Embedded system software and hardware analysis methodology

Abstract

In this report a HW/SW analysis methodology is described. Many techniques are collected (in addition to the android-based one we are going to use in the pilot project) that allows logging execution traces. A "methodology" that, based on some questions, will help us to select the possible or best technique amongst these is given. The methodology also includes the possible applications of the inferred data. Besides, it gives answer on how trace can be extracted from embedded systems, what are debug interfaces, instrumentations with COM port, LAN communication, etc.

This work were done in the Cross-border ICT Research Network (CIRENE) project (project number is HUSRB/1002/214/044) supported by the **Hungary-Serbia IPA Cross-border Co-operation Programme**, co-financed by the European Union.

The project is co-financed by the European Union

Table of contents

1 Introduction

This document mainly targets white-box testing where concrete analysis of the internal device software and hardware is done. As such, it is suited well for testing methodology of devices under development and developed devices with unknown reliability, where debugging of particular device part is needed. Since UNS has vast experience in hardware and design of embedded systems in consumer electrons and USZG has high competence in systematic software evaluation (in terms of its security, quality, reliability, etc.) the two together have the aim of merging the experience in order to develop an efficient approach and system for analyzing the software and hardware of an embedded device, that can be used to produce appropriate information to improve the quality of the testing of hardware and software of the device, for which the specification is given.

The document gives a methodology that helps the assessment of embedded systems SW/HW architecture and defines white-box testing techniques that, based on the assessment, can be used to evaluate and improve the quality of functional testing by determining untested part of the embedded architecture.

Prior to this document, a survey of embedded systems testing methods [1] was prepared and a general methodology for functional testing of embedded systems [2] was defined in a second document within the CIRENE project. Based on the second document, this third document helps choosing implementation parameters of the relevant parts of the methodology based on the hardware-software attributes of the target system the methodology is applied for.

2 Background

2.1 Embedded systems architectures

Embedded systems are designed to perform a specific task, rather than being general-purpose systems. Real-time embedded systems have to meet specific requirements, while some embedded systems have low or no requirements to meet. Many embedded systems are not standalone devices, but rather consist of small computerized parts. Firmware is the set of instructions written for embedded systems and it is most commonly stored in a read-only memory.

Embedded systems range from having no user interface to having very complex graphical user interfaces. Most common interfaces are simple electronic interfaces – switches, buttons, LEDs and LCDs with some menu commands. The most complex systems have graphical interfaces with menus with a lot of options.

Embedded processors can be divided into two main categories:

- Ordinary microprocessors which use separate circuits for memory and peripherals,
- Microcontrollers which contain many peripherals on chip, reducing power consumption.

Many embedded systems are implemented as Systems-on-Chip (SoC) in one of the two ways:

- Application Specific Integrated Circuits (ASIC) which are digital systems specifically designed to perform a single task, or
- Field Programmable Gate Arrays (FPGA) which are general digital systems which can be programmed to behave as a wanted digital system.

Embedded systems often reside in machines that are expected to run continuously for years without errors, and in some cases recover by themselves if an error occurs. Therefore the software is usually developed and tested more carefully than that for personal computers, and unreliable mechanical moving parts such as disk drives, switches or buttons are avoided. Debugging of the software for embedded systems can be performed at different levels: (1) interactive resident debugging, (2) external debugging, (3) in-circuit debugging, (4) in-circuit emulation and (5) complete emulation.

There are several common software architectures for embedded systems:

- Simple control loop – a software consisting of a loop which controls the system and executes indefinitely,
- Interrupt-controlled system – tasks performed by a system are triggered by different kinds of events generated e.g. from a timer,
- Cooperative multitasking – the programmer defines a series of tasks and each task gets its own environment to be executed in,
- Preemptive multitasking - a low-level piece of code switches between tasks or threads based on a timer; here the system is considered to contain an operating system kernel,

- Microkernels - operating system kernel allocates memory and switches the CPU to different threads of execution. User mode processes implement major functions such as file systems, network interfaces, etc.
- Monolithic kernels - a relatively large kernel with sophisticated capabilities is adapted to suit an embedded environment, very complex and often expensive.

The following is a list of some possible interfaces for communication with an embedded system:

- Serial communication – RS232, RS422, RS485, etc.
- Synchronous serial communication – I2C, SPI, SSC, etc.
- Universal serial bus (USB)
- Multimedia cards – SD cards, flash, etc.
- Networks – Ethernet
- Fieldbuses – CANBus, LINBus, etc.
- Timers – PLL,
- General purpose input/output (GPIO)
- Analog-to-digital and Digital-to-analog conversion,
- Debugging interfaces – JTAG, ISP, ICSP, etc.

2.2 Techniques for information sharing

One of the most important things concerning white-box testing of embedded systems is the mode of information retrieval. During white-box testing a large amount of data are generated that have to be processed. This usually cannot be done on the embedded system under test, thus have to be transferred to a 'test server' machine. As different embedded systems have different communication capabilities, different methods have to be used. The availability of such an information sharing method can also determine the usable testing techniques.

Considering embedded systems, the following ways are possible to transfer the generated execution data to the test server:

- Drive/file: the trace logger application writes the information to a file on the file system. The advantages of this technique are that the processing of the trace is separated from the execution, and the trace is saved and can be processed more times in the future. A disadvantage is the storage space requirements (traces, especially those with fine granularity, can be very large). With the separate execution and processing phase the coverage information are not available during execution, but this is a minor issue. There are more technical solutions of storing the file system exists:
    - The file system can be stored on an integrated storage device the trace logger application saves the trace on the internal storage device of the embedded system and different communication interfaces have to be used to access the information (e.g. share the device over Ethernet connection).
    - In the case the file system is stored on a removable memory card (e.g. SD card), the memory card itself can be used to transfer the traces to the 'test server'. The disadvantage of this technique is that the memory card has to be removed from the device and must be physically transferred to the test server.

    - A third option is when the files are stored on an mounted, remote file system. The trace logger application simply writes the trace on the mounted drive, thus it can be stored e.g. on the 'test server' immediately. This is usually done over some other interfaces (e.g. over a network connection)
- Network: the logger application sends the trace information via network on the fly. Advantage of this technique is that the information is immediately sent to the 'test server', it does not consume embedded system storage resources and it is processed almost real time.
    - TCP connection: the logger application connects the test server and sends the trace information through Wi-Fi/LAN network connection. Bandwidth is an advantage of LAN (required in case of fine granularity), while wireless connection increases mobility of the device and supports more flexible usage/tests.
    - FM transmitter: the trace logger application transmits information via FM wave.
- Communication ports: through some communication ports a p2p connection can be established between the system under test and the 'test server', and through this connections the embedded device can send the execution data. This direct connection usually has a lower overhead than a general network connection, and it is more often can be used as a two-way communication channel.
    - USB
    - Serial port (DB-25 and DB-9 port).
    - Bluetooth: the logger application as a new slave device sends trace information to the 'test server'.
    - Infrared port (IrDA): the logger application as an IrDA client send trace information to the server.

2.3 Trace generation methods

Tracing information can be generated in various ways. Some of them are generally applicable, but some of them require specific hardware support.

2.3.1 Integrated Circuit level (debug port in embedded systems)

In general, debuggers provide very detailed information on the program execution. This information is more than enough for tracing purposes. The task in this case is to collect runtime information from the embedded system through some debug interfaces, and then filter out the irrelevant and extract the necessary information.

Many embedded systems (at least those version of the commercial ones that are dedicated for development) has hardware debugging interfaces. These interfaces (as their name indicates) are designed for debugging, but as mentioned above, can be used to retrieve the necessary data on program execution. From the test server we can access the executed instructions; we can query different variables (e.g. those storing the result of a decision). We also have the possibility to create a more sophisticated testing system/environment that not only examines but also dynamically modifies the execution. (E.g. modifies the value of some variables to reach the other branch of the code to achieve higher coverage.)

There are many debugging interfaces exist in different embedded systems.

JTAG is widely used for IC debug ports. In the embedded processor market, essentially all modern processors support JTAG debuggers. Embedded systems development relies on debuggers talking to chips with JTAG to perform operations like single stepping and break pointing. Advantages of JTAG are that we can obtain information about any program that can be executed on such a device..

The JTAG interface, collectively known as a Test Access Port (TAP), uses the following signals to support the operation of boundary scan.

- TCK – the **test clock** synchronizes the internal state machine operations
- TMS – the **test mode state** is sampled at the rising edge of TCK to determine the next state.
- TDI – the **test data in** represents the data shifted into the device's test or programming logic.
- TDO – the **test data out** represents the data shifted out of the device's test or programming logic and is valid on the falling edge of TCK when the internal state machine is in the correct state.
- TRST – the **test reset** is an optional pin which, when available, can reset the TAP controller's state machine.

The JTAG cable might connect to a PC's

- Parallel port
- USB port
- Ethernet port

The simplest is a parallel port JTAG cable. USB and Ethernet JTAG cables are also avaible. They are faster at streaming large amount of data, but more complex to control and have more overhead (slower) in case of small amount of data.

An example for utilize JTAG is the ARM11 processor family, the debug TAP of an the ARM1136core. The processor itself has extensive JTAG capability, similar to what is found in other CPU cores, and it is integrated into chips with even more extensive capabilities accessed through JTAG. *Serial Wire Debug* (SWD) technology is available as part of the CoreSight™ Debug Access Port and provides a 2-pin debug port, the low pin count and high-performance alternative to JTAG.

SWD replaces the 5-pin JTAG port with a clock + single bi-directional data pin, providing all the normal JTAG debug and test functionality plus real-time access to system memory without halting the processor or requiring any target resident code. SWD uses an ARM standard bi-directional wire protocol, defined in the ARM Debug Interface v5, to pass data to and from the debugger and the target system in a highly efficient and standard way.

SWD provides an easy and risk free migration from JTAG as the two signals, SWDIO and SWCLK, are overlaid on the TMS and TCK pins, allowing for bi-modal devices that provide the other JTAG signals. These extra JTAG pins are available for other uses when in SWD mode.

SWD is compatible with all ARM processors and any processor using JTAG for debug and provides access to debug registers in Cortex™ processors (ARM) and the CoreSight debug infrastructure.

Embedded Trace Module is a scan chain in which 40 bits (7 bit address, one 32-bit long data word, and a R/W bit) are used to control the operation of a passive instruction and data trace mechanism. This feeds either an on-chip Embedded Trace Buffer, or an external high speed trace data collection pod. The collected data share on PC or database. If trace data share database then we can use to evaluation at runtime.

2.3.2 Virtual machine level

Virtual machine is a software layer between the executable binary code and the operating systems. (It is a little bit lazy definition and allows more than a simple hardware virtualization layer.) It is an environment in which special binary code can be executed. Special binary is an intermediate language which is typically compiled from 'normal' source code. The virtual machine runtime executes the special binary files, emulating the virtual machine instruction set by interpreting it. Hardware emulators and simulators can also be treated as virtual machines. These virtual machines can be modified to provide necessary information about the code execution. E.g. we can use traces which consist of information of the executed instructions. An advantage of them over using debugging interfaces is that generation of unnecessary data can be omitted, while through debugging interfaces many data are generated that have to be filtered later. We can get the VM to generate only the necessary information, saving communication bandwidth, memory and storage usage.

An example for such virtual machines is Java. The Java uses Java Virtual Machine (JVM) which can be modified and used for tracing purposes.

The following example shows the profiling in Dalvik virtual machine. Dalvik is a process virtual machine in Android operating system. It is the software that runs the applications on Android devices. Android ensure a debugging tool called the Dalvik Debug Monitor Server (DDMS), which provides port-forwarding services, screen capture on the device, thread and heap information on the device, logcat, process, and more. DDMS provides method profiling. It means the tracking of certain metrics about a method, such as number of calls, execution time, and time spent executing the method.

There are some restrictions concerning the method profiling in DDMS:

- Android 1.5 devices are not supported.
- Android 2.1 and earlier devices must have an SD card present and the application must have permission to write to the SD card.
- Android 2.2 and later devices do not need an SD card. The trace log files are streamed directly to the development machine.

2.3.3 Middleware level

The middleware level is a software layer that lies between the operating system and hardware layer (e.g. Drivers). Through this layer we usually cannot acquire adequate information on the program executions.

2.3.4 Application level (instrumentation)

In application level we can use some kind of instrumentation for collecting trace information. As it was mentioned earlier, executable code is required as WB is a dynamic technique, thus the code gets executed during testing of the program to measure coverage. It can be given in compiled, binary form or in source code.

Java bytecode instrumentation is a widely-used technique and many libraries exist to it (e.g. ASM, Javassist, BCEL).

The following example shows the options of instrumentation on Dalvik virtual machine. A tool (dx) is used to convert some .class files (Java bytecode) into the .dex format (dalvik bytecode). Implementing a profiler to trace a program execution is non-trivial. One way to do this on running Java programs is through bytecode instrumentation. Bytecode instrumentation is a process where new functionality is added to a program by modifying the bytecode of a set of classes before they are loaded by the virtual machine. We can instrument the application bytecode before we convert it into dalvik bytecode. The Android system provides many communicational channels for the transfer of the data (e.g. WiFi, Bluetooth ...). All communicational channels must have permission in the application and we need to other bytecode because of their differences.

Similar but a little bit different are the Just In Time compilers, that also handles some binary intermediate code, but translates it into machine code instead of interpreting it. These JIT compilers can also be modified to instrument the binaries on the fly.

3 Methodology

In this section, we collect different solutions from different perspectives. The goal of this section is to help to select the appropriate parts of the general methodology for particular environments. We defined four dimensions (see the next sections) according to which the feasible modules of the methodology can be selected. When the different modules are to be selected, first determine the most important dimension, and jump to the corresponding subsection.

Our dimensions are represented by questions:

At first, we want to answer the question: “What do we want to do?” Here we collect different tasks that have to be done in order to perform successful embedded systems testing tasks. For each task, we list the possible solutions, and we describe each solution. Description includes the preconditions, hardware / software requirements, applicability, advantages and disadvantages of the given solution.

Second, the question “What solutions do we have?” becomes the most important one. Here all the solutions are listed. The descriptions of the solutions include preconditions and hardware / software requirements of the solution. Then we list a set of problems the corresponding solution can be applied to, and give the advantages/disadvantages of using this solution for the given problem.

Third, the main point of view is the hardware/software configuration. The listing question is: “What attributes the hardware / software system has?” For each attribute we list those solutions that require the given attribute, and list the additional requirements of the corresponding solutions.

Fourth, the question “What information do we have?” refers to already existing information. They are listed and the solutions that requires the information, and give the advantages/disadvantages of using the information for the given problem.

These lists can be used to make decisions when a new embedded system testing toolchain is to be created. Based on the available hardware / software properties of the system, one can exclude technical solutions that are not applicable to the given environment, or based on the proposed testing methodology one can raise requirements against the system.

3.1 What do we want to do?

3.1.1 Generate coverage information

Coverage information is necessary to perform any coverage-based testing tasks like test selection, test prioritization, test case generation, etc. Coverage information is usually based on a “trace” of the program. This trace may contain various information about the actual program execution, like function-entries or returns, line number of executed instructions, etc. To manage these information we need to store them in local files, or transfer the data to a remote computer that proceeds the further computations.

To produce this trace, we have the following solutions:

1. **Source code instrumentation**: The source code instrumentation is a language specific procedure. We can use this technique if the source code and source code analyzer are available. The source code analyzer is required because it creates the abstract syntax graph (ASG) which can be modified, and from this modified ASG the new instrumented source code can be generated.

   Advantages: the modifications are "readable" by the programmers, the instrumented code can be further modified if necessary (e.g. for debugging), the transformation of the program is more clear, and the instrumentation is a separate, removable step in the build-chain.

   Disadvantages: source code is necessary, the compiling time grows up, and the build toolchain/environment has to be slightly modified.

2. **Static binary code instrumentation during build**: This can be a language or architecture specific procedure. We need to be able to modify the compiler program (or use the compiler's built-in functionality, e.g. gcc –instrument-function) in a way that it generates extra instructions into the binary code. It can be very similar to source code instrumentation if the ASG of the program is modified, and the extended ASG is used for binary code generation. But we can also use the internal representation of the binary code as the base of instrumentation, which is more comparable to an "after built instrumentation". This technique requires source code and a compiler with instrumented functionality.

   Advantages: it requires no, or only a few fixed modifications in the build-chain; mostly automated process; instrumentation in the same step than the build.

   Disadvantages: source code is necessary and that the modifications in the binary code are "hidden" from the programmers, thus it cannot be easily checked, and the code can be treated less reliable because of the modifications in the compiler.

3. **Static binary code instrumentation after build**: This is an architecture specific procedure. The compiled binaries are processed, instrumented, and the new binaries are executed. This solution requires tools that can read, process, and reliably modify the binary code.

   Advantages: it does not require source code, and works on any executables or binaries; can be done in any time; easily automatable.

   Disadvantages: reading and modifying a binary code is not an easy task, and (as a consequence) the modified binaries are treated even more unreliably than those produced by the build time instrumentation.

4. **Dynamic binary code instrumentation**: This is also a language specific procedure. This technique is available in two levels: execution framework and OS. This makes available the modification of the binary in runtime.
   In execution framework level this technique needs an appropriate instrumentation tool or library and the expandable framework (plug-in, agent, etc.).
   The OS level's instrumentation requirement is an expandable OS (OS source, library, etc.).

Advantages: no need for source code, we may use this tool for third-party programs; add and remove instrumentation code in runtime; flexible modifiability; review of the code during runtime.

Disadvantages: log filtering is difficult because all programs generate lots of information; runtime may slow down a bit; instrumented versions not preserved; the same binaries are instrumented over and over again.

5. **Hardware debug information**: It collects information via debug port on hardware. This technique needs debug port on the embedded system and external debug tool or device.

   Advantages: we can obtain information without modifying the source code; we can step-by-step execute and pause program running with external tool; very fast because the hardware support; we can supervise the code during run; no code modification required; very low level information can be extracted.

   Disadvantages: a debug port is necessary on the device; compatible debug hardware needed; need to have a support in the device's processor.

6. **Software debug information:** This technic's requirements are appropriate emulator or simulator and binary with debug information. Usually the developer tools have debugger part.

   Advantages: hardware device is not required; we can supervise the code and the binary during execution; very low level information can be extracted.

   Disadvantages: slow, not real time; needs an intermediate layer; the emulator/simulator may not emulate/simulate correctly the device.

3.1.2 Test executor and controller

1. **External control:** This technique's requirements are suitable connection points and programs on the device. This technique needs controller hardware or software.

   Advantages: we can directly control the embedded system; it is well automatable.

   Disadvantages: it needs specific devices and connectors.

3.2 What solutions do we have?

1. **Appropriate binary instrumentation tool**: If a binary instrumentation tool is available, we can use static or dynamic binary code instrumentation. For this we need to have the binary code of the program. Using instrumentation, trace generation can be solved and coverage information can be computed.

   Advantages: we don't need the source code of the program appearing on the device.

   Disadvantages: the binary is modified.

2. **Appropriate source code instrumentation tool**: We can use this tool to source code instrumentation if the source code is available. Using instrumentation, trace generation can be solved and coverage information can be computed.

   Advantages: can collect many kind of information, loading time doesn't change subsequently, manually modifiable source code, clearer transformation of the program, opportunity of separated instrumentation.

   Disadvantages: compilation time grows.

3. **Compiler is able to perform instrumentation**: if we use compiler which can place debug information into the application, we can make static binary code instrumentation. Trace and coverage information can be generated.

   Advantages: we don't need to care about the proper place of the instrumentation, the compiler makes it automatically.

   Disadvantages: we can't fine-tune the information-gathering.

4. **Source code**: If the source code is available, we can perform source code instrumentation by appropriate code instrumentation tool.

   Advantages: can collect many kind of information in many levels.

5. **Appropriate emulator/simulator source code**: If the emulator's or simulator's source code is available, we can modify it to collect information with these. Also we can run an instrumented program on these and study the behavior of the code and the program. Trace generation is feasible using this technique.

   Advantages: don't need a real device; any kind of damage is not going to happen.

   Disadvantages: maybe the simulator or the emulator is not perfectly simulates/emulates the device.

3.3 What attributes the HW/SW configuration has?

3.3.1 Hardware attributes

1. **Debug port**: if the debug port is available, we can use this for hardware debugging. In additional, we need an external device that connects to the debug port and communicates with the processor of the embedded system. Trace and coverage information can be gathered through this debug interface.

   Advantages: we can collect information directly from the device, even in the basic levels.

2. **Memory card socket**: one way to save and transfer the trace information to the 'test server' is to store it on a device. Removable memory cards can be used for this purpose.

Advantages: trace information is stored, and later can be processed many times.

Disadvantages: no online data processing is available, requires physical access to the device; limited storage capacity.

3. **Mounted remote file system:** one way to save and transfer the trace information to the 'test server' is to store it on a device. A mounted file system can be used to store this data. This requires some network connections, and an external storage server, which can be the test server itself, or another dedicated file server.

   Advantages: no physical access to the device is required to access the trace;. trace information is stored, and later can be processed many times; storage capacity of the device under test is not a limit.

   Disadvantages: no online data processing is available.

4. **Integrated network connection port**: this device helps us to connect to the test server and send information through the network. This also enables mounted file systems.

   Advantages: network connection enables online trace processing; storage capacity of the device is not a limit.

   Disadvantages: this technique needs continuous network connection and a remote computer that receives the data.

5. **Wi-Fi port**: if this device is available, we can use wireless network connections. In addition to the properties of a network connection, the wireless property provides more freedom in assembling and executing some special tests (which requires e.g. mobility).

   Advantages: network connection enables online trace processing; storage capacity of the device is not a limit; device is not tied to a physical location.

   Disadvantages: signal quality may vary.

6. **USB port**: this device helps us to connect to the computer via USB protocol. We can make direct connection between device and computer, and we can use it to transmit trace information or to debug the device. Or it may enable the connection of some external storage.

   Advantages: universal; easy data transfer; easy communication.

   Disadvantages: necessary drivers and software both on the computer and the device.

7. **Serial port**: we can make direct connection with serial port. This is a common connection interface, but since the USB is appeared, the serial port is obsolete.

   Advantages: easy legacy protocol.

Disadvantages: this technique is slow.

8. **Infrared port**: if the infrared device is available, we can connect to the computer that also have infrared connection device and we can send data to it.

   Advantages: "wireless" connection.

   Disadvantages: technique needs direct line of sight to receiving device; doesn't suitable for long distances; unreliable.

9. **Bluetooth:** same as the infrared connection, but using radio waves instead of light.

   Advantages: technique's energy demand is small; good reliability.

   Disadvantages: short-distance and small bandwidth.

3.3.2 Software attributes

1. **OS level**: if the system has OS level, we can use dynamic binary instrumentation by modifying OS level for information collection. The OS level facilitate that we can use own simulator tool. In addition the OS level supports many communication interfaces (e.g. socket, USB, file system …).

   Advantages: no need for source code of software under test; works for libraries and third-party codes.

   Disadvantages: Modifies the executed code; requires OS source code.

2. **Execution framework level**: in this level we can use dynamic binary code instrumentation from this framework or we can modify the framework to output trace information.

   Advantages: no need for source code of software under test; works for libraries and third-party codes.

   Disadvantages: Modifies the executed code or the framework.

3.4 What information do we have?

3.4.1 Traceability

Traceability can be computed based on the connection between the functionalities, the test cases and the coverage information. We make the test cases from the specification, separately for each functionality. Thus we know which test case set refers to which functionality. During the execution we measure the coverage of the test cases, so we know how many and what lines, methods, etc. are reached by each test case. With this chain we can compute traceability links between test cases and source code, and through these and the existing traceability links, we can define missing links between other levels (e.g. requirements and source code). Or, if traceability exists between code and higher level elements, we can compute different coverage based on code coverage (e.g. compute requirements coverage if requirements-code traceability links are available).

Advantages: can be used for calculating different coverage based on code coverage and for directing random testing to find uncovered paths and braches.

Disadvantages: functionality, test cases, and code coverage information must exist (implicitly, source code and requirements must exist, as well).

3.4.2 Test cases

Base of simple test cases can be used to find uncovered paths and branches.

Advantages: code coverage level can be increased by adding simple test case to the test scenario.

Disadvantages: Large base of simple test cases must exist, and it is different for different devices.

3.4.3 Usage models

If we have a model of the device, we can make a usage model of it, by calculating probabilities based on device usage, i.e. while using the device, user entries can be traced along with the simulation of device software execution. When we have a usage model, we can make a test campaign where all paths are covered according to their probability level.

Advantages: it can reduce testing duration.

Disadvantages: Some paths can remain uncovered.

3.4.4 Specification

According to specification we can make the test cases separately, for each functionality.

Advantages: we know which test case set refers to which functionality

4 Summary

Embedded systems architectures with a focus on hw/sw layers and communication interfaces are analyzed.

As one of the most important things concerning white-box testing of embedded systems, techniques for information sharing are listed and evaluated: drive/file, network, and certain communication ports. Tracing information can be generated in various ways: integrated Circuit level, virtual machine level, and application level (instrumentation). Regarding methodology, by using test executor and controller, it is possible to generate coverage information according to: source code instrumentation, static binary code instrumentation during build, static binary code instrumentation after build, dynamic binary code instrumentation, hardware debug information, and software debug information. Possible solutions for generating coverage information are evaluated, while some already existing information like traceability, test cases, usage models and specifications can be reused for calculating coverage information.

5 References

[1] Árpád Beszédes, Tamás Gergely, Kornél Muhi, Róbert Rácz, Csaba Nagy, István Siket, Gergő Balogh, Péter Varga, Miroslav Popovic, István Papp, Jelena Kovacevic, Vladimir Marinkovic. Survey on Testing Embedded Systems. Technical report. 2012.

[2] Embedded systems functional testing methodology. Cirene project technical report. 2012