"Faust" – a fully configurable automatic software test system

Dr. Anne Kramer, sepp.med gbmh

Gerhard Baier, AFRA GmbH

Testing CAN applications requires complex test systems. Several interfaces are tested simultaneously in real-time conditions. Some components might not yet be available leertaste anzeigen and must be simulated. The simulation must be as straightforward as possible – otherwise, precious time is lost in testing the simulation rather than the real system. While specialized hardware test tools exist that cover some of these requirements, they usually do not support persistent archiving and test management.

.faust is a fully configurable automatic software test system that combines aspects of quality assurance (version management, audit trails, user management,…) with requirements for real-time tests of multiple interfaces and/or components. Its kernel contains basic functions to test standard hardware protocols (CAN Bus, CANopen, LIN, Ethernet etc.), which are completed by a set of parameters describing the particularities of the component under test. A powerful scripting language offers the possibility to send, receive and process complex data or events to test diverse software interfaces and/or CAN-Bus protocols. Error situations and missing components can be simulated. Test data and results are stored in an Oracle database.

In this paper we will present this tool that has been specifically designed for tests of embedded software in highly complex, safety-critical environment.

1 Introduction

Companies working in complex, safety-critical domains have to fulfill highest quality standards. For example, medical products must comply with standards such as the Medical Products Act (MPG) or the regulations of the American Food and Drug Administration (FDA). Similar standards exist in the automotive industry. They require exhaustive tests and complete traceability of results and changes. Appropriate tools exist on the market for requirement management, change control, configuration management and possibly test management.

However, products are becoming more and more complex. The amount of embedded software is continuously increasing. Testing hardware interfaces imposes additional requirements on the test system. First, tests are subject to real-time conditions similar to the productive environment. Second, it might be necessary to test several interfaces simultaneously. Third, some components might not be available yet and must be simulated. It is crucial that the development of these simulations is as simple and straightforward as possible. Otherwise, precious time is lost in testing the simulation rather than the real system.

We wanted to use a test frame generator to develop appropriate test scripts at an early moment, since this helps reducing costs. If the execution of these test scripts can be automated, they can be repeated at any moment, e.g. after integration of a modification or during acceptance tests. We, therefore, searched for a combined test frame generator / test management tool that supports real-time tests of hardware interfaces.

2 Existing approaches in research and industry

2.1 Test frame generators

Different test frame generators are available on the market. These systems generate rough test frames that have to be adapted manually to fit the specific functionality. Some of these tools are designed for specific programming languages or application domains, e.g. C++, Java, Web applications (e.g. see [SLi04, p. 174]). Others rely on a particular development environment (e.g. Rational).

The advantage of these tools consists of the automatic generation of identical frames for different applications. The inconvenience is that these frames must be filled with code each time a new application is introduced.

There are no test systems that automatically generate test frames without any further manual adaptations, essentially because no formalism is defined to describe the system under test.

Some approaches for test data generation have been developed (see for example [Bred04]) according to specific coverage criteria [Bal98, p. 391]. Likewise, approaches exist for test case generation both in research and in industrial domains, but there is no tool to generate complete test frames, i.e. all programs that are required to execute tests [SLi04, p. 215]. However, this is of particular importance when testing embedded systems, as tests of various hardware and software configurations are required – a time and cost consuming task.

2.2 Test management tools

For test management tools, the variety of commercially available solutions is even more diverse, varying from open source software up to relatively expensive products (for an overview, see [Fau04]). The choice is quite exhaustive and you will probably find whatever you need off-the-shelf.

2.3 Hardware test tools

Finally, specialized hardware test tools exist such as CANoe (Vector Informatik GmbH, Stuttgart) [Vec04], canAnalyzer (IXXAT Automation GmbH, Weingarten) [Ixx04] or X-Analyser (Softing AG, Haar/München) [Sof04]. These tools are designed to test hardware layers, but do not focus on software interfaces. Moreover, they generally do not support persistent archiving and configuration management of test data.

3 Our software test system

In order to combine the aspects of quality assurance (version management, audit trails, user management) with the requirements for real-time tests of multiple interfaces and/or components, we developed our own software test system (called .faust). With .faust we can test diverse interfaces, software, e.g. Dynamic Link Libraries (DLLs) and hardware protocols (e.g. CAN-Bus, CANopen, LIN, Ethernet, V24...) simultaneously and in nearly real-time conditions.

Our test system is based on a modular design. Functions to test standard hardware protocols are part of the system kernel. This kernel can be extended dynamically just by adding further software components (DLL, COM).

A powerful scripting language offers the possibility to send, receive and process complex data or events using common script language elements such as data type definitions, loops etc. This makes it possible to respond to incoming messages and their content in a specific way and even to simulate scenarios for emitter and receiver.

The integrated recording mechanism offers the possibility to highlight or filter messages and to log interpreted messages based on the project description.

Test data and test results are stored in an Oracle database. Different versions are kept so that it is possible to track changes and to restore older versions. This is of particular importance for tests of safety relevant components such as medical devices which is one of our domains.

3.1 Testing embedded software

Figure 1 visualizes the integration of an embedded system into the software test system. In this specific example, two interfaces are tested simultaneously. For example, these interfaces may be a software interface (API) and a CAN bus component.

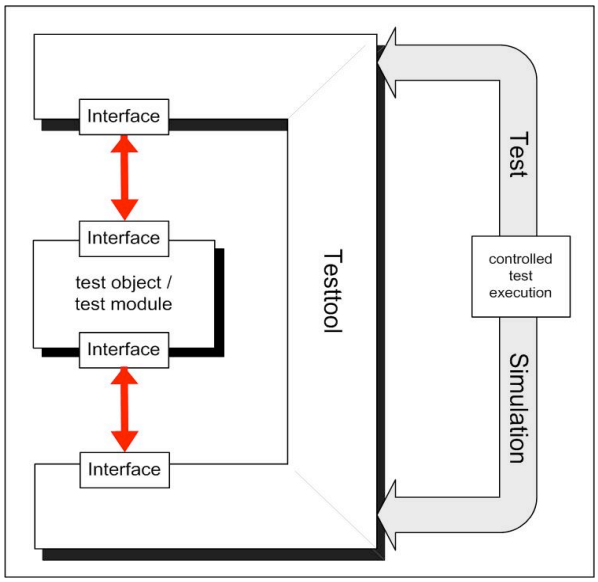

Figure 1: Integrating the test object

The integration of the test object is done automatically without any manual rework. Also, no recompilation of the test system is required. The basic functions to address hardware interfaces (e.g. bit patterns to address a CAN bus) are integral part of the test system. They are completed by a set of parameters that describe the particularities of the component under test which are specified in XML description files. For CAN bus systems, these description files contain the telegram structure and the valid variables together with their symbolic names. The test system will then display telegrams in a readable (interpreted) form. The parameter description is also required for simulation test scripts.

Software interfaces are integrated by adding libraries (DLLs), a fact that considerably enhances the reusability in other projects. The integration of DLLs under test depends on its class model. For example, .NET DLLs can be integrated via reflection.

Especially for software interfaces it is possible to generate the description files automatically by analyzing an appropriate specification of the component, e.g. using UML 2.0. The methods are extracted from the UML model of the test object and stored in the description files.

3.2 Operating system

We chose Microsoft Windows as operating system, and this even for tests under real-time conditions since Windows is commonly used as development and test platform. In addition, various connections to hardware interfaces are already available using Microsoft Windows. Also, the software should be able to handle hardware independently from the manufacturer.

These premises possibly conflict with other requirements resulting from machine-oriented software development. When testing hardware interfaces it is essential to react within short delay (real-time systems). Using Windows we usually cannot guarantee a fixed response time. However, this is not always essential. In the majority of tests, it is possible to split the task into parts and, thus, to obtain a practicable solution, provided that no other software than .faust is running on the test computer.

We have to distinguish two domains:

1. Reporting

Obviously, no message exchanged via the interface may be lost. The required frequency depends mostly on the physical interface. Using special hardware .faust reaches a time resolution of approximately 1 micro second.

2. Simulation

A simulated component has to respond to bus signals within a given delay. The response time of a simulation in .faust is below 1 ms.

3.3 Simulation

Using the integrated scripting language it is possible to simulate data exchange, events, or error situations. This scripting language is quite similar to C, based on procedures and completed by special functions to simulate communication via

interfaces. To reach the time resolution of 1ms or below, elements of the test scripts are precompiled in multi-thread capable runtime structures that will then be executed as fast as possible.

```
Bool bExitLoop = false;
Int32 nController = 0x12;

respond until ( bExitLoop )
{
   case CAN.SwitchLightNo42On :
   {
      CAN.Send ( nController,
                  nIsLighingNo42On);
   }
   case CAN.SwitchLightNo42Off :
   {
      CAN.Send ( nController,
                   nIsLighingNo42Off);
   }
   case CAN.any:
   {
      CAN.Send ( nController, nUnknown);
   }
}
```

Figure 3: Script example

Messages are interpreted depending on the interface as defined in the description files mentioned above. For example, if the bit pattern “11101010b” is defined as “SwitchLightNo42On” in the description file, the test protocol will contain the symbolic value rather than the bit pattern. This considerably enhances the readability of the test protocols and allows a more abstract view of the interface. In the same way, the mapping between composite data structures (e.g. tables) and the corresponding interface-specific messages can be defined in the description files.

Another important element of the .faust scripting language is the “respond/until” command. Analogous to the switch command in C, “respond/until” reacts on input data (see fig. 3). It is particularly important for simulations of micro controllers. The simulated micro controller has to react to incoming messages. Its response behaviour is defined in small test scripts using “respond/until”. Its response time is governed by the actions defined in the case branches which must be designed accordingly.

3.4 Integrated test management

Test cases are designed directly in the test system. It is possible to combine test cases into modules, i.e.:

- test sequences (test cases forming a building block),
- test scenarios (set of interdepen-dent test cases and sequences) and
- test suites (set of independent test cases, sequences or scenarios that belong to a specific test phase, e.g. integration test or system test).

These modules are held under configuration management. It is possible to restore older versions simply by selecting the version number from a dropdown list. All elements are versioned independently. Test cases and modules are released on a separate tab.

Starting from a complete state transition diagram of the system under test, it is possible to generate test suites automatically using a proprietary test case generator. These generated test suites are then imported into .faust. They contain a set of test cases that test different combinations of state transitions up to a configurable limiting path depth. Each transition corresponds to a basic script or function. These scripts are implemented once and for all test cases contained in the suite. Thus, a great variety of transitions can be tested seamlessly.

Single test cases or entire modules can be executed either manually or automatically within the test system. We distinguish between “ad hoc tests” and “documented tests”. Results of ad hoc tests are not stored anywhere and will be used for quick verifications. The results of documented tests are automatically saved and versioned. It is also possible to save the test run planning, i.e. the selection (and order) of test cases and/or modules that are part of a specific test run. Saved test runs can then be executed again either manually or in batch mode.

The test execution results are then analyzed by the user. These interpretations can then be stored in the relational database for further evaluations.

.faust keeps audit trails for all relevant elements, i.e. it is possible for all versions to reconstruct when and by whom an object was released.

Import and export interfaces exist to requirement management tools of the customer (e.g. IBM Rational Requisite Pro, TeleLogic DOORS).

To assure access right restrictions, the users of the test system are assigned to roles with specific rights. For example, users with the role “Tester” will normally not be able to modify test cases. Every user has to login into the system with ID and password.

4 Advantages

Using our fully configurable, automatic software test system, we can conduct early and thorough tests. Missing components can be simulated. The integration of interfaces can be done rather seamlessly, which helps us to react efficiently to find existing errors and errors of modifications which have to be integrated. Tests can be repeated easily, the test execution is reproducible, and error situations can be simulated for developing robust systems. All test results are contained in the database and are automatically documented through the reporting functionality.

The test system can be integrated into existing development processes, but we also use it as stand-alone test system for service and support. For the latter, a version of the test project is released at a given state. This release version contains a fixed status of test scripts and suites. The service personnel run these tests on site. The test results can be re-imported into the database for further analysis.

Last but not least it is possible to run tests over a longer period. Afterwards, the event log can be analyzed offline. This is of considerable help in case of sporadic errors and errors that are difficult to reproduce.

5 Summary and outlook

In this paper we presented our fully configurable automatic test system called .faust, which was specifically designed for tests of embedded software in highly complex, safety-critical environment (medical devices, automotive industry). This software test system combines aspects of quality assurance (e.g. version management, audit trails, user management) with requirements for real-time tests of multiple interfaces and/or components.

.faust is “fully configurable” in the sense, that test objects are added without further recompilation. Software interfaces are integrated just by adding libraries. For hardware interfaces, the basic functions are already part of the system kernel. The particularities of the component under test are described in a parameter description file that is added to the system.

It is “automatic”, because test cases can be executed automatically. The test system is used as stand-alone tool for service and maintenance, where it is especially helpful to identify sporadically occurring errors. Moreover, it is possible to automatically generate test cases from state transition diagrams, if a test case generator is used.

Customized versions of the test system are used by companies in medical industry (SIEMENS Medical Solutions, Philips). The full version is used internally.

In the future, we plan to integrate test case libraries to further enhance interface independent test case design. The user will then compose modules using elements from different libraries. Moreover, we plan to enable electronic signatures.

References

[Bal98] Balzert, H.: Lehrbuch der Software-Technik. Heidelberg 1998.

[Bred04] Brederek, J.: Suchalgorithmen für die Testdatengenerierung. AGBS-Kollo-quium, Universität Bremen, 2003-07-18 http://www.tzi.de/~brederek/papers/suchalgorith-test.pdf

[Fau04] Faught, D.: Test Drivers and Test Suite Management Tools. http://testingfaqs.org/t-driver.html

[Ixx04] IXXAT Automation GmbH (Hrsg.): canAnalyser - Das leistungsstarke CAN-Werkzeug für Entwicklung, Test und Service http://www.ixxat.de/cananalyser_de,569,147.html

[SLi04] Spillner, A., Linz, T.: Basiswissen Softwaretest. Heidelberg 2004.

[Sof04] Softing AG: X-Analyser http://softing.com/de/communications/produkte/can/tools/x_analyzer.htm

[Vec04] Vector Informatik GmbH: CANoe / DENoe 5.2 http://www.vector-informatik.de/deutsch/index.html?../produkte?canoe_features.html

All links were called on April 11, 2006.