A Script-Based Testbed for Mobile Software Frameworks

Walter Hargassner, Thomas Hofer,
Claus Klammer, Josef Pichler
Software Competence Center Hagenberg
forename.surname@scch.at

Gernot Reisinger
COMNEON electronic technology GmbH & Co. OHG
gernot.reisinger@comneon.com

Abstract

Software testing is essential and takes a large part of resources during software development. This motivates automating software testing as far as possible. Frameworks for automating unit testing are approved and applied for a plethora of programming languages to write tests for small units in the same programming language. Both constraints, unit size and programming language, inhibit automation of software testing in domain of mobile software frameworks. This circumstance has motivated the development of a new testbed for a framework in the domain of mobile systems. In this paper, we describe requirements and challenges in testing mobile software frameworks in general and present a novel testbed for the APOXI framework that addresses these requirements. The main ideas behind this testbed are the usage of a scripting language to specify test cases and to incorporate domain-specific aspects on the language level. The testbed facilitates component and system testing but can be used for unit testing as well.

1. Introduction

Comneon (www.comneon.com) is a leading supplier of software for mobile systems. As its top product it markets and supports the APOXI (**A**pplication **P**rogramming **O**bject-oriented e**X**tendable **I**nterface, www.apoxi.com) framework. APOXI is an object-oriented application framework for rapid development of man machine interfaces (MMIs) and integration of various applications for mobile communication products, i.e. mobile handsets. APOXI has been designed to meet the special requirements of mobile, embedded devices and provides specific, easy to use and comprehensible APIs for MMI and application development. The main components of APOXI are graphical user interface services, data control, stack control and an application control mechanism. Aside from being an abstraction to the radio protocol stack, it also provides APIs for hardware drivers of SIM card, flash memory, display, audio device, video, and the underlying real time operating systems (see Figure 1).

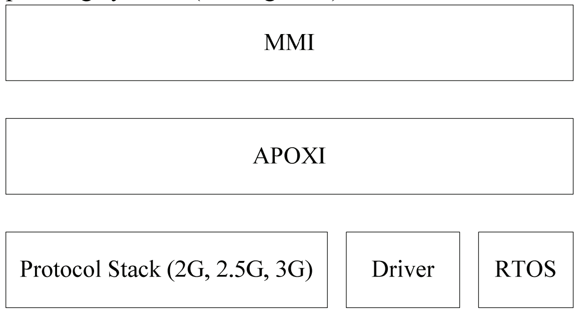


Figure 1. APOXI architecture

The development of software solutions with APOXI is a multi-stage process, involving several different stakeholders as follows [1]:

- The APOXI framework providing the base functionality is developed by the Comneon framework team, which develops, maintains, and enhances the main APIs and services.
- Based on the APOXI framework, a reference implementation of an MMI (called MMI Template) is developed by the Comneon MMI team.
- Mobile phone manufacturers (MPMs) are the customers of Comneon and develop customized software solutions for their products either by adapting and extending the MMI Template or starting from scratch on top of APOXI.

This multi-stage development process exhibits a significant problem concerning system testing of APOXI. The MMI Template available for APOXI developers does not provide an exhaustive usage of APIs and services provided by the framework. Hence, using the MMI Template as test driver for system tests of APOXI is not sufficient to achieve full coverage of all APIs and services. Due to the framework nature of APOXI, even a full-featured MMI developed by an MPM does not necessarily cover the entire API.

The dependency of APOXI from underlying components like the protocol stack constitutes another kind of problem because system tests include communication between APOXI and protocol stack, too. For instance, testing an API function concerning functionality provided by the protocol stack requires availability of network signal operated by the protocol stack. However, hardware testbeds providing simulation of network signals are expensive and not available to all testers and developers.

In response to these problems, Comneon launched the Testbed project aiming the development of a test environment for system testing of the APOXI framework in isolation decoupled from both the MMI and the underlying protocol stack. The test environment shall support system testing done by testing experts as well as functional tests performed by developers for single modules.

A further motivation for the Testbed project was to improve the localization of errors and to reproduce error situations that appear at MPM developer sites in a local APOXI development site.

The remainder of this paper is structured as follows: Section 2 will further discuss different test configurations, as well as the APOXI framework to derive requirements and constraints on a test environment. Based on the identified requirements, we will present the main design decisions for the Testbed project in section 3. In section 4 we will outline the overall architecture of the created testbed and present some components in more detail. This is followed by an overview of provided tools in section 5. Section 6 will compare the presented concepts and tools with related systems and demonstrates its unique features. Finally, section 7 will conclude with a summary.

2. Challenges and Requirements

In this section, we briefly outline the challenges, requirements, and constraints that have driven the design of our testbed and which resulted in the design decisions presented in the next section. The identified requirements mainly stem from the APOXI framework but hold in general for module/system testing of mobile software frameworks.

Figure 2 shows an APOXI-based system during host development. This system differs from the architecture in Figure 1 in three ways. First, the real protocol stack is replaced by a host implementation that simulates the stack on the host partially. Second, hardware drivers are replaced by stubs that implement driver functionality based on the host hardware (e.g. keyboard) or provide a simple stub for development and testing purpose only (e.g. battery) if no reasonable implementation is possible on the host. Third, instead of the RTOS, the host operation system is used.

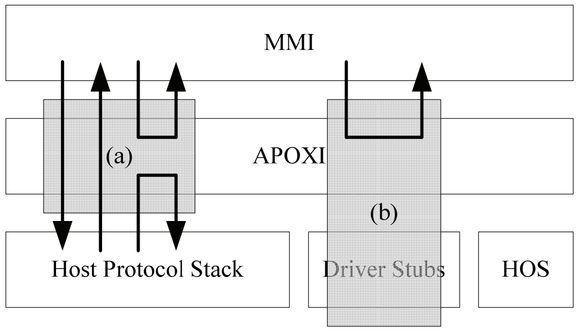


Figure 2. System under test on host

In its simplest form—depicted in Figure 2a—APOXI is the system under test (SUT) whereas both the MMI and the protocol stack and drivers are not part thereof. Based on this architecture, we can identify four basic test cases indicated by arrows in Figure 2 :

1. Receive an event from the MMI and forward an event to the protocol stack or a driver.
2. Receive an event from the protocol stack or from a driver and forward an event to the MMI.
3. Receive an event from the MMI and send an event back to the MMI.
4. Receive an event from the protocol stack and send an event back to the protocol stack.

For all these test cases, different mappings between receiving and responding events are possible so that one event may trigger several events and several events may trigger a single event only. Basic test cases as described above are either tested alone or composed to more complex test cases. Complex test cases also motivate the combination of APOXI, the protocol stack, and drivers to another configuration of the SUT, as depicted in Figure 2b. Test case 3 and the combination of test cases 1 and 2 are reasonable for this SUT configuration too. Test cases specify functionality that must be provided to the MMI without specification of communication between APOXI and the underlying protocol stack and driver stubs. However, as the protocol stack is simulated on host only, most functionality concerning the protocol stack cannot be tested in this way (but by test cases 1 and 2).

On a target platform, both configurations of the SUT as well as all identified test cases are reasonable too, however with real protocol stack and real driver implementations.

The presented system configurations serve as the basis for the description of requirements for test environments in general and for the APOXI testbed in particular.

1. *Consistent testing on host and target*. Testing on the host covers testing of single modules as well as integration tests of all modules. It makes sense to reuse both kinds of test cases on the target without modification. Reusing host test cases on the target assures same behavior on host and target and makes it easy to find specific problems on the target that does not appear on the host. Apparently, testing on the target will cover more test cases that are not applicable on the host due to simulation of protocol stack and hardware drivers.

2. *Consistent testing with different configurations of SUT*. Test cases apply for different configurations of the SUT, namely including or excluding protocol stack and hardware drivers. Testing different SUT configurations with exactly the same test scripts requires a runtime that executes statements (e.g. to simulate an event from protocol stack to SUT) against a distinct SUT configuration.

3. *Replacement of protocol stacks and driver implementations*. The replacement of protocol stacks and hardware drivers is necessary for testing APOXI alone. The replacement of protocol stack and hardware drivers includes receiving and sending events by the corresponding communication mechanism as well as the connection to the test environment to control and verify simulated events.

4. *Replacement of MMI to drive test cases*. The main focus on testing APOXI or frameworks in general is to test functionality provided for the above layer, i.e. the MMI. For automatic testing, the MMI must be replaced by a testing component that is functionally equivalent to the MMI concerning the communication with lower layer. Furthermore, the testing component must provide hooks to drive individual test cases and to verify test case results as well.

5. *Support to reproduce real communication in test scenarios*. Developers do not have a full-fledged hardware equipment for testing. A testbed should facilitate the reproduction of communication in hardware test environment in typical developer environment to reproduce errors. This includes the capability to record communication between APOXI and protocol stack without side effects. The same scenario holds for communication between MMI and APOXI due to distributed development sites. When the MMI developer discovers an error in APOXI, the APOXI developer should be able to reproduce this kind of error without having an MMI available and, after fixing the problem, to verify the correct behavior.

6. *Support for different types of communication*. Different communication mechanisms are used by protocol stacks and hardware drivers. For example, APOXI is aware of AT command interface [2] and SDL [3]. A testbed for APOXI, therefore, has to intercept the mechanisms implemented in APOXI both to intercept and transmit events. Furthermore, hardware drivers may provide a simple C programming interface for communication only. Communication between middleware (APOXI) and MMI depends on the middleware itself. For instance, APOXI provides a rich object-oriented application programming interface (API) to implement an MMI on it.

7. *Extensibility*. A framework like APOXI must be extensible in view of new protocol stacks and, hence, in view of new communication mechanisms. This extensibility is required for a testbed too. When APOXI is changed in respect of the new communication mechanism, existing test cases that rely on the old mechanism shall be easily adapted to the new mechanism.

8. *Automatic and continuous test case execution*. The development of frameworks like APOXI follows an agile approach with short release-cycles and distributed development teams. With this agility, continuous and automatic execution of test cases becomes a crucial factor to achieve and maintain required software quality. As required for automatic execution in general [4], a failed test case shall not harm other test cases so test cases must be self-contained and executed independent of each other. A further requirement for continuous testing is to prevent rebuild and deployment of test cases together with the SUT when new test cases are added or existing test cases are changed or extended. Especially for target testing, single deployment must be sufficient even new test cases are added or existing test cases are changed.

9. *Integration with test management system*. Component and integration testing are only one

approach for testing that completes traditional unit tests (or programmer tests) and full functional system tests performed manually. A testbed for module/system testing must seamlessly be integrated with an existing test management system.

Most of the presented requirements are typically for testing frameworks for mobile software systems in general.

3. Principal Design Decisions

Based on a detailed analysis of requirements and technical considerations as outlined above, the main design decisions are described in the next sections.

3.1 Usage of Scripting Language

Implementing test cases in the same programming language used to implement the SUT seems obvious at first glance. Frameworks for unit testing following this approach exist for a plethora of programming languages including the C++ programming language that is widely used to implement frameworks like APOXI. For example, CppUnit [5] is an open-source framework for unit testing in C++ and C++test [6] from Parasoft is a commercial product including capability for C++ unit testing even for embedded systems.

Using the same language will promote reuse programming interfaces to intercept communication for sending and receiving events. For example, APOXI provides an API for sending and receiving SDL signals that could directly be reused for testing purpose too. The downside of this tight integration of SUT and test case implementations is that after adding or changing a test case, the test cases must be recompiled and linked together with the SUT and, for testing on the target, redeployed on the target device. This circumstance contradicts requirements elaborated in the previous section. To resolve this shortcoming, we introduced a scripting language to implement test cases and a runtime environment that interprets and executes these test scripts in context of a SUT.

In general, test cases implemented in scripting language have many advantages compared with a compiled language like C++.

- In contrast to unit tests, module/systems test cases are not only written by developers but also by testers. Testers usually are not familiar with source code of the SUT even are not usually C++ experts. This makes it difficult for testers to implement test cases in C++ against the API of the SUT.
- Languages like C++ cause many programming errors, most notable errors concerning memory management. An uninitialized pointer variable within a test case implementation may crash the entire system. Using a scripting language prevents many programming errors and facilitates more robust test cases that cannot the SUT.
- Using a scripting language stringently leads to decoupling of the SUT and test cases. Instead of using the API to send and receive events directly, high-level language constructs can be used instead that are more stable with regard of changes of the SUT.
- A scripting language also facilitates high-level constructs for testing tasks like sending and receiving events. For instance, sending an SDL signal by means of the APOXI API typically requires 10 lines of source code at a minimum whereas the same signal can be sent by a single instruction in the scripting language.
- Using a runtime that interprets test scripts allows using different scripting languages. This makes it easier to integrate the test environment with other existing environments.

The usage of a scripting language, however, results in some additional challenges compared to usage of the same language. Most notable is the need for a script execution environment, as described in the next section.

3.2 Script Execution Environment

Execution of test scripts requires an interpreter that is tight integrated with the SUT. A number of scripting engines exists for the C/C++ programming languages. For instance, SpiderMonkey [7] is a C implementation for the JavaScript language. Even as SpiderMonkey was successfully integrated with the APOXI framework, we decided to compile test scripts in a compact code format that is interpreted by a special interpreter integrated with the SUT. The following considerations have motivated this decision.

First, interpretation of a script including lexical, syntax, and semantic analysis at runtime results in poor execution time of test scripts as well as late detection of lexical and syntactical errors. As consequence, parsing and analysis of test scripts must be performed prior to deployment and execution of test scripts.

Second, reusing a runtime environment for an existing scripting language binds us to one language. Separation of the execution runtime from any scripting language facilitates changing the scripting language used for writing test scripts later on.

Third, for execution on the target device, it should be possible to store all test scripts on the device so that script execution is possible on self-contained devices without further communication with a host environment. Hence, scripts shall be as small as possible. Furthermore, access to test scripts located on flash file system is typically time intensive. So the size of test scripts has an influence on the loading time of test scripts too.

Fourth, tracing support can be integrated more easily with an interpreter built from scratch.

Another factor that encouraged us to implement our own interpreter is that an execution runtime for test scripts is rather small compared to a full scripting engine and the integration of the engine—new or reused—with the SUT is the most expensive part.

3.3. Script Test Environment

In context of the Testbed project, tool support was desired and necessary for several distinct but closely related activities concerning testing: writing test cases, verifying test cases prior to execution, and verify test cases against different versions of the SUT.

4. APOXI Testbed

Figure 3 depicts the main architecture of our testbed for APOXI including APOXI as SUT as well as protocol stack, hardware drivers, and operating system. The testbed serves as umbrella around the SUT. Testbed decouples it from underlying components like protocol stack and replaces the omitted MMI.

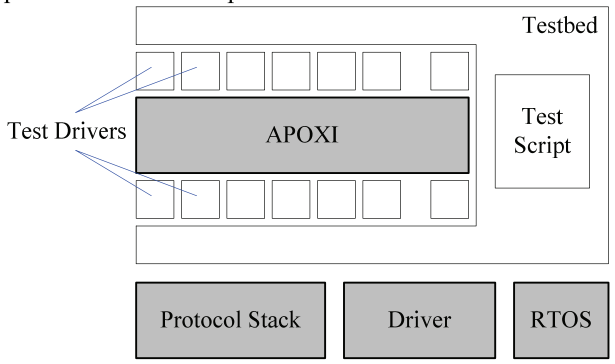


Figure 3. Testbed overview

The testbed executes a test script that typically contains instructions to simulate the communication between MMI and the SUT. Usually, an MMI calls functions from the framework and registers callback functions called by the framework later on. The testbed forwards such function calls to a so called *test driver*. The *test driver* is responsible for converting parameters and actually invoking the corresponding function of the SUT. The SUT again, will trigger an event normally sent to the protocol stack or to a driver. The testbed intercepts this event that can be consumed by an instruction of the test script. The event that is responded by the protocol stack in normal operation is now responded by an instruction of the test script. The SUT receives this event and optionally invokes a call back function whose invocation can be verified from the test script again.

4.1. Scripting Language

For our testbed project, we defined a new language Tbl (Testbed Language) in order to establish a minimal language including first-class concepts used for testing. Thus, Tbl can be considered as a domain-specific language (DSL) intended for implementing test cases for mobile systems.

The example in Figure 4 shows language elements intended for writing test scripts within the domain of mobile systems. The instruction in line 1 simulates a function call from MMI to the SUT which in turn sends an SDL signal `MN_ATTACH_REQ` to the protocol stack. This event is consumed and verified from the statement in line 2. To simulate the response back from the protocol stack to the SUT, an SDL signal is created and initialized (lines 3-5) and sent to the SUT in line 6. Triggered by this event, the SUT will call a callback function from the MMI. This call is consumed and verified by the script in line 7.

```
1: ExecSync("Driver.Method",{1, 2, 3});
2: WaitForSDLSignal(MN_ATTACH_REQ);
3: cnf = CreateSDLSignal(MN_ATTACH_CNF);
4: cnf.Param1.mcc = 1;
5: // ...
6: SendSDLSignal(cnf);
7: WaitForEvent("callback");
```

Figure 4. Example test case

When expected events (lines 2 and 7) are not sent by the SUT, the test case execution aborts with a corresponding error message. As shown in this example, the language provides statements to create, send, and wait for events of a certain kind like SDL. Such statements are required to simulate both the MMI above and the protocol stack under the SUT.

In the rest of this section, we go through the most important language features to show the simplicity of the language and to demonstrate how domain concepts are considered in the language.

For reasons of simplicity, Tbl knows only two kinds of literals: numbers and strings, whereas logical values are number values 0 and 1 Literal values may be assigned to variables in the following way.

```
i = 10;
s = "abc";
b = i > 10;
```

Tbl is dynamically typed so the variable type is inferred from the right side expression of an assignment. After the assignment, variables have a type that can be checked for other operations. For example, the expression `i == s` leads to an error as a number value cannot be compared with a string. Tbl knows arrays of a single element type or of variant element types.

```
a[] = {1,2,3};
b[] = {1, "abc", 3};
```

Arrays have a fixed length that is checked at compile time or runtime, depending on the index expression.

```
a[4]; // error at compile time
i = 4;
a[i]; // error at runtime
```

Tbl knows structure types commonly used to define the payload of signals, as shown at line 3 in Figure 4. To verify the contents of a structure variable, all fields can be converted to a string and compared to a string representation.

```
s.a = 1; s.b = 2; s.c = 3;
Assert("{1,2,3}"==ToString(s), "...");
```

This avoids large number of statements to verify the value of individual structure fields.

Tbl contains statements intended for writing test cases. The function `DebugOut` sends a string to the trace console for logging purpose. The function `Assert` evaluates an expression and terminates the script execution when the expression is evaluated to 0. The function `Wait` holds on the execution of the script for a certain amount of time.

```
DebugOut("Waiting for signal ...");
Assert(i == 0, "...");
Wait(1000);
```

Tbl provides a function to call test drivers with optional arguments.

```
ExecSync("Driver.Method");
ExecSync("Driver.Method", {1,2,3});
```

The first parameter of this function is an arbitrary string that identifies the test driver. The second (optional) parameter is an array of parameter values that are mapped to method parameters of the test driver.

To simulate the protocol stack for both sending and receiving events from a test script, the language provides functions to create, send, and receive such events. Events like SDL signals optionally have a payload containing data sent between processes. Tbl provides functions for every event mechanism to create the payload of an event. In case of SDL, the function takes the signal name to create the corresponding signal payload on C++ side. The function returns a handle to a variable that can be used to initialize the payload of the event.

```
s = CreateSDLSignal(MN_ATTACH_CNF);
s.Param1.mcc = 1;
s.Param1.mnc = 2;
```

The function `SendSDLSignal` sends an event to the SUT with or without payload.

```
SendSDLSignal(s);
SendSDLSignal(CHR_CHARGE_IND);
```

Events without payload are sent by specifying the identifier, for example the SDL signal name (`CHR_CHARGE_IND`). Events with payload are created and initialized and sent by the event variable (`s`).

The function `WaitForSDLSignal` waits for an event sent by the SUT.

```
s = WaitForSDLSignal(MN_ATTACH_REQ, 1000);
Assert(s.Param1 == 0, "Failed");
```

The event is specified by a unique identifier, for example the SDL signal name. The function accepts an optional timeout value as second parameter.

For AT and SDL supported by our testbed no function is necessary to destroy an event. The payload of the SDL signal is destroyed automatically after script execution.

4.2. Script Code

Test scripts written in the Tbl language are compiled into a compact script code that is executed by the testbed. This decouples the testbed runtime from the actual scripting language and leads to more efficient scripts at runtime both in memory size and loading time.

The Tbl script code provides about 50 instructions that are executed by an interpreter. Instructions fall into two categories: general instructions and test instructions. The first category includes instructions for control flow, arithmetic and relational operations, assignments, and field access. The second category includes instructions for testing purpose like invocation of test drivers, sending and receiving events and so on.

4.3. Testbed Components

Figure 5 shows the main components of our testbed including main dependencies between components. Testbed accepts test scripts in compact code format generated by the script compiler from test scripts. Following components are part of our testbed.

- The *Loader* reads script code from various sources like flash file system or from a network and provides the code to subsequent components.
- The *Interpreter* gets the script code from the loader and executes instructions contained in script code. Besides internal instructions including

arithmetic operations and variable assignment, the interpreter is aware of domain-specific instructions like call a test driver or send an SDL signal. However, these instructions are not implemented directly in the interpreter but deferred to the event router.

- The *Variable Store* is a vector of variables available in the script code. Variables are variants containing a number, an integer, or a structure value. Values stored in variables are copied when a test driver is called or an event is sent. Payload from received events is copied to the variable store too.
- The *Event Router* component serves as router for all kinds of events like SDL signals or AT commands. The event router is responsible to send events and holds an event queue for every kind of event to store incoming events.
- The *Controller* provides an interface to control execution of test scripts and recording of tracing information.

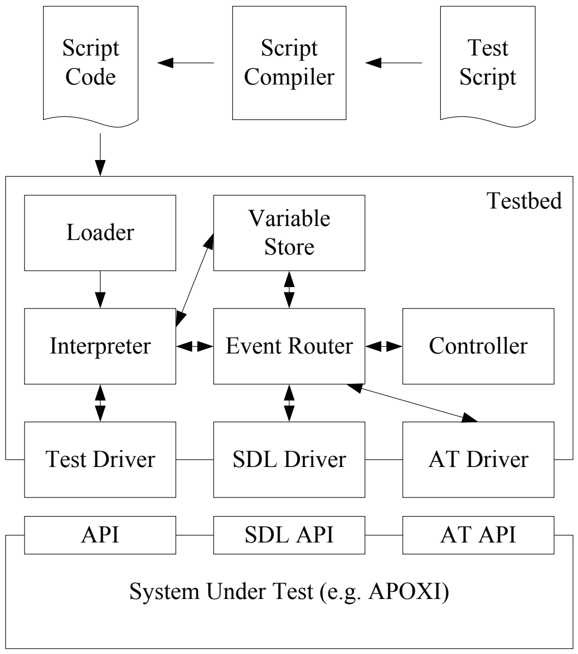


Figure 5. Testbed components

The presented components are basically self-contained and stable relating to a specific version of the SUT. This means, these components need not be changed when new functionality is added to APOXI and needs to be retested. To keep these components stable, the connection to the SUT is done by special driver components. *Test Driver*s contains bindings of a function call contained in a test script to a C/C++ function of the SUT. Testbed provides a generator to produce empty stubs of test drivers that are completed manually by the developer.

The *SDL Driver* binds SDL signals including optional payload to the SDL interface of APOXI to send and receive SDL signals. The driver contains interceptors and hooks to send and receive signals and a mapping of signal payloads to variables storable in the variable store. This mapping is generated automatically for every signal without any further assistance of the developer. As consequence, SDL drivers need to be generated again when new signals are added to the protocol stack or when the payload structure of an existing signal changes. In the same way, the *AT Driver* binds AT commands to the AT interface of APOXI. In contrast to SDL, AT commands are strings only so a generic mapping between AT commands and testbed variables is sufficient.

5. Tool Support

In this section, we give a rough overview of tools provided for our testbed for following tasks:

- generate test drivers
- compile test scripts into compact code
- write, maintain, and run test scripts

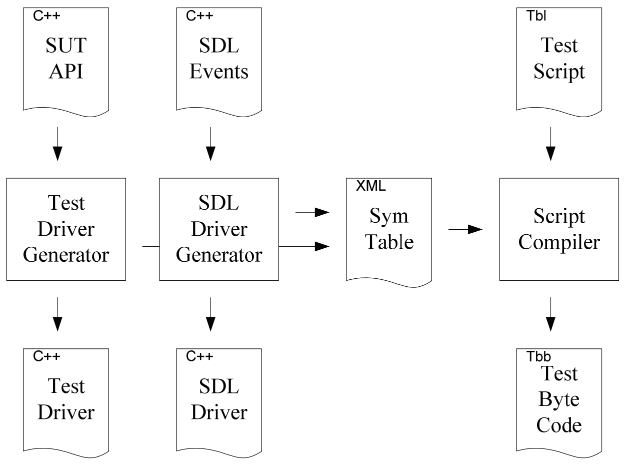


Figure 6. Testbed tool chain

Figure 6 shows the tool chain available for our testbed environment together with ingoing and out coming artifacts. All tools shown in this overview are implemented in the Java™ programming language.

5.1. Generate Test Drivers

The purpose of test drivers is to bind function calls contained in a test script to a function/method of the

SUT. The *Test Driver Generator* (see Figure 6) parses the programming interface of the SUT and generates empty stubs for test drivers, whereas every test driver is intended to call exactly one function of the SUT. Developers have to complete this generated code to map parameters and the return value to actual parameters of SUT functions. Consider following C function part of the programming interface:

```
BOOL NetworkRegistration(Int a, Word b);
```

The *Test Driver Generate* generates following test driver for this function.

```
String TestNetworkRegistration(
   const String& a, const String& b) {
  Int p_a;  Word p_b;
  /* Add conversion here. */
  Bool result = NetworkRegistration(p_a, p_b);
  return "Not implemented";
}//TestNetworkRegistration
```

Initially, the generated code contains a one-to-one mapping of functions available in test script and API functions. Testbed provides functions to convert string representations of standard C/C++ primitive data types to the actual C/C++ types. As most systems do not use C/C++ types directly but define own data types like `Int`, `Word`, and `Bool`, conversion is not generated automatically but deferred to the developer.

Besides the actual test driver, the generator also generates code that is used by the interpreter to invoke the corresponding test function. The *Test Driver Generator* does not depend on an actual SUT but can be reused for all SUTs implemented in C/C++ programming languages.

The *SDL Driver Generator* generates C source code to send and receive SDL signals and to convert signal payload structures from testbed variables to SDL structures and vice versa. SDL signal names as well as signal structures are defined as texts and then compiled into the target language, for instance, into C in case of APOXI. To generate the mapping for testbed scripts, both the text representation and the generated C/C++ source code can be used.

Both generators produce C/C++ source code but also an XML representation of parsed content. This content constitutes a *symbol table* that is used later on by the script compiler to validate the script contents with respect to available test drivers, for instance, available SDL signal names.

Both generators are available as command line program and integrated with our interactive testbed environment to write, maintain, and run test scripts. The command line versions allows the generation of test drivers as part of the build process to ensure accurate versions of test drivers that correspond to latest version of the SUT.

5.2 Compile Test Scripts

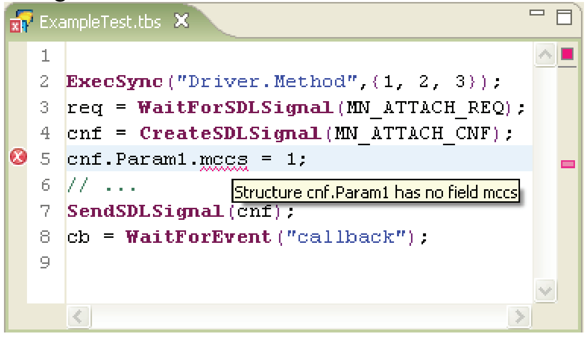
The *Script Compiler* takes a set of test scripts and a symbol table file (see Figure 6) to generate the script code that is executed by the interpreter. The script compiler performs lexical, syntax, and semantic analysis and generates the actual script code.

The front-end (lexical and syntax analyzer) is generated by the Java™-Version of the compiler-generator CoCo/R [8]. This generated front-end constructs an explicit abstract syntax tree (AST) for a parsed test script. The abstract syntax tree in turn is the input for code generation and the script editor described in the next section.

5.3 Write and Maintain Test Scripts

Test scripts are plain text documents that can be manipulated with any editor. However, writing test scripts in adequate time and with adequate quality requires same full-featured editors and tools as writing program source code. We have developed a full-featured development environment based on the Eclipse platform [9] with editor, problem view, console view, launch configuration dialogs etc. as expected from a development environment.

The editor, shown in Figure 7, provides syntax highlighting, problem decoration, and code completion. The editor relies on the symbol table generated by driver generators to provide code completion and validation for structure fields and event names like SDL signals. For instance, the editor in Figure 7 shows an unknown structure field in line 5.



```
1
2 ExecSync("Driver.Method",{1, 2, 3});
3 req = WaitForSDLSignal(MN_ATTACH_REQ);
4 cnf = CreateSDLSignal(MN_ATTACH_CNF);
5 cnf.Param1.mccs = 1;
6 // ...
7 SendSDLSignal(cnf);
8 cb = WaitForEvent("callback");
9
```

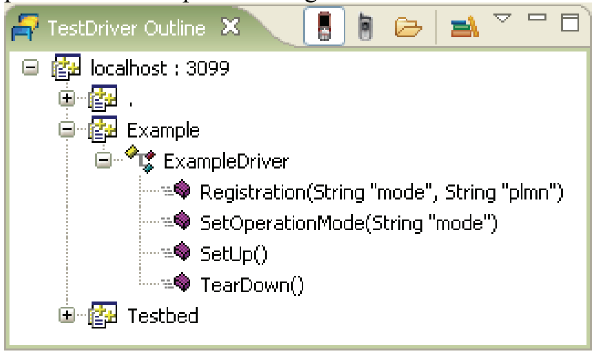
Structure cnf.Param1 has no field mccs

Figure 7. Test script editor

Providing a first-class environment for testers to write and maintain test scripts is crucial for successful and economic testing. In particular, maintenance of a huge set of test scripts between different versions of SUTs where signals are added or removed, payload structure of signal changes etc. can turn out to be cumbersome activity without support by tools to detect changes immediately.

The editor is also integrated with the testbed to execute test scripts immediately both on the host and on the target. Tracing information and test results are written to a console.

An outline view shows test drivers available in a testbed configuration. The view is connected to the testbed on the host or on the target and displays all test drivers including names, methods, and method parameters as depicted in Figure 8.



```
localhost : 3099
  .
  Example
    ExampleDriver
      Registration(String "mode", String "plmn")
      SetOperationMode(String "mode")
      SetUp()
      TearDown()
  Testbed
```

Figure 8. Test driver outline

This view enables the tester to execute a selected test driver immediately. Optional parameters can be entered on the fly. This gives the tester the possibility to execute a test driver and to observe the reaction of the SUT. This feature is most valuable with the *Recorder* that records events between the SUT and the protocol stack.

The *Recorder* can be used to record events between the SUT and the protocol stack. This is useful for reproducing the communication with the real protocol stack within a test script to reproduce communication from a real scenario.

6. Related Work

The testbed presented in this paper combines scripting techniques with a domain-specific approach for a testbed in domain of mobile software systems and provides a rich tool set to create, maintain, and execute test scripts. We give a rough overview of related work in these areas.

Scripting languages have been successfully applied in various fields of software testing. As pointed out in [10], "*scripting techniques are identified for test automation for their incredible features like reuse of source code, simpler scripting languages and easy to maintain*". For example, TAPIoca [11] is a Tcl/Tk based automatic test environment. It can send commands to a SUT, get responses from the SUT, and control signal generation equipment connected to the same SUT. Like other approaches reusing an existing scripting language, Tcl/Tk was chosen as the basis for this automatic test environment because of the wealth and maturity of its "language" features, as well as its ease of extensibility.

Besides normal scripting languages, some approaches depend on certain testing languages. Jammer [12] is an interactive script language processor designed to enable testing of the prototype ATM switches designed at Washington University. TTCN-3 [13][14] includes a testing language for specification and implementation of test cases. TTCN-3 supports any kind of black-box and grey-box testing for reactive, centralized or distributed systems in various domains like telecommunication and embedded systems in automotive industry.

The concept of domain-specific testing has also been recognized by researches to improve various aspects concerning testing.

Domain-specific testing is already recognized for automatic test case generation. Sinha [15][16] proposes a technique for domain-specific testing of applications that uses HaskellDB specifications of the software to extract domain-specific properties and embed them into the test generation model. Siddhartha [17] provides automated test development support in the form of a program synthesizer. The synthesizer transforms a domain-specific, formal test-specification into a test driver, which automates test execution and test result verification. The commonality of these approaches is that domain-aspects are used to generate test cases in general programming languages. As result, generated test cases are more complex compared to domain-specific language.

The *Software Testing Automation Framework* [18], initially developed by IBM, is an open source, multi-platform and multi-language framework designed around the idea of reusable components. Bhaskar Rao.G and Albee Vimal [19] report about SDL based test automation for real time systems.

Various commercial and non-commercial vendors provide tools for the the TTCN-3 testing language. For example, OpenTTCN [20] provides compiler that is used to execute TTCN test suites, has open test management and control, platform adapter, and SUT adapter interfaces that allow creation of test systems by integrating existing software and hardware components.

7. Conclusion

We have presented requirements and a testbed in the field of mobile software frameworks. Many of the presented requirements and challenges that stem from experience with the APOXI framework are important in general. The APOXI testbed presented in this paper is novel concerning the usage of a scripting language for unit/module/system testing of mobile software frameworks. The testbed is currently used by developers and testers of the framework. Developers write test drivers for their responsible modules and specify test scripts concerning unit testing and module testing. Testers reuse test drivers, complete module tests and provide test scripts for system testing of the APOXI framework.

The usage of our testbed for another platform in domain of mobile systems showed us that the testbed is rather independent of a certain platform or framework within the same domain.

The usage of a scripting language together with full-fledged tool support to create, maintain, and execute test scripts turned out to be a very promising and successful way. In particular, a single environment for continuous unit/module/system testing facilitates the automation of testing. A further advantage comes from integration of domain-specific aspects in this language that simplifies test scripts concerning readability and maintainability.

8. References

[1] J. Pichler, G. Reisinger, H. Prähofer, and G. Leonhartsberger, "Aragon: An Industrial Strength Eclipse Tool for MMI Design for Mobile Systems", *Proceedings of the IASTED International Conference on Software Engineering*, ACTA Press, Anaheim, CA, February 2007, pp. 156-163.

[2] GSM 07.07. Digital cellular telecommunications system (Phase 2); AT command set for GSM Mobile Equipment (ME). ETSI TS 100 916 V7.5.0, European Telecommunications Standards Institute, 1999.

[3] Z.100 Functional Specification and Description Language, Recommendation Z.100 – Z.104, ITU-T, March, 1993.

[4] Fewster, M. and D. Graham, *Software Test Automation*, Addison-Wesley, 1999.

[5] M. Feathers, CppUnit, http://cppunit.sourceforge.net/cppunit-wiki/FrontPage, accessed January 2008.

[6] Parasoft, C++test, www.parasoft.com, accessed January 2008.

[7] SpiderMonkey, http://www.mozilla.org/js/spidermonkey/, accessed January 2008.

[8] Terry, D.P, *Compiling with C# and Java*, Addison-Wesley, 2004.

[9] Gamma, E. and K. Beck, *Contributing to Eclipse – Principals, Patterns, and Plug-Ins*, Addison-Wesley, Reading, MA, 2004.

[10] R. Pallavi, "Software Test Automation – Scripting Techniques", 3rd Annual International Software Testing Conference, Bangalore, India, 2001.

[11] A. Flick and J. Dixson, "Using TCL/TK for an Automatic Test Engine", *Proceedings of the Sixth Annual Tcl/Tk Workshop*, San Diego, CA, 1998.

[12] M. Beal, *Jammer Language Description: A Script Language for GigaBit Switch Testing*, Applied Research Laboratory Working Note ARL-96-01, 1996.

[13] Grabowski, J., D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, "An introduction to the testing and test control notation (TTCN-3)", *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Volume 42, Issue 3, 2003, pp. 375-403.

[14] ETSI ES 201 873-1 (V2.2.1): *Methods for Testing and Specification (MTS)*; The Testing and Control Notation version 3; Part 1: TTCN-3 Core Language, October, 2003.

[15] Sinha, A., C.S. Smidts, and A. Moran, "Enhanced Testing of Domain Specific Applications by Automatic Extraction of Axioms from Functional Specifications", *Proceedings of the 14th International Symposium on Software Reliability Engineering*, IEEE, Los Alamitos, CA, 2003, pp. 181-190.

[16] Sinah, A, *Domain Specific Test Case Generation Using Higher Order Typed Languages for Specification*, Ph.D. Thesis, University of Maryland, College Park, MD, 2005.

[17] A. Reyes and D. Richardson, "Siddhartha: A Technique for Developing Domain-Specific Testing Tools", *Proceedings of the 14th International Conference on Automated Software Engineering*, Cocoa Beach, FL, 1999, p. 81.

[18] C. Rankin, "The Software Testing Automation Framework", *IBM Systems Journal*, Vol. 41, No. 1, 2002. p. 126.

[19] R. G. Bhaskar and A. Vimal. "SDL Based Test Automation for Real Time Systems Testing", *3rd Annual International Software Testing Conference*, Bangalore, India, 2001.

[20] OpenTTCN: Testing and Test Control Components for Tester Developers, http://www.openttcn.com/Sections/Products/, accessed January 2008.