.

The authors also shows a method briefly for test selection and for the test suit optimisation.

input type	program and test suit
output/result	test set
programming language	C, Java
implemented/tool support	no
applied in real environment	yes
specific to embedded systems	yes
use some coverage measure	no
computes some coverage measure	model
instrumentation technique	source code, model
requires source code	yes
BB testing method(s)	model-based
Makes prioritization/selection	yes

4.3 Grey-box

In this section white-box aided black-box testing methods (specially, coverage aided random testing, test case prioritization and selection) are assessed.

Achieving both Model and Code Coverage with Automated Gray-box Testing [38]

The Microsoft Research have developed a device for helping black-box testing. It makes a tree from the specification by model checking and makes Model-Based Tests by discovering the paths in this tree. This device is the Spec Explorer.

An other device developed by them, the Pex, is helping White-Box Testing by making parametrized unit tests from program-trees and specifies the inputs itself. It collects informations during the execution to make better random inputs and to groups the paths that have the same outcome. The execution stops when all inputs are tried or all groups are defined. In this way, Pex can provide good path coverage.

Both device can be integrated into Visual Studio thus they are very effectively usable. Combined usage computes the minimal number of parametrized unit tests which provides high coverage.

The Spec Explorer is able to leave variables symbolic during the discover of the specification. This process is building up a mathematical structure about the interdependence of the variants. The result is a program-tree which discovered by Pex, that provides not only inputs, but relevant values for the symbolic variants. In this way we can provides better coverage and reduce the number of necessary unit tests.

Pex is monitoring the data and control flow by instrumenting the source code and gives reports about bugs and coverage.

We can build up the model (tree, data flow, control flow) manually with Spec Explorer by the provided notation and style. Next, running the Spec Explorer on this model is providing the parametrized unit tests in C# and also compile these. Then Pex is using a symbolic execution on these tests to compute the inputs and the values for the symbolic variables.

input type	specification, implementation
output/result	program-tree, test scenario, test inputs
programming language	C, C++, C#, .NET
implemented/tool support	SpecExplorer, Pex
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	model (path, branch, etc.)
instrumentation technique	instruction level, code instrumentation
requires source code	yes
BB testing method(s)	model-based
Makes prioritization/selection	yes
Prioritization/selection based on	paths in the program tree

Generating Test Cases from UML Activity Diagram based on Gray-Box Method [41]

The authors proposed an approach to generate test sequences directly from the UML activity diagram using a gray-box method, where the design is reused to avoid the cost of test model creation. The paper shows that test scenarios can directly derive from the activity diagram that modeling an operation. Therefore, all the information, such as test sequences or test data, is extracted from each test scenario. Gray-box testing method, in the designers' viewpoint, generates test sequences based on high level design models which represent the expected structure and behavior of the software under test. Those specifications preserved the essential information from the requirement, and are the basis of the code implementation. The design specifications are the intermediate artifact between requirement specification and final code. Gray-box method extends the logical coverage criteria of white box method and finds all the possible paths from the design model which describes the expected behavior of an operation. Then it generates test sequences which can satisfy the path conditions by black box method and provide high path, structure, method and model coverage.

input type	UML Activity Diagram
output/result	test scenario
programming language	special UML
implemented/tool support	implemented, but no tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	behavior, method, model, path, structural
instrumentation technique	no
requires source code	no
BB testing method(s)	model-based
Makes prioritization/selection	no

DART: directed automated random testing [27]

The authors of this paper want to eliminate the handwritten test drivers and test harnesses and give an automatism to generate these thus make the test environment. To reach this goal they developed an approach, DART, which contains the three techniques below:

- retrieve the interface and the harness of the program automatically by static code analysis,
- automatic test driver generation for this interface, which simulates the most common harness of the program by random testing,
- dynamic behavior analysis during tests to generate the next inputs thus we can systematically control the execution between the alternative paths.

In testing, DART can reveal the regular errors like program crush, assertion violation, infinitive running. DART makes an instrumentation on the code in the level of RAM machine, collects data during running and calculates values in the executed branch. By these informations DART defines the inputs for the next execution thus an other branch will be covered. The first inputs are random values. Repeating the execution we can cover all the branches in the program tree (branch/path coverage). DART can run symbolic and real executions parallel.

input type	source code
output/result	interface graph, test driver, test inputs
programming language	C, C++, Java
implemented/tool support	it is implemented, but not have a tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	path, branch
instrumentation technique	in RAM-machine level
requires source code	yes
BB testing method(s)	model or graph based
Makes prioritization/selection	no

Robust test generation and coverage for hybrid systems [35]
This paper presents how to develop a framework for generating tests from hybrid systems' models. The core idea of the framework is to develop a notion of robust test, where one nominal test can be guaranteed to yield the same qualitative behavior with any other test that is close to it.

Our approach offers three distinct advantages:

1. It allows for computing and formally quantifying the robustness of some properties;
2. It establishes a method to quantify the test coverage for every test case;
3. The procedure is parallelizable and therefore, very scalable.

The ultimate goal of testing is to cover the entirety of the set of testing parameters so in the end provide high path and model coverage.

When the set of testing parameters is an infinite set, it is obvious that we cannot exhaustively test each of the testing parameters. However, it is possible that one testing parameter is representative of many others. A testing parameter is said to be robust if a slight (quantifiable) perturbation of the parameter is guaranteed to result in a test with the same qualitative properties. Robustness can lead to a significant reduction in the set of testing parameters.

They use a specific bi-simulation, where are no inputs, but properties. This bi-simulation is symmetric and somehow same to pairwise testing.

input type	model
output/result	set of robust tests + inputs
programming language	none
implemented/tool support	implemented, but not have a tool
applied in real environment	yes
specific to embedded systems	yes
use some coverage measure	no
computes some coverage measure	model, path
instrumentation technique	no
requires source code	no
BB testing method(s)	model-based, random seed for inputs
Makes prioritization/selection	selection
Prioritization/selection based on	robustness

Specification Coverage Aided Test Selection [50]

This paper considers test selection strategies in formal conformance testing. Ioco [56] is used as the testing conformance relation, and extended to include test selection heuristic based on a specification coverage metric. The proposed method combines a greedy test selection with randomization to guarantee completeness. Bounded model checking is employed for lookahead in greedy test selection.

It is particularly useful in testing implementations of communication protocols like as tele- and data communication fields. Formal conformance testing formalizes the concepts of conformance testing.

Essential notions, like ioco, include the implementation, the specification and conformance relation between these two. Ioco is defied by restricting inclusion of out-sets to suspension traces of the specification. It uses labeled transition system to introduce conformance relation.

Using coverage that measures the execution of all the lines of a source code at least once is a good choice to enhance test selection. Unfortunately, in black box testing this is not possible, because we do not know the internals of the actual implementation. From a pragmatic point of view, if the implementation is made according to the specification (or vice versa) it is somewhat likely that they resemble each other. Therefore this paper takes the assumption that in many cases arising in practical test settings, specification based coverage can "approximate" coverage used in white box testing.

This paper describes the used labeled transition system's notation, the ioco conformance relation, on-the-fly testing, petri nets, and in the end, it describes the developed test selection methodology and algorithm.

They extended an on-the-fly algorithm from an other work [18].
The first extension is to keep track of the used coverage metric.
The second change is to use the HeuristicTestMove algorithm as the TestMove subroutine. It will call a greedy coverage based test selection subroutine. If the greedy test selection subroutine could not provide anything, it calls the already presented random test selection subroutine.

input type	test set
output/result	selected test's set
programming language	none
implemented/tool support	implemented
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	specification coverage
computes some coverage measure	no
instrumentation technique	no
requires source code	no
BB testing method(s)	random with greedy selection
Makes prioritization/selection	selection
Prioritization/selection based on	specification coverage

4.4 Tools

In this section, the overview of existing solutions in the field of automated software and hardware testing is given. The most information and theoretical knowledges are still offered by achievements in domain of academic research, with huge number of published scientific papers and tools developed through the realization of international projects. Beside, this section analyses industrial solutions for automated testing that are more functional and less complicated for both installation and usage unlike the academic solutions (this is justified by the fact that their continuous development and improvement are provided by the company). Finally, significant source of information is the database of patents, due to the tendency of many companies to protect their intellectual Property.

Majority of these tools are intended for testing both software and hardware. When the hardware of embedded systems is tested, custom interfaces (in terms of software) are used for that purpose. These interfaces interact with the system by controlling and observing it through general interfaces (ports) that the system already has (in the case of black-box), or by making special support for testing. Support added for testing purposes can be consisted of both hardware (e.g. adding debug interface) and software (adding support for communication with testing interface through dedicated debug interface or through existing interface like COM port, Ethernet, different serial interfaces, etc.).

Based on the relationship of the process of generating and executing tests, the existing approaches in the field of automated testing can be divided into the following groups of solutions:

- Automated test generation (for off-line execution),
- Automated test generation integrated with test execution (on-line testing),
- Automated test execution (off-line testing).

Some solutions additionally offers support for off-line test analysis.

The Overview of Existing Approaches and Tools for Automated Model-Based Test Generation

MaTeLo Tool for making the model of system, model check, generation of test scenarios based on the given model and the analysis of test execution results [22].
The starting point of the modeling is the specification that describes the usage of the system with certain level of abstraction. The model of the system is consisted of the states and transitions among them with assigned probabilities (the model describes expected usage of the system and is based on Markov chains). One of the biggest challenge during the modeling is giving precise probability distributions. Tests are generated by making patch through the model according to one of following criteria for test steps selection: Chinese postman algorithm (tests are generated to cover all transitions, disregarding the probability distribution) and selection on the principle of probability (leaving a state, the transition with the highest probability is elected). Though supported test formats are TTCN-3 and XML, the tool generates tests in several special-purpose formats adapted to customers (National Instruments TestStand, MBtech PROVEtech, IBM Rational Functional Tester, HP QuickTest Professional, SeleniumHQ). Test results analysis gives information like model coverage, reliability of software/hardware, mean time to failure, and failure probability. The tool is intended for functional testing, testing of integration and acceptance in the field of embedded systems. During the usage of the tool, following deficiencies are observed:

- The size of the test set that can be generated in one pass is limited to 400,
- There is no support for the calculation of the number of the tests required to achieve desired reliability of the system.

The tool is developed through the international project of the Fifth Framework Programme (FP5). Nowadays, it is own by the All4tec company and is available on the market as a commercial solution requiring an appropriate license.

input type	some kind of model
output/result	test cases, scenarios
programming language	TTCN-3, XML, user defined
implemented/tool support	tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	specification, arc and state
instrumentation technique	no
requires source code	no
BB testing method(s)	Markov-model-based
Makes prioritization/selection	can make selection

mbt Open source tool for automated generation of test scenarios according to the model [37]. It doesn't support graphical presentation of the model, thus the model given in .graphml format is required to be passed as input parameter (it doesn't use UML format, avoiding unnecessary complexity). For making the model, yEd tool could be used. The model is consisted of the states and transitions among them with assigned probabilities. As the criteria of test selection

A* algorithm and random selection, coverage of states and transitions and others are used. Beside generating tests for later (indirect) execution, generating integrated with execution (on-line testing) is also supported.

input type	model in GraphML
output/result	test cases, scenarios
programming language	Java
implemented/tool support	tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	yes
instrumentation technique	no
requires source code	no
BB testing method(s)	model-based
Makes prioritization/selection	selection

TorX The tool for automated generation of test scenarios for testing the compliance of the system with a certain standard, intended for the class systems whose operating mode involves interaction with the environment (reactive systems), e.g. embedded systems, communication protocols, etc. [55]. Tests are derived from system behavioral model and some environmental aspects could be partially described also (system's environment model). For generation of tests scenarios the ioco algorithm is used, which aims the definition of finite test set which will discover as much errors as possible during testing with limited duration. Test scenarios are selected on several ways: randomly, by usage of ad hoc test specification, based on some heuristics, or by the criteria of model coverage. In earlier versions, the tool supported integrated test generation and execution only (on-line testing), i.e. test scenarios were generated as needed during the execution. The regime where previously prepared test set is used in execution (off-line testing) is enabled later. Basic characteristics of the tool are flexibility and openness. The flexibility provides simple substitution of any component of the tool with the improved one, while the openness relates to the possibility of adding new independent (third-party) components. The tool supports repeated execution of test sets derived from different specifications, with different configurations, and the like (test campaign). Additionally, archiving results on a systematic way is supported. The tool is used in several studies. Lucent R&D Center Twente is successfully used by TorX for testing of network protocols [55]. The tool is also used for testing the system for conference protocol [19] and for testing the highway tolling system [17]. However, some deficiencies of the tool are observed during the usage [55]:

- Insufficient support for testing the real-time applications, and
- Bad performance of generating test scenarios.

Other deficiencies of the tool that are observed:

- No possibility for model analysis (e.g. model coverage) and the analysis of test results,

- No possibility for assigning the probability of transitions between states,
- Big complexity of installation and configuration of the tool, and
- Though the tool supports separated generation and execution of tests (off-line testing), the documentation about that is not available.

Though the tool is available for academic researches [26], the complexity of the process of installation and configuration limits its practical application to a large extent. Moreover, studies in which the tool was used were performed or assisted by the author of the tool. The aforementioned reasons have contributed to the development JTorX tools.

input type	behavioral and environment model
output/result	test cases, scenarios
programming language	any
implemented/tool support	tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	yes
instrumentation technique	no
requires source code	no
BB testing method(s)	ioco algorithm
Makes prioritization/selection	selection
Prioritization/selection based on	randomly, ad hoc test specification, heuristics, criteria of model coverage

JUMBL The tool for statistical model-based testing [49]. It is developed in Java programming language, in order to be platform independent. TML language (notation for description of Markov chains) is used for the model description. The model is consisted of the states and the transitions among states related to pairs of input events and corresponding probabilities. The tool doesn't support graphical model description, but the model parameters are given in text format, through the command line. The tool supports model analysis in terms of model size, expected length of the test scenario, expected duration of retention in the each state of the model during testing, expected number of occurrences for each state and transition in the test scenario, etc. JUMBL enables the analysis of test results and the measure of tested system reliability. Calculation of system reliability is based on the previously proposed model [44]. In first step, the best reliability is calculated, i.e. the reliability that will be achieved if all tests pass once they are executed. This step doesn't require execution of tests and serves to calculate the size of test set needed for achieving desired reliability level. In the next step, real reliability is calculated as the ratio of successfully and unsuccessfully executed tests. The deficiency of the tool is lack of support for graphical notation of the model and, more important, though the tool was originally available for academic usage, currently it is not.

input type	model in TML language
output/result	test cases, scenarios
programming language	Java
implemented/tool support	tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	yes
instrumentation technique	no
requires source code	no
BB testing method(s)	Markov-model-based
Makes prioritization/selection	selection

TGV The tool for generation of the tests intended for verification of compliance of the system with the standard in the area of the protocol [57]. The model of the system under test is based on the principal of labeled transitions (labeled transition systems). Ioco algorithm is used for the generation of test scenarios, with the criteria for test selection defined by test specification. The tool supports the assignment of time controls at the time of test execution [23]. E.g. time control is started in the moment then input event is expected. If the input event happens, time control is stopped. Otherwise, the test execution is considered as unsuccessful. The tool is used in the studies of protocol testing [34].

input type	labeled transitions model
output/result	test cases, scenarios
programming language	TTCN
implemented/tool support	tool for protocol testing
applied in real environment	no
specific to embedded systems	no
use some coverage measure	branch
computes some coverage measure	yes
instrumentation technique	no
requires source code	no
BB testing method(s)	ioco algorithm
Makes prioritization/selection	selection
Prioritization/selection based on	specification

AETG The generator of inputs for combined model-based testing [11]. In combined testing approach, test scenarios are defined so that all the combinations of test parameters are covered (user inputs, internal and external parameters, etc.). Number of these test scenarios could be huge in practice. The tool provides optimal selection of double, triple and quadruple inputs, i.e. it defines inputs, but it doesn't support providing of expected outputs which are necessary in the case of automated testing. Though the tool models system environment, there is no support for describing the behavior of system under test. AETG is commercial tool intended for testing different configurations of device or any other product where parameters selection is important. It is used in several studies for testing compliance with the protocol specification.

input type	some model
output/result	test cases, scenarios
implemented/tool support	tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	n-*way coverage
instrumentation technique	no
requires source code (yes/no)	no
Makes prioritization/selection	selection
Prioritization/selection based on	all parameters covered

LTG Commercially available tool for the generation of tests intended for the testing of the systems that reacts to the stimuli from the environment, embedded systems and applications for electronic transactions [6]. The generation of tests is based on the system usage model, where the coverage of the model is used as the criteria for test selection. The tool is used for testing of the smart card applications [8].

input type	system usage model
output/result	
implemented/tool support	tool
applied in real environment	yes
specific to embedded systems	yes
use some coverage measure	yes
computes some coverage measure	yes
instrumentation technique	no
requires source code	no
Makes prioritization/selection	selection
Prioritization/selection based on	coverage

Conformiq Tool Suite The Conformiq company provides the Conformiq Tool Suite for modeling the system and for automated generation of model-based test scenarios [12]. It is possible to describe the model graphically (UML notation) or textually (QML - Qtronic Modelling Language, based on Java and C# languages) [32]. Beside the generation of test set for later execution (off-line testing), the test generation integrated with test execution is also supported (on-line testing). It is possible to use the tool from Eclipse environment or as the standalone tool. It is available for both Windows and Linux operating systems. It supports several test file formats: TCL, TTCN-3 Visual Basic, HTML, and XML. The tool is available with commercial license.

input type	UML or QML model
output/result	test cases, scenarios
programming language	Python, TCL, TTCN-3, C, C++, Visual Basic, Java, Junit, Perl, Excel, HTML, Word, Shell Scripts
implemented/tool support	tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	yes
instrumentation technique	no
requires source code	no
Makes prioritization/selection	selection
Prioritization/selection based on	state coverage, transition coverage, 2-transition coverage, Boundary Value Analysis, Branch Coverage, Atomic Condition Coverage, Method Coverage, Statement Coverage, Parallel Transition Coverage

Spec Explorer Microsoft introduced the Spec Explorer tool designed to test the software on the principle of modeling [43]. Behavioral model is generated by the software based on the source code and defined by C# programming language. The model is also represented as the graph for easier readability for the user. After verifying the correctness of the model, test scenarios are generated. Spec Explorer is an extension of Microsoft Visual Studio tool set, and is supplied as an integral part since the version 2010 of Visual Studio.

Microsoft has patented a method and system for software testing and modeling of user behavior [2]. Aspects of using the software under test are described by the model, which is then used to generate tests. The method uses several algorithms for test execution, depending on the goal of testing: Chinese postman algorithm, the selection of test steps in a random manner or contrary to the principle of random selection, i.e. the next test step is one that has not previously been selected.

input type	specification or model
output/result	test scenarios or a graph
programming language	C#, .NET
implemented/tool support	Visual Studio 2010 Ultimate and above, SpecExplorer
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	behavioral, branch
computes some coverage measure	code
instrumentation technique	.NET assembly level, binary inst.
requires source code	yes
BB testing method(s)	Markov-model-based
Makes prioritization/selection	can make both
Prioritization/selection based on	some user-defined aspect

JTorX The successor of TorX tool, developed to remove some of the drawbacks of the previous version [5]. TorX is developed to support the flexibility and openness, while some important features such as ease of installation, multi platform support, ease of use, and others are ignored. JTorX is developed using Java programming language, thus facilitating the installation. Also, added a graphical user interface, which enables easy configuration of the tool. Besides improved ioco algorithm for test generation [54], JtorX supports uioco algorithm. One feature that characterizes this particular tool and distinguishes it from similar tools is the advantage for use in teaching. JTorX is available for academic purposes [25].

input type	behavioral and environment model
output/result	test cases, scenarios
programming language	Java
implemented/tool support	tool
applied in real environment	yes
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	yes
instrumentation technique	no
requires source code	no
BB testing method(s)	improved ioco algorithm, uioco algorithm
Makes prioritization/selection	selection
Prioritization/selection based on	randomly, ad hoc test specification, heuristics, criteria of model coverage

AGEDIS The tool for automated model-based testing of distributed systems. It consolidates the environment for model description (UML model description), the model-check, test generation, model coverage analysis, test execution, the analysis of detected failures, and the generation of testing reports, [33, 30]. The tests are generated by the kernel of TGV tool, while the analysis of model coverage is realized with FoCuS tool [4]. Test execution is supported in distributed work regime. The tool was at first available for academic purposes, however, it is not maintained and currently not available.

input type	UML model
output/result	test cases, scenarios, reports
programming language	Abstract Test Suite (ATS)
implemented/tool support	tool
applied in real environment	no
specific to embedded systems	no
use some coverage measure	no
computes some coverage measure	yes
instrumentation technique	no
requires source code	no
BB testing method(s)	based on coverage of inputs to the model
Makes prioritization/selection	no

The Overview of Existing Approaches and Tools for Automated Test Execution Sony has patented a system for automatic testing of TV sets, which is a unit testing approach using a black box [59]. The tests consist of a series of sequences that are sent in the first step to the TV. After processing, output signals from the TV are recorded and compared with expected according to the relevant principles. The system consists of: (i) the unit to record the TV output, (ii) devices for the TV remote control, (iii) a PC that performs the appropriate application for testing and is associated with a database to store the tests, and (iv) test results. Another solution patented by Sony in the field of system testing is the system for automated testing of consumer electronics devices (audio / video devices, TV sets), with a focus on device performance testing [24]. Unlike previous solutions, communication with the tested appliance is accomplished via the command codes that are transmitted wireless. Similar to the previous design, the system is designed to test the video quality on the TV. Unlike the previous one, this solution verifies the memory consumption of the test device.

Philips has patented a system and method for automated testing of the TV sets [51]. The system consists of a unit that sends digital video signals to the TV as inputs and, after processing the test signal, receives output video signals from the TV. Processing unit performs comparison of the reference and the output (test) signal and, based on appropriate algorithms, evaluates the quality of video signal from the TV. Jitter, SNR (signal-to-noise ratio) measure, and blocks' similarity percentage are used as the criteria for comparison of test and reference signals.

The company Hon Hai Precision Industry has patented a system for automated performance measurement for set-top box devices [42]. The system consists of the audio and video test signals source, the testing process controller (PC), and the encoder and analyzer of audio and video signals. Based on the content of the test scenario, the controller of the testing process triggers sources of audio and video signals, to generate test signals for the system under test. The signal is then converted to the corresponding data stream format and transferred to the system under test. By passing of a given data stream through the system, output test signal is received. Based on the test scenario, the controller of the testing process sets the parameters of audio and video signals' analyzer. Test signal is analyzed according to these parameters. The system is applicable for audio and video signal analysis and performance measurement for set-top-box devices.

5 Evaluation and comparison

In this section, a detailed assessment of relevant tools is shown. We separately evaluate black-box, white-box, and gray-box techniques and tools.

5.1 Black Box Testing

P.	Input	Output / Result	Programming Language	Implemented Tool Support	Applied in Real Environment	Specific to Embedded Systems	Instrumentation: source ◐/ binary ◑	Requires Source Code	Selection ◐/ Prioritization ◑
[13]	Matlab, Simulink, Stateflow model	Classification tree	PROGRES	●	●	●	○	○	○
[20]	Some kind of model	Test cases, scenarios	–	●	●	○	○	○	○
[29]	Some kind of model	Test cases, scenarios	TTCN-3	●	●	○	○	●	◐
[15]	Requirements specification	Test suite	C	●	○	●	○	●	◐
[45]	UML use cases	Test suite	C++	○	●	●	○	○	○
[39]	State diagram	Test cases	–	●	○	○	○	○	◐
[52]	State chart	Test cases	C++	○	●	○	○	○	○
[9]	Specification or model	Test scenarios or a graph	C#, .NET	●	●	○	◑	●	●
[53]	Any kind of unit tests or implementation	Parameterized unit tests	Java, .NET	●	●	○	○	○	○
[47]	Model or source code	Test suite and inputs	Java, .NET	●	●	○	○	●	○
[28]	Control flow	Test suite	C, Prolog	○	●	○	○	○	○
[10]	Input domain	Test inputs	C++	○	○	○	○	○	○

Table 1: Assessment of black-box testing methods.

In Table 1 we present an overview of the overall classification and evaluation criteria. The first column presents the citation index of the method, and the next columns are as follows:

Input Input type, which is usually some kind of model, source code or specification given in a suitable form (chart or diagram).

Output / Result A brief output/result description.

Programming Language The used programming language for each method.

Implemented Tool Support Whether the supporting tool is implemented.

Applied in Real Environment Whether the method is applied in real environment.

Specific to Embedded Systems Is the method used for embedded systems testing.

Instrumentation The used instrumentation technique, source code or binary instrumentation.

Requires Source Code Is the source code required?

Selection / Prioritization Is test case selection or prioritization possible?

The most promising method in BBT is the MaTeLo testing suite for automatic software validation, although it is not common in embedded system usage.

5.2 White Box Testing

The evaluation criteria for white box methods are the following:

Input type Gives the input of the evaluated method.

Output / result Gives output and / or result of the evaluated method.

Programming language Denotes whether the evaluated method is specific for some programming languages, or it can be applied to any programming language.

Implemented / tool support Indicates whether the method is implemented fully or partially, or there are tools that support this method.

Applied in real environment Indicates whether the method is purely theoretical, or it has been applied and its applicability has been proven in real scenarios.

Specific to embedded systems Indicates whether the evaluated method is specific to embedded systems environment, or it is general and can be effectively used not only in embedded systems.

Use some coverage measure Indicates whether the method uses some kind of coverage values (e.g. code or functional coverage) as input.

Computes some coverage measure Indicates whether the method computes some kind of coverage values (e.g. code or functional coverage) as output.

Instrumentation technique If instrumentation is used in the method, this point gives the instrumentation technique (e.g. source code, binary, etc.)

Requires source code Indicates if the method requires the source code of the system under test, or works from some other test basis.

BB testing method(s) This point indicates the general black-box testing methods that are specialized in the evaluated solution.

Makes selection / prioritization Indicates usage of test case selection/ prioritization techniques and shows exactly what kind of technique is used.

Prioritization / selection based on Shows the base measure or data of the used test case prioritization/selection techniques (e.g. extent of code covered, time required for execution, etc.).

P.	Input	Output / Result	Programming Language	Implemented Tool Support	Applied in Real Environment	Specific to Embedded Systems	Instrumentation: source ◐/ binary ◑	Requires Source Code	BB testing method	Selection ◐/ Prioritization ◑
[14]	source code	percentage of observed statements	C, C++	○	●	○	◐	●	–	○
[46]	source code, implementation, interface	sets of closes	–	○	●	○	○	●	use cases, boundary values, abstract implementation	○
[40]	formal model	boundary coverage	B, Z, VDM, UML/ OCL	●	●	◐	○	○	–	◐
[31]	Embedded System	instrumented system	C	●	●	●	◑	○	–	○

Table 2: Assessment of white-box testing methods.

Also, in Table 2 concerning WBT techniques, for each one, the input type, outputs/results, programming language, implemented tool support, is the method supplied in real environment, or specific to embedded systems, can it implement the instrumentation technique, does it require the source code, is it possible to combine with the BBT testing technique, or can the selection/prioritization be implemented during testing.

We can conclude that the “Boundary coverage criteria for test generation from formal models” is the most promising method, but it does not perform instrumentation, nor does it require source code. It also can perform selection and prioritization, but is not used in BBT.

5.3 Gray Box Testing

In this section white-box aided black-box testing methods (specially, coverage aided random testing, test case prioritization and selection) are assessed.

In Table 3, we give a briefing of the methods for gray-box testing. For each method, a brief evaluation concerning main specifications is presented. It seems that the first method which achieves both model and code coverage has the best options.

P.	Input	Output / Result	Programming Language	Implemented Tool Support	Applied in Real Environment	Specific to Embedded Systems	Instrumentation: source ◐/ binary ◑	Requires Source Code	BB testing method	Selection ◐/ Prioritization ◑
[38]	Specific-ation, implemen-tation	Program tree, test scenarios, test inputs	C, C++, C#, .NET	●	●	○	◐	●	Model-based	●
[41]	UML activity diagram	Test scenarios	UML	○	●	○	○	○	Model-based	○
[27]	Source code	Interface graph, test drivers, test inputs	C, C++, Java	○	●	○	◑	●	Model or graph-based	○
[35]	Model	Test cases, test inputs	–	○	●	●	○	○	Model-based, random	◐
[50]	Test set	Selected test set	–	●	●	○	○	○	Random	◐

Table 3: Assessment of gray-box testing methods.

5.4 Tools

Table 4 gives a briefing of approaches and tools for automated model-based test generation with similar properties overview like in previous tables, but also with information of coverage usage and its computation, selection/prioritization possibilities and the methods they are based on. Which one of these tools are mostly efficient, of course depends on the needs of the user. For example, LTG is both used in embedded systems and has many other advantages. Another good example is the Spec-explorer tool, but it is not for embedded systems usage.

Table 5 shows only two existing tools for automated model-based test generation integrated with test execution. The same properties are presented for each one.

6 Conclusions

During the assembly of this survey, we made the following observations.

There are many black-box and white-box testing techniques exist that are not specific to but can potentially be used in embedded systems enviromnents. Although the combination of black-box and white-box testing methods is mentioned many times as a method that can result in better testing, in these papers different techniques are rarely combined. Mostly fragments and partial solutions, but not complex processes are presented. For example, even if test execution produces some additional data, there is no feedback into some previous step of the process. Overall, although there are many possibilities to be used in embedded systems testing, these are not utilized (or at least not reported).

Tool	Input	Output / Result	Programming Language	Applied in Real Environment	Specific to Embedded Systems	Uses (◐) / computes (◑) coverage	Instrumentation: source (◐) / binary (◑)	Requires source code	BB testing method	Selection (◐) / prioritization (◑)
MaTeLo	Model	Test cases, scenarios	TTCN-3, XML, custom	●	○	◑	○	○	Markov model	◐
mbt	GraphML	Test cases, scenarios	Java	●	○	◑	○	○	Model-based	◐
TorX	Behavioural and environmental model	Test cases, scenarios	Any	●	○	◑	○	○	Ioco alg.	◐
JUMBL	TML Model	Test cases, scenarios	Java	●	○	◑	○	○	Markov model	◐
TGV	Labelled Transition Model	Test cases, scenarios	TTCN	○	○	●	○	○	Ioco alg.	◐
AETG	Model	Test cases, scenarios	–	●	○	◑	○	○	–	◐
LTG	System usage model	–	–	●	●	●	○	○	–	◐
Conformiq	UML or QML model	Test cases, scenarios	Python, TCL, TTCN-3, C, C++, Visual Basic, Java, Junit, Perl, Shell Scripts	●	○	◑	○	○	–	◐
Spec Explorer	Specification or model	Test scenarios	C#, .NET	●	○	●	●	●	Markov model	●

Table 4: Overview of Existing Approaches and Tools for Automated Model-Based Test Generation.

In addition, despite of there are some promising tools, which could be effectively used to ease testing and/or improve its quality, neither of them are specialized for embedded systems. And there are only a very few papers report on the application of these testing techniques in embedded systems, and most of these papers report on results, and not on technical details.

Tool	Input	Output / Result	Programming Language	Applied in Real Environment	Specific to Embedded Systems	Uses (◐) / computes (◑) coverage	Instrumentation: source (◐) / binary (◑)	Requires source code	BB testing method	Selection (◐) / prioritization (◑)
JTorX	Behavioural and environmental model	Test cases, scenarios	Java	●	○	◑	○	○	Improved ioco alg., uioco alg.	◐
AGEDIS	UML Model	Test cases, scenarios	ATS	○	○	◑	○	○	Based on coverage of inputs to the model	○

Table 5: Overview of Existing Approaches and Tools for Automated Model-Based Test Generation.

Thus, it seems to be that a good general framework for embedded systems testing is still missing from the market.

Acknowledgement

This work were done in the Cross-border ICT Research Network (CIRENE) project (project number is `HUSRB1002/214/044`) supported by the **Hungary-Serbia IPA Cross-border Co-operation Programme**, co-financed by the European Union.

References

[1] Debugging, August 2012.

[2] D. Achlioptas, C. Borgs, J. Chayes, Robinson H., J. Tierney, and Microsoft Corporation. Methods and systems of testing software, and methods and systems of modeling user behavior, 2009.

[3] Luca De Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[4] alphaWorks. Focus homepage, `http://www.alphaworks.ibm.com/tech/focus`.

[5] A. Belihfante. Jtorx: A tool for on-line model-driven test derivation and execution. *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, 6015:266–270, 2010.

[6] E. Bernard, F. Bouquet, A. Charbonnier, B. Legeard, F. Peureux, M. Utting, and E. Torreborre. Model-based testing from uml models. *Lecture Notes in Informatics*, pages 223–230, 2006.

[7] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. A model-based regression test selection approach for embedded applications. *SIGSOFT Softw. Eng. Notes*, 34(4):1–9, July 2009.

[8] F. Bouquet, B. Legeard, F. Peureux, and E. Torreborre. Mastering test generation from smart card software formal models. In *Proceedings of the International Workshop on Construction and Analysis of Safe Secure and Interoperable Smart devices*, pages 70–85. Springer-LNCS, 2004.

[9] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Testing concurrent object-oriented systems with spec explorer. In *Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 542–547. Springer, 2005.

[10] T.Y. Chen. Adaptive random testing. In *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, page 443, August 2008.

[11] D.M. Cohen, S.R. Dalal, A. Kajla, and G.C. Patton. The automatic efficient test generator (aetg) system. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 303–309, November 1994.

[12] Conformiq Inc. Homepage, `http://www.conformiq.com/products/`, August 2012.

[13] Mirko Conrad, Heiko Dörr, Ingo Stürmer, and Andy Schürr. Graph transformations for model-based testing. In *Modellierung in der Praxis - Modellierung für die Praxis*, Modellierung 2002, pages 39–50. GI, 2002.

[14] José C. Costa, Srinivas Devadas, and José C. Monteiro. Observability analysis of embedded software for coverage-directed validation. In *In Proceedings of the International Conference on Computer Aided Design*, pages 27–32, 2000.

[15] S.J. Cunning and J.W. Rozenblit. Automatic test case generation from requirements specifications for real-time embedded systems. In *Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on*, volume 5, pages 784–789, 1999.

[16] Alan M. Davis. A comparison of techniques for the specification of external system behavior. *Commun. ACM*, 31(9):1098–1115, September 1988.

[17] R. G. de Vries, A. Belinfante, and J. Feenstra. Automated testing in practice: The highway tolling system. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, pages 219–234. Kluwer Academic Publishers, 2002.

[18] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using spin. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:382–393, 2000. 10.1007/s100090050044.

[19] L. Du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, and R. de Vries. Formal test automation: The conference protocol with tgv/torx. In *Proceedings of the 13th International Conference on Testing Communicating Systems (TestCom 2000)*, page 221–228, August 29 - September 1 2000.

[20] I. K. El-Far and J. A. Whittaker. Model-based software testing. In *Encyclopedia of Software Engineering*, pages 1–22. John Wiley & Sons, 2001.

[21] V. Encontre. Testing embedded system: Do you have the guts for it?, 2004.

[22] A. Feliachi and H. Le Guen. Generating transition probabilities for automatic model-based test generation. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, pages 99–102, April 2010.

[23] J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. An experiment in automatic generation of conformance test suites for protocols with verification technology. *Science of Computer Programming*, 29:123–146, 1997.

[24] P. Flores, V. Mehta, H. Nguyen, M. Sharma, C. Walsh, T. Xiong, and Sony Electronics. Automated test for consumer electronics, 2010.

[25] Formal Methods and Tools research group, University of Twente. JTorX - a tool for model-based testing, homepage, `http://fmt.cs.utwente.nl/tools/jtorx/`, August 2012.

[26] Formal Methods and Tools research group, University of Twente, Eindhoven Technical University, Philips Research Laboratories, and Lucent Technologies. TorX test tool homepage, `http://fmt.cs.utwente.nl/tools/torx`, August 2012.

[27] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[28] Arnaud Gotlieb and Matthieu Petit. Path-oriented random testing. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 28–35, New York, NY, USA, 2006. ACM.

[29] A Guiotto, B Acquaroli, and A Martelli. *MaTeLo: Automated Testing Suite for Software Validation*, pages 253–261. ESA, 2003.

[30] A. Hartman and K. Nagin. The agedis tools for model based testing. *Test generation - ACM SIGSOFT Software Engineering Notes archive*, 29(4):129–132, July 2004.

[31] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM.

[32] A. Huima. Implementing conformiq qtronic. In *Testing of Software and Communicating Systems (TestCom/FATES'07)*, volume 4581/2007, pages 1–12. Springer-LNCS, 2007.

[33] IBM Research Laboratory in Haifa, Oxford University Computing Laboratory, Verimag laboratory at Universite Joseph Fourier in Grenoble, France Telecom R&D, IBM development Laboratory in Hursley Park (UK), Intrasoft International, and imbus AG, Moehrendorf, Germany. Automated generation and execution of test suites for distributed component-based software, agedis homepage, `http://www.agedis.de/index.shtml`, August 2012.

[34] C. Jard and T. Je'ron. Tgv: Theory, principles and algorithms. In *Proceedings of the Sixth World Conference on Integrated Design and Process Technology, IDPT-2002*, June 2002.

[35] A. Agung Julius, Georgios E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust test generation and coverage for hybrid systems. In *Proceedings of the 10th international conference on Hybrid systems: computation and control*, HSCC'07, pages 329–342, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] S. Kandl, R. Kirner, and P. Puschner. Automated formal verification and testing of c programs for embedded systems. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*, pages 373 –381, may 2007.

[37] Kristian Karl and Johan Tejle. mbt homepage, `http://mbt.tigris.org/`, August 2012.

[38] Nicolas Kicillof, Wolfgang Grieskamp, Nikolai Tillmann, and Victor Braberman. Achieving both model and code coverage with automated gray-box testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, A-MOST '07, pages 1–11, New York, NY, USA, 2007. ACM.

[39] Nicha Kosindrdecha and Jirapun Daengdej. A test generation method based on state diagram. *Journal of Theoretical and Applied Information Technology*, 18(2):28–44, August 2010.

[40] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary coverage criteria for test generation from formal models. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 139–150, November 2004.

[41] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating test cases from uml activity diagram based on gray-box method. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 284–291, Washington, DC, USA, 2004. IEEE Computer Society.

[42] P. Liu and Hon Hai Precision Industry. Automated test measurement system and method therefor, 2008.

[43] Microsoft. Spec Explorer homepage, `http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745/`, August 2012.

[44] K.W. Miller, L.J. Morell, R.E. Noonan, S.K. Park, D.M. Nicol, B.W. Murrill, and M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, 18(1):33–43, 1992.

[45] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel. Automatic test generation: a use case driven approach. *Software Engineering, IEEE Transactions on*, 32(3):140–155, March 2006.

[46] Hanne Riis Nielson and Flemming Nielson. Flow logic: a multi-paradigmatic approach to static analysis. In Torben ÆMogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The essence of computation*, pages 223–244. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[47] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84, May 2007.

[48] A. Penttinen, R. Jastrzebski, R. Pollanen, and O. Pyrhonen. Run-time debugging and monitoring of fpga circuits using embedded microprocessor. In *Design and Diagnostics of Electronic Circuits and systems, IEEE*, pages 147–148, 2006.

[49] S. Prowell. Jumbl: A tool for model-based statistical testing. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, page 337.3, 2003.

[50] T. Pyhala and K. Heljanko. Specification coverage aided test selection. In *Application of Concurrency to System Design, 2003. Proceedings. Third International Conference on*, pages 187–195, June 2003.

[51] A. Rau and Philips Electronics. System and method for automated testing of digital television receivers, 2004.

[52] Valdivino Santiago, Ana Silvia Martins do Amaral, N. L. Vijaykumar, Maria de Fatima Mattiello-Francisco, Eliane Martins, and Odnei Cuesta Lopes. A practical approach for automated test case generation using statecharts. In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, volume 02 of *COMPSAC '06*, pages 183–188, Washington, DC, USA, 2006. IEEE Computer Society.

[53] N. Tillmann and W. Schulte. Unit tests reloaded: parameterized unit testing with symbolic execution. *Software, IEEE*, 23(4):38–47, July-August 2006.

[54] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949, pages 1–38. Springer-LNCS, 2008.

[55] J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In *Proceedings of the First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.

[56] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence, 1996.

[57] Verimag. TGV, test generation with verification technology, homepage, `http://www-verimag.imag.fr/tgv.html`, August 2012.

[58] T. Wei-Tek, Y. Lian, Z. Feng, and R. Paul. Rapid embedded system testing using verification patterns. *Software, IEEE*, 22(4):68–75, July-Aug 2005.

[59] M. Wu and Sony Electronics. Automated software testing environment, 2010.

[60] T. Yu. Testing embedded system applications, 2010.

[61] Tingting Yu, Ahyoung Sung, W. Srisa-an, and G. Rothermel. Using property-based oracles when testing embedded system applications. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 100 –109, march 2011.