Function Test Framework for Testing IO-Blocks in a Model-Based Rapid Prototyping Development Environment for Embedded Control Applications

Stefan Pitzek[1] and Peter Puschner[1]

[1]Institut für Technische Informatik,
Technische Universität Wien, Vienna, Austria
{pitzek,peter}@vmars.tuwien.ac.at

Abstract — *Testing and verification are important methods for gaining confidence in the reliability of a software product. Keeping this confidence up is especially difficult for software that has to follow fast changing development cycles or that is targeted at many platforms.*
In this paper we present a test framework for creating and executing function (black-box) tests of I/O control blocks, which are part of a model-based rapid-prototyping development environment for distributed embedded control applications.
The framework supports test engineers by trying to minimize their required effort for specification, setup, and execution of tests. This is achieved by defining test environment specifications that decouple the test specification from many properties of the test environment. The same mechanisms are also used to improve the test-cycle turn-around time for large test suites by supporting the automated parallelization of tests for efficiently distributing them over the available test hardware.

1 Introduction

One important problem when developing software products is to get confidence in their reliability and adherence to their intended specification. This problem is amplified for software that must follow fast development cycles and/or must support many different target platforms. The two main approaches for improving the confidence in the reliability of a system are formal verification and testing. In this paper we present a test framework for creating and executing function (black-box) tests for a software application that must deal with these problems.

The targeted implementation under test is an I/O control block set that is *part of* a model-based rapid prototyping (RP) development environment for the design of distributed embedded control applications. The development environment itself is based on *Matlab/Simulink* and includes full support for the generation of executable embedded applications for multiple different target architectures. The I/O block set consists of several blocks, each representing particular I/O-functionality on the corresponding special RP-

hardware boards. These blocks can be included in application models of embedded control applications, where they represent the driver for the respective physical I/O-element on the target hardware board.

For the proposed test framework we have identified two central requirements. The first one is to support test engineers during the specification, generation and execution of tests and test scenarios, since without such support testing of the many targeted hardware architectures and specialized RP blocks and corresponding hardware boards is not possible with justifiable effort.

Second, care must be taken to keep the test-cycle times low. This requires that block tests must be intelligently parallelized and distributed over the available physical test setup, in order to fully use the available testing resources.

In the proposed test framework (TF) we address both of these issues by introducing test environment and test specifications that try to decouple tests from the test environment.

Unlike some other approaches for the automated derivation and generation of tests, we make no particular coverage assumptions. Instead we focus on the generic structure of the test framework i. e., we define a framework that supports the efficient definition of a wide range of tests, but the quality and actual coverage of actual test campaigns still lie in the responsibility of testing engineers.

The paper is structured as follows. In section 2 we present related work. Section 3 gives an overview on the parts of the rapid prototyping kit. Section 4 presents the test framework model in greater detail, while section 5 presents implementation experiences from prototype implementations of some of the proposed ideas. Section 6 discusses the experiences gained throughout this work and section 7 summarizes and concludes this paper.

2 Related Work

Much work on testing frameworks and automatic test derivation has been performed in the protocol conformance testing community. These approaches mostly discuss models for generating tests that have a guaranteed full coverage according to a particular specification. Petrenko et al. present a formal framework for automatically deriving tests from formal specifications based on finite state models [1]. Bernot et al. construct *practicable test sets* from algebraic specifications in [2]. Bousquet et al. present *Lutess*, a testing environment for reactive synchronous systems in [3]. Algar et al. propose a framework for testing real-time reactive systems based on object-oriented formal specifications [4]. Jagadeesan et al. base their testing on the specification of safety properties defined with temporal logic [5].

Other work on testing distributed systems has been performed by Schütz, who proposes an approach for testing distributed real-time applications based on the MARS-architecture in [6], and Thane, who presents a systematic approach for testing distributed real-time systems in [7].

The concepts proposed in this paper should not be confused with general approaches for testing real-time systems. We are more concerned with classical function (black box) testing of software components [8], whereas part of these components are hardware drivers in embedded nodes.

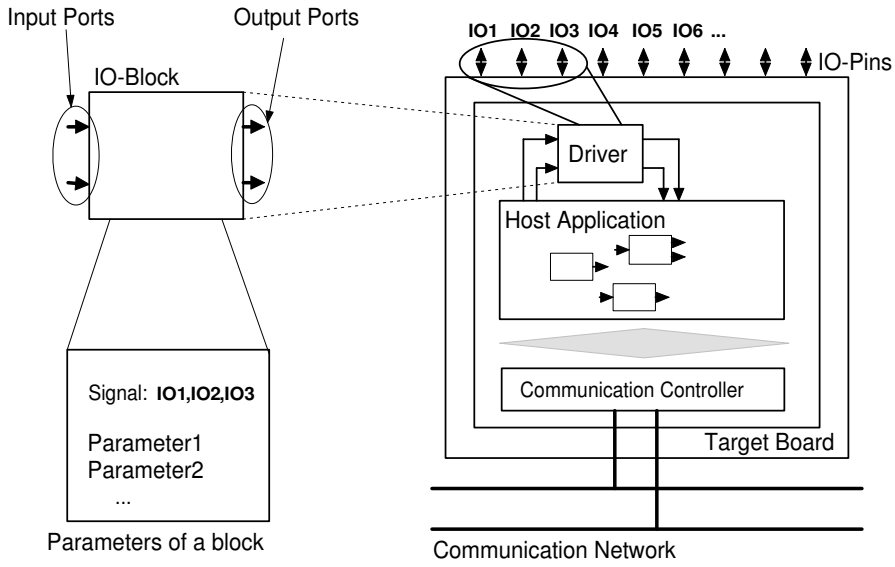

Figure 1: Functional structure of the Rapid Prototyping Kit

3 Rapid Prototyping Kit

The Rapid Prototyping Kit (RPK) consists of software, as well as hardware components. There are two block sets for Matlab/Simulink; one used for modelling the host application and simulating the communication behavior of the network protocol, and a second (I/O) block set for representing the control interface to physical I/O-interfaces. The block sets, Matlab/Simulink and additional software tools required for creating the embedded application code run on a host computer (usually a workstation or PC) that is connected via Ethernet to a test network containing at least one hardware target board. A *hardware target board* consists of a network controller and a multi-purpose I/O-controller, which is customized to cover a wide range of control functions for different application domains (e. g., x-by-wire applications, sensor applications).

Both block sets support the generation of embedded-application code for different hardware target boards, by using the existing tool chain for communication and application schedule generation and the code-generator provided by Mathlab/Simulink. Figure 1 gives an overview on the structure of the RPK. Each *IO-block* represents a corresponding *driver* on the *target board* and is usually configurable to control one of some functionally equivalent *I/O-pins*, selected via a *signal* parameter (e. g., a digital out block can be used to control any of 16 available multi-purpose I/O-pins). In the following we will also refer to the physical I/O-pins as *signal ports*. The block interacts with the host application over the *input and output ports*. There can be blocks which do neither have *ports* nor control pins. Such blocks usually are configuration blocks that must be present in the application to perform required setup operations (e. g., initialization of a communication controller). The *parameters* of a block support the configuration of the corresponding driver. The *host application* is the actual fieldbus application that uses the driver and is built as a Simulink model. The *communication network* in the RPK is the time-triggered protocol TTP/C [9], which is intended for highly dependable applications in the automotive and aerospace in-

dustries. TTP/C provides some properties that prove to be very beneficial for the creation of the test framework:

- Highly deterministic communication.
- Availability of a global time base, which leads to exact global synchronization of the nodes in the network.

In the proposed test framework, the function test applications for testing the blocks of the I/O-block set will be created using the infrastructure of the RPK. Thus the block-under-test will be part of the test application. The I/O-block function test only deals with a subset of required tests for testing the RPK. The actual RPK testing environment integrates and evaluates several testing strategies in an *outer testing framework*. This outer framework uses conventional automation techniques for executing and evaluating the different sub-tests.

4 Function Test Framework

Main task of the test framework (TF) is to provide the infrastructure for testing the I/O blocks and thus the drivers of the RPK. While automation is an important target of the framework, test engineers still play the most important role in the development of test campaigns. In order to best support them during this process we must identify their central tasks, which are:

Specification of test scenarios: The most important function of the test engineer is to conceive testing strategies for finding bugs in an implementation under test (IUT). Such strategies can range from intuitive ad hoc tests to whole test suites that systematically test certain aspects of an IUT.

Creation of test drivers for executing tests: In order to apply the developed testing strategies, the tester must somehow execute such tests. While often the manual execution of a test will suffice, this becomes more difficult when dealing with large test suites or when testing embedded applications that often can only be tested in their actual execution environment, where it is often quite difficult to observe a system's behavior (due to lack of adequate debugging interfaces and speed of execution).

Evaluation of tests: For successful evaluation of tests a tester depends on complete and detailed logging information on the executed tests.

Besides supporting test engineers the *efficient use of the available test resources* is another central requirement. Following are some reasons, why these requirements are important:

- Due to the many different blocks, parameters of blocks and supported target boards, exhaustive manual testing is hardly achievable with justifiable effort.
- Many RP hardware boards are structurally similar (e. g., they use the same I/O-controller), but differ in their detailed actual I/O layout (i. e., which of the I/O capabilities of an I/O-controller are actually used in a particular implementation), that is, for each target board a more or less different test environment has to be built.
- Fast development cycles and changing hardware platforms require a regular execution, re-evaluation and modification of tests.

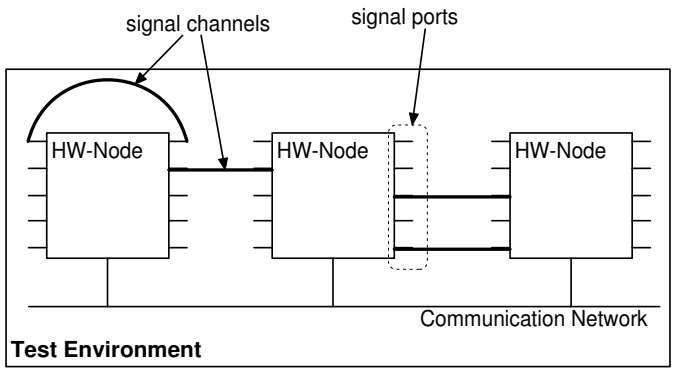

Figure 2: Overview on the test environment

– In order to test the hardware drivers, at least parts of the tests must be executed as embedded application on the host nodes. The generation of these applications and the application download process takes much longer than the execution of the actual test runs in the cluster. Thus, there should be as few test applications as possible to cover a test specification in order to keep the duration of the test cycle short.

In the following sections we will show how these requirements can be reached.

4.1 Architectural Structure of the Test Framework

The basic structure of a black box test consists of a *component under test* and the *test environment* for execution of the tests. In our case the *components under test* are the *blocks and drivers from an I/O block set*. We first take a look at the high-level structure of the physical test environment (see Figure 2). The *test environment* consists of a network of *hardware nodes* controlling *signal ports*. If two signal ports are connected these ports define a *signal channel*. As we will see later, a signal channel is a likely place for execution of a block test. Figure 1 shows the relation between signal ports, drivers and blocks in the test environment. In order to perform an I/O block test in this environment, we require a test application that executes a test in the network.

4.2 Generic Block Function Test Bench

At the core of a test application is the generic block function test bench (see Figure 3) that defines the test environment for individual block tests. The block-under-test interacts with its environment over four different interfaces:

Configuration Interface The configuration interface allows the modification of the state of the block under test.

Block Input Interface The block input interface allows the provision of input parameters to a node. It is placed between the host application and the block-under-test.

Block Output Interface The block output interface delivers the outputs from a block-under-test. It feeds the outputs of a block back into the host application.

Signal interface The signal interface represents physical connections between multiple instances of the function test bench.

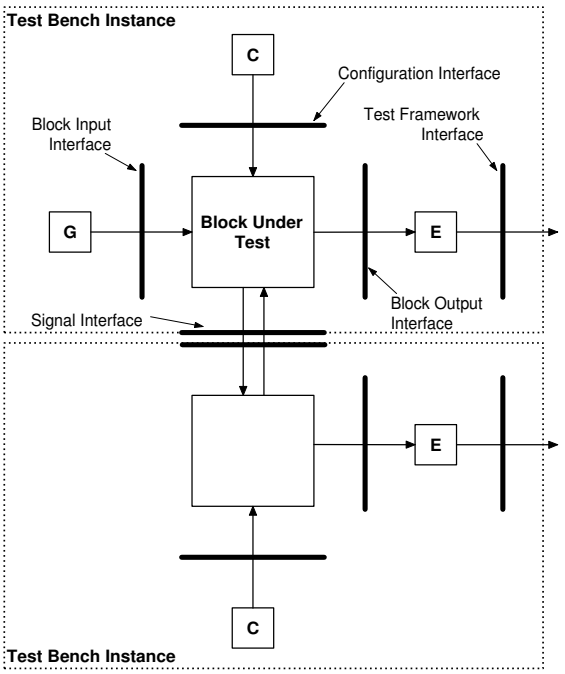

Figure 3: Generic block function test bench

The *configuration* and *block interfaces* are part of the same *test bench instance*, whereas the *signal interface* opens a *signal channel* to other instances of the test bench.

The labelled blocks represent the sources for the data that is provided over the respective interfaces:

G – Block Inputs: This block represents functions that provide input values for the block-under-test.

E – Test Evaluation: This block receives the block outputs and evaluates the test. The result is provided to the outer test framework (see Section 3) over the *test framework interface*.

C – Configuration: A configuration defines the state of the block-under-test. Usually the configuration will not change during execution of an individual block test.

The proposed embedded test bench supports the testing of arbitrary blocks. This is an important property, since blocks can greatly differ in how they interact with their environment (there are blocks that have no input interfaces, no output interfaces, no signal interfaces, or any combination of the above). As the term *test bench instance* implies, we will usually deal with multiple such instances, i. e., the test bench represents the smallest test scheduling unit within the test environment. We make no assumptions on where such instances have to run in the environment. The association to the physical test environment is only specified with the signal interface.

After this introduction of the two architectural core components of the framework we introduce the mechanisms we propose for solving the original requirements.

An important prerequisite for executing repeatable and accurate tests is the specification of a stable and well-defined test environment. For that reason we introduce the test

environment specification.

4.3 Test Environment Specification

The central role of the test environment specification (TES) is to hold all the information required for successfully setting up the test environment. This information is available for all other parts of the TF (e.g., automation tools, test case generation tools) and can also be shared between multiple tests. The specified information includes descriptions on the:

Available test hardware: The hardware specification represents information on the nodes in a test environment. It specifies the available physical signal-ports of a node that could be used for establishing signal channels. On the actual hardware board, a signal port typically corresponds to a physical pin.

Mapping of Signal Ports: The signal mapping explicitly defines the layout of the test environment by specifying which signal ports are connected and thus establishing the physical signal channels that are available for assigning tests.

I/O-block set under test: In addition to the physical properties of the test environment, we also provide specifications for the block set under test. Each block is specified by its name, sets of input, and output ports, and parameters. According to the model in Section 4.2 the in- and out-ports define the structure of the block input and block output interfaces, and will be used during the generation of an instance of the function test bench. It also provides descriptions for the parameters of a block. Each parameter is specified with a particular data type, whereas parameters that control signal ports have a special data type that allows to identify them as signal parameters.

Parts of the TES can be automatically extracted by analyzing the models of the I/O-blocks, while other parts have to be defined manually.

4.4 Test Specification

The role of a *test specification (TS)* is to provide test-specific information. The TES and a TS together must provide sufficient information to generate embedded test applications that can be executed in the specified test environment and that provide full coverage of the test specification. The TS specifies the following information:

Block configurations for blocks partaking in a test: These block configurations define the blocks at the ends of a *test channel*[1]. For testing I/O-functionality we always require a block that transmits a *test signal* and a corresponding block that receives this signal. In order to test blocks that do not use I/O pins or that only use one direction of communication, the respective block configurations must only be provided on one side of the test channel.

Test Function: The test function leads to the creation of several related instances of the function test bench for the same block under test. It specifies how the parameters of a block must be modified in order to create multiple related block tests.

Test Signal Generation Functions: The test generation functions provide the input values for the input ports of the block under test and thus define the structure of the port input interface.

[1] A test channel is a *logical connection* between the sending and receiving blocks of a test

Test Evaluation Functions: The evaluation functions receive the values transmitted over the output port and perform evaluation actions that yield information on failure or success of a test.

Note that we do not demand that these functions must be executed as part of the embedded application. For example, an evaluation function could be composed of two functions; one running as part of the embedded application, e. g., simply passing through the received values and another one running on the host, yielding the actual test results.

4.5 Generating Test Applications

In order to create an instance of the function test bench we must fully specify the four interfaces of a block under test. This information is distributed over the TES and the TS and we must define a way how the required information from both sources can be combined. Since we are still at the level of individual block tests, we only have to take the block configuration in the TS and "merge" it with the block specification from the TES. The matching can be easily done by searching for the block specification that corresponds to the block specified in the TS (e. g., by using the name of the block as a label). The resulting *merged block configuration* consists of the data from both specifications, whereas data provided in the block configuration overwrites the equivalent data in the block specification (e. g., a value for a parameter in the block configuration overwrites the corresponding default value in the block specification). The four interfaces of the block test bench are served as follows:

- The *merged block configuration* fully specifies the configuration of a block (since it contains values for all parameters of a block).
- For each input port there must be a an according test signal generation function in the TS.
- For each output port there must be an according test evaluation function in the TS that receives its values and provides the test result.
- The dedicated signal-parameters (either taken from the block specification or provided in the TS) define the physical association to the test hardware.

Next we will discuss how the proposed framework supports the original requirements.

4.6 Support for the Test Engineer

The distribution of the information required for executing a test over the TES and TS leads to simpler test specifications, since the test framework itself can collect much of the required information for creating the test application from the TES.

An additional side-effect of this approach is the possibility to perform basic validity checks on the specified tests. For example, if the numbers of test signal generation functions and test evaluation functions do not match the number of respective ports of the block-under-test, the test specification can be recognized as being inconsistent. The same is true if the values provided for a parameter in the block configuration do not adhere to the specified data type for this parameter.

By specifying a test function the test engineer can directly generate multiple tests from one test specification, whereas the framework directly creates optimized executable test

applications from these specifications.

Due to the definition of generic interfaces most parts of the framework can be exchanged or reused

4.7 Automatic Distribution of Tests

Up to this point, we have mostly looked at individual block tests. Since the test framework shall be capable of intelligently assigning tests to the available signal channels, we introduce additional mechanisms, which must address these two important issues:

1. How to assign block tests to particular physical signal ports (pins)[2].
2. How to optimize the function test application to efficiently cover multiple tests.

In the TF we propose the following approach. From a block under test get the data type of the signal parameter (which specifies the supported signal ports) and search the available signals in the TES for compatible signal ports. Then check wether there are I/O-mappings that use signal ports of this particular signal type. If this is the case, check if the second test bench instance specifies a block that can be mapped to the other end of the signal channel. In short the TF tries to map the specified test channels to the available signal channels.

According to this mapping of tests to signal channels, the number of tests and the layout of the test environment, several strategies for optimizing the generated function test application can be chosen:

- Schedule related block tests as multiple local test bench instances over multiple available compatible pins on a node for parallel execution of these tests. If there are more tests than available pins, schedule the tests in consecutive local test rounds.
- Schedule tests that use non-overlapping signal ports in parallel on the same node.
- Schedule multiple subsystem-based tests by starting them in succession with multiple global triggers.

Since the underlying network provides an exactly synchronized global clock the specification of exactly timed sequential and parallel test runs is straightforward.

5 Implementation Experiences

For the implementation of the presented concepts we use a variety of different mechanisms. The test environment and test specifications formats have been defined with XML [10]. This introduces an additional processing layer during test generation and execution, but decouples the framework from simple changes in the model specification formats that are used to automatically extract the block specifications. This leads to an improved stability of the test descriptions [11]. Additionally, XML-based formats are easy to process and extend. For the test environment specifications we want to extract as much information as possible from the library model of the block set. Currently this information includes in- and out-ports for all blocks, and specifications for all parameters, describing their structure (as generated XML Schemas) and default values. Where these values are not available from the analysis of the model (e. g., specification of ranges for

[2]This question must also be answered when designing a test for a single block

numeric data types), we want to find ways how the library models could be extended with minimal impact to make this information directly extractable from the models. While the block specifications can be nearly completely generated, the hardware specifications must be specified manually. The same holds true for the I/O-mapping.

Currently most test specifications are created manually. This does not contradict our automatic test generation requirement, since on the one hand a test specification usually defines several actual tests in the embedded systems, and on the other hand the format is simple enough to be generated. For example, one could easily create sets of test specifications testing the blocks from a stable version of the block specification with different configurations.

In the case study we only support constants and arrays (that provide their values according to a local test step selection component) as possible functions for generating signals for the input ports and evaluating the values received over output ports. In general we find it easier to evaluate tests at a higher framework level (e. g., on a personal computer) than to create complex embedded evaluation functions.

Function test applications are created from few simple template application models that represent the various parts of the test (block under test, test evaluation function, trigger functions) which are initialized according to the information from the TES and the TS. While the structure of the function test application is very simple, it also is easily extendable, since the inner interfaces are very narrow, and the test-relevant functions are strictly separated from each other and the framework. This structural simplicity also greatly eases the generation of the function test applications models.

The most important reason to generate the application models instead of directly generating the embedded code is that in the former case, the existing protocol scheduling and code generation facilities provided by the RPK can be used.

Because of the deterministic nature of TTP/C tests can be timed very exactly. Test results (the outputs from the evaluation functions) are transmitted over the TTP/C bus and received via a monitoring tool that traces the bus traffic. Due to the deterministic timing behavior of the protocol, mapping the test results to the test specifications is straightforward.

At this stage of development the test framework consists of a set of scripts implementing multiple parts of the targeted environment. Most of these parts are in early stages of development and are also not integrated in the “outer” test-framework, which shall later integrate these function tests in an encompassing unified regression test framework.

6 Discussion

In the following we discuss some capabilities, related design decisions, and problems of the framework:

- The modular structure of the proposed test environment eases the exchange and reuse of most parts of the test specification (such as test and evaluation functions, test specifications, block specifications), thus making the framework highly configurable for different tests.
- The use of XML for defining the test specification format eases processing and management of tests and supports the early discovery of structural errors and inconsis-

tencies in a test specification.

– By automatically creating the models of the test applications the existing facilities for creating the embedded target code be used.

One of the most difficult problems for the actual implementation of the framework is the problem of mapping the signals used in the abstract model to the signals in the different versions of the I/O-Toolbox, since these signals are often named differently. This mapping is also important to automatically match both subsystems at the ends of a signal-port test channel. Here we make the assumption that only compatible blocks are physically connected in the test environment. If this mapping of block signal ports to physical signal channels is not automatically possible (e. g., due to inconsistencies in the blocks under test), the framework also supports manually providing a list of compatible signal ports for the scheduling of tests.

Some special blocks (such as configuration blocks) do neither provide block input, nor block output or signal interfaces. These blocks can only be tested implicitly by testing blocks that depend on their presence in an application.

A wide range of tests should be coverable by changing the function scripts for the test specifications. Of course, the test engineer still carries the responsibility for creating practical test suites.

7 Conclusion

We propose a respective testing framework for creating and executing function tests for the blocks and according hardware drivers of an I/O block set of a model-based rapid prototyping development environment for embedded control applications. The framework supports test engineers in the specification and execution of such function test by using generic formats for specifying the test environment and tests that supports the decoupling of tests from the test environment. The framework also supports the generation of executable embedded function test applications from these specifications. These applications are built from one or multiple instances of a generic function test bench, which then are intelligently assigned to the available test resources in a particular test setup. Parts of the required specifications can be be directly extracted from the model of the system under test.

While the actual intended application is quite narrow, we tried to create a very generic underlying test model, which should be adaptable for creating embedded function test applications to test additional properties of the system-under-test. Parts of this model have been implemented in a case study.

In the future we want to create implementations of further concepts of the TF, and examine whether the framework can fulfill its expectation.

Acknowledgments

We want to thank our colleagues Raimund Kirner and Wilfried Elmenreich for their insightful comments on earlier versions of this document. This work was supported by the FIT-IT project “Rapid Prototyping Kit” under contract No 806033. FIT-IT is funded by the Austrian ministry for transport, innovation and technology (BM-VIT).

References

[1] A. Petrenko, N. Yevtushenko, G. v. Bochmann, and R. Dssouli. Testing in context: framework and test derivation. In *Computer Communications Journal, Special issue on Protocol engineering*, volume 19, pages 1236–1249, Nov 1996.

[2] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. In *Software Engineering Journal*, volume 6, pages 387–405, Nov 1991.

[3] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanom. Specification-based testing of synchronous software. In *Proceedings of the FMICS'2000: 5th International Workshop on Formal Methods for Industrial Critical Systems*, April 2000.

[4] V.S. Alagar, O. Ormandjieva, and M. Zheng. Specification-based testing for real-time reactive systems. In *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems - TOOLS 34*, pages 25–36, 2000.

[5] L. Jagadeesan, A. Porter, C. Puchol, J.C. Ramming, and L. Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the 19th International Conference on Software Engineering*, May 1997.

[6] W. Schütz. *The Testability of Distributed Real-Time Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1992.

[7] H. Thane and H. Hansson. Towards systematic testing of distributed real-time systems. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 360–370, Dec 1999.

[8] B. Beizer. *Software Testing Techniques. Second Edition*. Van Nostrand Reinhold, New York, 1990.

[9] H. Kopetz. *Specification of the TTP/C Protocol*. TTTech, Schönbrunner Straße 7, A-1040 Vienna, Austria, July 1999. Available at http://www.ttpforum.org.

[10] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000. Available at http://www.w3.org.

[11] C. Liu. Platform-independent and tool-neutral test descriptions for automated software testing. In *Proceedings of the 2000 International Conference on Software Engineering.*, pages 713–715, 2000.