# Embedded TestFrame

## An Architecture for Automated Testing of Embedded Software

Harro S. Jacobs and Peter H.N. de With

*In this paper, we introduce an architecture for automated testing of embedded software, called* Embedded TestFrame*. Testing is performed at two levels: 1) test specification based on spreadsheets and 2) test implementation using mature programming languages. In addition, test implementation is partitioned over a test computer and the embedded system, to minimize the overhead for the embedded system. The use of mature programming languages is advantageous, because experience with and tooling for these languages is widespread. The use of spreadsheets supports an abstract test specification in an early stage without having the final interface available of the software to be tested. We have successfully implemented this architecture at Philips Semiconductors, where Embedded TestFrame has been accepted as the primary solution for all test activities.*

## Introduction

With the advent of digital television and set-top boxes, embedded systems which were conventionally performing control only, have become so powerful that a multitude of processing tasks, including applications and user interaction, are carried out. Recent architectures for high-end digital audio and video systems contain (multiple) 32- or even 64-bit CPUs and DSPs and up to 64 KB RAM. The corresponding embedded software shows a strong complexity increase due to augmented memory size. As a result, the total development time is increasingly determined by the software development time.

Due to the complexity and size of embedded software together with strong demands on time-to-market and quality, testing is a crucial point that should be addressed during software development. Traditionally, testing is carried out during the last phases of the software development life cycle. As a consequence, testing activities are often subject to high time pressure, which either results in delayed market introduction or low product quality. Furthermore, high recall costs for embedded systems should be avoided.

In this paper, we propose a novel architecture for automated testing of embedded software, named Embedded TestFrame, featuring the possibility to start test development in an early stage. We advocate an incremental approach for test development that can already be started as soon as the first requirements are fixed. Test execution can then take place as soon as the first component[1] is developed and thus provide early feedback in case of errors. The advantage is that the effort can be spread over a longer, better manageable period.

[1] In this paper, we do not have the intention to distinguish between components, modules, etc., but use the term ‘component’ for any clearly defined piece of software that can be tested.

During software development, it is advisable to re-execute tests for completed components on a regular basis, because context changes may impact components that were considered to be correct. Moreover, re-execution of tests plays a crucial role during software maintenance, where new releases should be verified thoroughly. In conclusion, many situations exist in which it is required to repeat test execution regularly. In these situations, automated testing is often cost effective. The benefit of automated tests is that they provide a rapid though very reliable and reproducable statement of the product quality. As such, repeated execution of automated tests gives a good indication of the product quality over time, offering valuable metrics for project control. For the above-mentioned reasons, automated test execution has been adopted as a key feature of Embedded TestFrame.

Development of an automated test suite must not be underestimated, because test suites often turn out to be equally large or even larger than the software to be tested. One should always be aware of the trade-off between effort and (knowledge about) product quality; one may choose to only automate tests for very critical components, and to do manual tests for the remaining system parts.

## Characteristics and Requirements

Prior to presenting an architecture for the automated testing of embedded software, the key characteristics of embedded software are discussed and the corresponding requirements for the architecture are mentioned.

### Relatively high complexity of software

The complexity of embedded software is rapidly increasing. As mentioned before, the size of a test suite may become very large, and sometimes even exceeds the size of the software to be tested. Thus, an architecture for testing embedded software should enable a controlled and incremental development of test suites.

### Large variety of embedded systems

Embedded software runs on dedicated embedded systems, which will be referred to as targets from now on. A large variety of targets exists given the broad choices of processors, boards, (real-time) operating systems, programming languages, development environments, etc. An architecture for the automated testing of embedded software must deal with this large variety.

### Resource-constrained targets

Typically, targets have constrained resources with respect to, for example, processing power and memory size. Although Moore's law - the periodical doubling of resource capacities - also applies to the embedded domain, embedded systems are often still not 'oversized', due to small profit margins. An architecture for testing embedded software should be apt to such situations and should provide means to keep the major part of a test suite outside the target.

### Software interfaces

An example of a software interface is the Application Programmers Interface (API) of the software to be tested, which can typically be controlled by software executing on the target. Other examples are those applications that provide for or absorb data of the software to be tested. An architecture for testing embedded software should enable the test suite to control this software interface.

### Hardware interfaces

Hardware interfaces are the interfaces on the physical boundaries of the target, which are controlled or observed by the embedded software. Examples are serial and parallel ports, but also manual switches, LEDs, and display devices. An architecture for testing embedded software should enable the test suite to control the hardware interfaces. The architecture should not be limited to a certain set of known hardware interfaces, but it should be extensible, because the number and variety of these interfaces are continuously growing.

### Software reusability and portability

Since the complexity of embedded software is increasing rapidly, components are no longer developed for a single system, but are applied in classes of systems. Therefore, reusability and portability of embedded software is of growing importance. Consequently, it must be possible to develop a test suite - or at least a large part of a test suite - that is target independent and can be used for a class of systems.

## TestFrame

### Method

We have developed a technique, called TestFrame, in order to deal with test suites of highly complex software (not specifically embedded software), see [1] and [2]. This technique makes a clear distinction between two phases: the test *specification* and the test *implementation* or *navigation*, which will be briefly explained.

**Test specification** In this phase, spreadsheets are used in which high-level keywords with parameters, i.e., action words, are listed. These action words are domain-specific and represent an abstract definition of the test stimuli and the expected responses. The spreadsheets are based on the software requirements and do not consider the actual interfaces of the software to be tested. The test developer defines the action words and constructs the spreadsheets.

**Test navigation** In this phase, the action words that have been defined during test specification, should be linked—or navigated—to the actual interfaces of the software to be tested. Sometimes, this link is a one-to-one mapping on the interface functions of the software to be tested. However, because of the allowed abstraction in the test specification, the test navigation can be considerably large.

### Architecture

The architecture of TestFrame is depicted in Figure 1. The figure shows the separation between the test specification and navigation, as well as the *TestFrame Engine* and the *test report*.

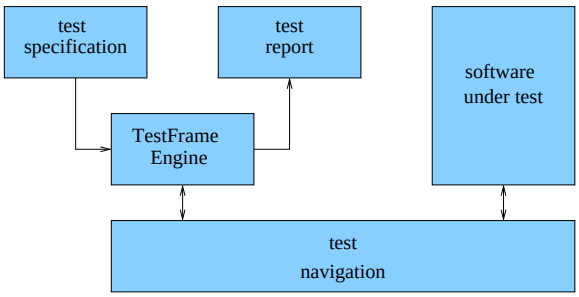


Figure 1: Architecture of TestFrame.

The TestFrame Engine is a batch tool, taking care of the test execution. It parses the test specification, i.e., the spreadsheets, and communicates each action word to the test navigation. The test navigation controls and observes the software to be tested. The TestFrame Engine generates a test report with a complete execution trace of the test as well as a management summary briefly showing which tests failed. The separation between the test specification and the test navigation allows for a structured development of test suites. The test specification can already be written when the first requirements are known. Test navigation can be developed in a later stage when the software and hardware interfaces of the software to be tested have been defined. Note that the distinction between test specification and navigation also allows for specialization in the project team, e.g., analysts writing test specifications, and software developers constructing test navigation.

## Embedded TestFrame

### Architecture

Embedded software is typically executed on a target with limited resources. For this reason, the Embedded TestFrame architecture uses a partitioning for minimizing the overhead on target, see Figure 2. The architecture distinguishes a test computer, i.e., the *host*, and a target. The host is used for storing large parts of the test suite, thereby minimizing overhead on the target.

As a result, the test navigation is split-up over the host and the target. An explicit communication means is required to communicate between host and target. Since many communication protocols (RS232, TCP/IP, JTAG, etc.) are available and proven standards for unified high-level communication are virtually absent, we developed *ActiveLink*. This tool offers a small-sized communication mechanism for transparent host-target communication at a functional level.

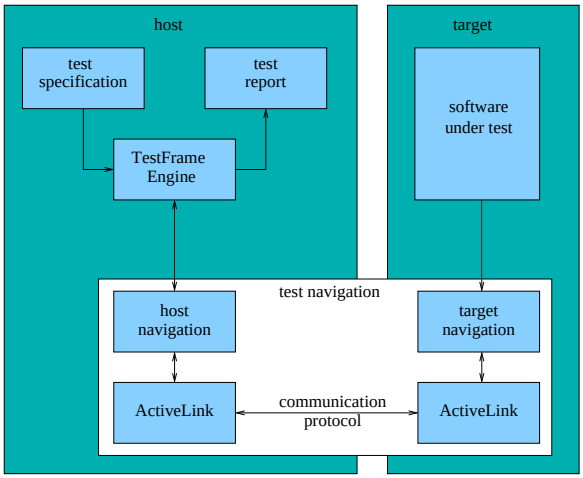


Figure 2: Architecture of Embedded TestFrame.

### Navigation

Figure 2 shows that navigation code can reside on the host as well as on the target. Navigation code should be developed in mature and proven languages (C, C++, Java). We explicitly avoided the development of a dedicated script language, because knowledge of and experience with proven languages is better available and tool support is mostly mature (integrated development environments, source level debuggers, etc.).

As discussed before, code on the target should be minimized and navigation should therefore as much as possible be implemented on the host. Although rules of thumb exist how this partitioning should take place, test developers are free to deviate and to apply a dedicated partitioning scheme. Another aspect of the partitioning is that host-target communication clearly influences the real-time behavior of the software to be tested. If this hampers testing, one should develop navigation code on the target that is critical for supporting real-time operation.

### Hardware interfaces

We discussed that the software to be tested can have hardware interfaces. Because of the variety of external interfaces, we do not strive for a library to control and observe all these interfaces. However, for many interfaces, such as serial and parallel ports, drivers are available. For other interfaces, such as manual switches and LEDs, dedicated hardware/software tools should be developed. Considering the effort, these are typical interfaces for which often is chosen to abandon automated testing, and instead to control these interfaces manually.

If tooling for external interfaces is available, it should be integrated in the navigation code on the host. Also in this case, the use of mature languages is beneficial as it eases integration. For example, Windows drivers for serial communication can be used in a straightforward way.

## ActiveLink

### Architecture

An important tool in the Embedded TestFrame architecture is ActiveLink that offers a seamless connection between host and target, while abstracting from the actual communication protocol. ActiveLink offers a Remote Procedure Call (RPC) mechanism as well as means to control remote memory, i.e., to allocate memory on target and to copy memory from host to target, and vice versa.

Because of the large variety of targets, the architecture of ActiveLink focuses on portability, see Figure 3. The figure shows two porting interfaces: the *platform interface* and the *protocol interface.*

**Platform interface** This interface abstracts from platform-specific details, such as the processor and the (real-time) operating system. For each platform, these specific details should be made available to ActiveLink, which has already been realized for Windows 95/NT, pSOS, and Posix. The platform interface enables us to port ActiveLink to other platforms with relatively little effort.

**Protocol interface** This interface abstracts from the actual communication protocol between host and target, and supports already communication over TCP/IP, PCI, and RS232. The protocol interface enables extending ActiveLink with any communication protocol as long as reliable bi-directional data transfer is available.

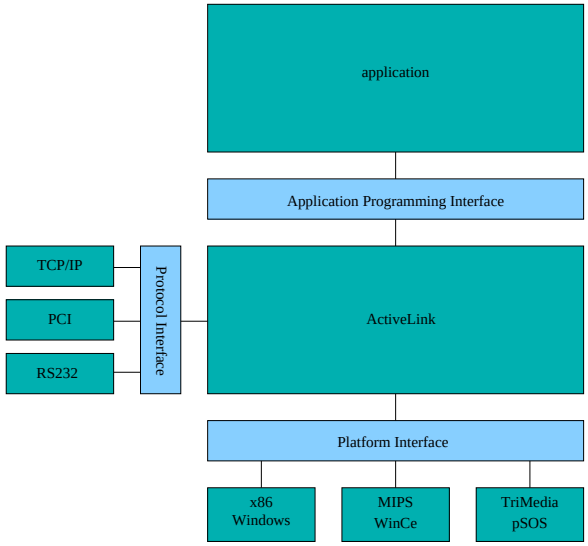


Figure 3: Architecture of ActiveLink.

### Wide applicability

ActiveLink has not been specifically designed for Embedded TestFrame; it is a highly portable tool for inter-platform communication on application level. Therefore, it enables development of distributed applications in heterogeneous environments. It can be used for other purposes as well, like remote maintenance and control, and remote diagnostics. Recently, we realized a tracing tool based on ActiveLink to analyze the dynamic behavior of embedded software.

## Implementation and Evaluation

Embedded TestFrame has been implemented successfully at Philips Semiconductors within the Software Services Group (SSG). This department develops reusable components for the domain of digital audio and video systems, such as digital televisions, set-top boxes (satellite receivers for digital video), and DVD players. Besides a PC-based simulation environment, SSG currently uses MIPS- and TriMedia-based systems with the operating systems pSOS and WinCE. Also dual processor solutions executing different operating systems are being used.

In the initial phase, Embedded TestFrame was used for the automated testing of a graphics component on these systems. An existing test application was integrated in the Embedded TestFrame architecture and a set of spreadsheets for additional test cases was written. ActiveLink was used for host-target communication to call the API of the graphics component. Additionally, ActiveLink was used for comparing bitmaps of the graphics component with reference files that were stored on the PC.

It was found that the choice for high-level languages C and C++, led to a steep learning curve for the test developers, because of their experience with these languages. The test suite was target independent and was executed on a periodical basis to test the component on these different systems.

The successful implementation of Embedded TestFrame and its ease of use has resulted in the full integration of this package in the SSG tool set, and it is currently being used for other projects as well.

## Conclusions

We have presented an architecture for the automated testing of embedded software. This architecture is generic and aids to structured development of test suites. Important requirements for this architecture are that it should cope with a large variety of targets and the constrained resources of these targets.

The presented architecture offers the ability to partition tests into three parts: test specification, test navigation on host, and test navigation on target. This partitioning is highly flexible, because it needs no *a-priori* decisions about *where* to put *what* functionality.

A key feature of the Embedded TestFrame architecture is that developers can concentrate on the test functionality, while two tools, i.e., the TestFrame Engine and ActiveLink, support the partitioning and hide the platform and interface specific features.

The successful introduction of Embedded TestFrame at Philips Semiconductors has resulted in a continued development of this architecture in order to cope with new technologies. It is our intention to expand the range of targets for using Embedded TestFrame and to increase the flexibility of this solution according to the needs of our customers.

## References


[1] CMG, *TestFrame, Een Praktische Handleiding Bij Het Testen van Informatiesystemen*, ten Hagen & Stam Uitgevers, ISBN 90-76304-67-X, Den Haag, 1999

[2] Hans Buwalda, Maartje Kasdorp, *Getting Automated Testing Under Control*, Software Testing & Quality Engineering, November / December 1999



In 1994, Harro S. Jacobs graduated in computing science from the Eindhoven University of Technology, The Netherlands. In 1996, he graduated in the post-graduate program on software design at the Stan Ackermans Institute in Eindhoven, The Netherlands. Since then, he is employed at CMG, first in Rotterdam and after two years in Eindhoven. Within CMG's Research Center Mission Critical Systems, he is working in the field of software architecture definition and evaluation. Within CMG's Competence Center Technical Software Engineering, he is heading the competence of automated testing. As such, he is working as software architect on further development and deployment of Embedded TestFrame.

Peter H.N. de With graduated in electrical engineering from the University of Technology in Eindhoven, The Netherlands. In 1992, he received his Ph.D. degree from the University of Technology Delft, The Netherlands, for his work on video bit-rate reduction for recording applications. He joined Philips Research Laboratories Eindhoven in 1984, where he became a member of the Magnetic Recording Systems Department. From 1985 to 1993 he was involved in several European projects on SDTV and HDTV recording. In the early nineties he contributed as a video coding expert to the DV standardization committee. In 1994 he became a member of the TV Systems group, where he was working on advanced programmable video processing architectures. In 1996 he became senior TV systems architect and in October 1997, he was appointed as full professor at the University of Mannheim, Germany. At the faculty Computer Engineering, he was heading a group working on video coding, processing and its realization. Since August 2000, he is with CMG Eindhoven, the Netherlands, as a senior consultant and parttime professor in Information and Communication Systems at the University of Technology Eindhoven, The Netherlands. Regularly, he is a teacher of the Philips Technical Training Centre and for other post-academic courses. He has written and contributed to numerous international publications and conference proceedings and he holds a list of US patents. In 1995 and 1999, he co-authored papers that received the IEEE CES Transactions Paper Award. In 1996, he obtained a company Invention Award. In 1997, Philips received the ITVA Award for its contributions to the DV standard. Mr. de With is a senior member of the IEEE, program committee member of the IEEE CES and board member of the Benelux working group for Information and Communication Theory.