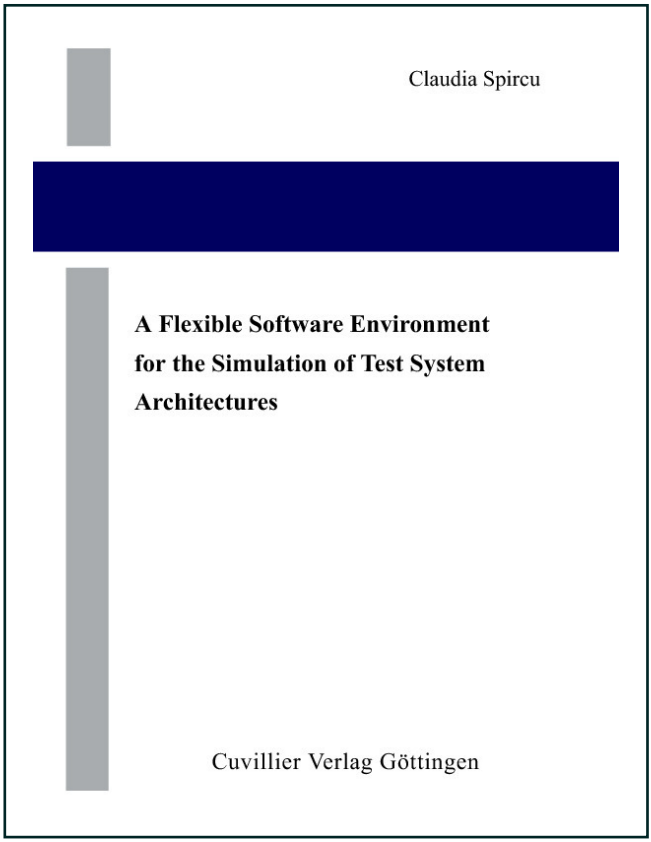
Claudia Spircu (Autor)

A Flexible Software Environment for the Simulation of Test System Architectures



https://cuvillier.de/de/shop/publications/2202

Chapter 1

Introduction

1.1 General Motivation

This work originated in the frame of the *Monarch* BMBF Project in co-operation with two companies: *SZ Testsysteme* and *ATMEL*. As the title of the project suggests, *Monarch* - "*Development and Modeling of a New ATE Architecture*" (in German "*Modellierung einer neuen Testerarchitecktur*)", had as a goal the development of new concepts for a new *Automatic Test Equipment* (ATE) system architecture.

A mixed signal ATE system used for testing integrated circuits with digital and analog parts contains heterogeneous components, such as: controllable test instruments (signal generators, signal capturers and processors), bus systems, current and voltage sources and meters, loadboard (adaptor between the test system and the device under test - DUT), operating system device drivers etc (see Figure 1.1).

Figure 1.1 An ATE system

The reason for this project was that existent (at that time) ATE architectures were old and weren't developed on an evolutionary basis. Therefore, the communication between different components of an ATE system showed often bottlenecks and the execution of a complete test program wasn't always finished in an optimum time. During the production test of integrated circuits when thousands of circuits were to be tested as quickly as possible, this was not to be desired.

Because of the number and complexity of the instruments and of the communication system, the development and, much more, the evaluation of a new ATE architecture cannot be done without previously modeling and massive simulation.

In the specifications of the *Monarch* Project was written that: "the new architecture should be developed with modern simulation techniques [...]". Therefore it is necessary to model

ATE instruments to analyse interaction of single modules. The simulation modules have to be flexible and customizable for evaluation and fine tuning".

Another specification said: "In between the next five years industry will demand for new modular test systems to meet future requirements. In this project new test system architectures will be composed, modeled and verified with a wide range of simulation procedures".

A final simulation run with high resolution for such a complex system is not possible, even with modern simulators and computers. Therefore, when modeling ATE architectures, several abstraction levels must be taken into consideration. Using only detailed electrical signal descriptions it is not possible to evaluate a complete architecture, that contains hundreds of instruments and several bus systems. A complete architecture can be evaluated only if the models are less refined, abstracted from the electrical level to a behavioral one, using instead of signals the appropriate data. Therefore several abstraction levels in the modeling of the system must be provided, dependent on the modeling goal: low level when evaluating individual components or high level for the entire architecture. Several views of the system must also be provided, at least a structural and a behavioral one.

1.2 ATE Modeling – Abstraction Levels and Views

As it is defined by J. Peterson, in [55], a model is a representation of what is felt to be the important features of the object or system studied. By the manipulation of the representation, it is hoped that new knowledge about the modeled phenomenon can be obtained without the cost or inconvenience of manipulating the real phenomenon itself. Most usual a model is a formal description (in a mathematical sense) of a system or parts of it.

An important characteristic of a model is that it abstracts the system, sometimes on several levels of detail. *Abstractioning* means representing only properties of the system that are important to the investigated problem. Through *model refinement* several properties can be added to the model.

On the other hand, a model shows only a *perspective* or *view* of a system. Abstraction levels are often orthogonal to views.

For modeling the hardware and the software components of a test system, several abstraction levels and views are used.

As views, one can consider the *behavioral*, the *structural* and the *physical* view, whereas the physical view is no concern of this work. For abstraction levels, two abstraction levels are important, namely the *high level* and the *low level*.

The abstraction levels and views can be represented in a so-called Y-chart, introduced by D. Gajski (see [25]), as shown in Figure 1.2.

The three axes represent the system perspective, on each axes the ticks represent the abstraction levels: the farthest away from the center they are in the Y-chart the more abstract.

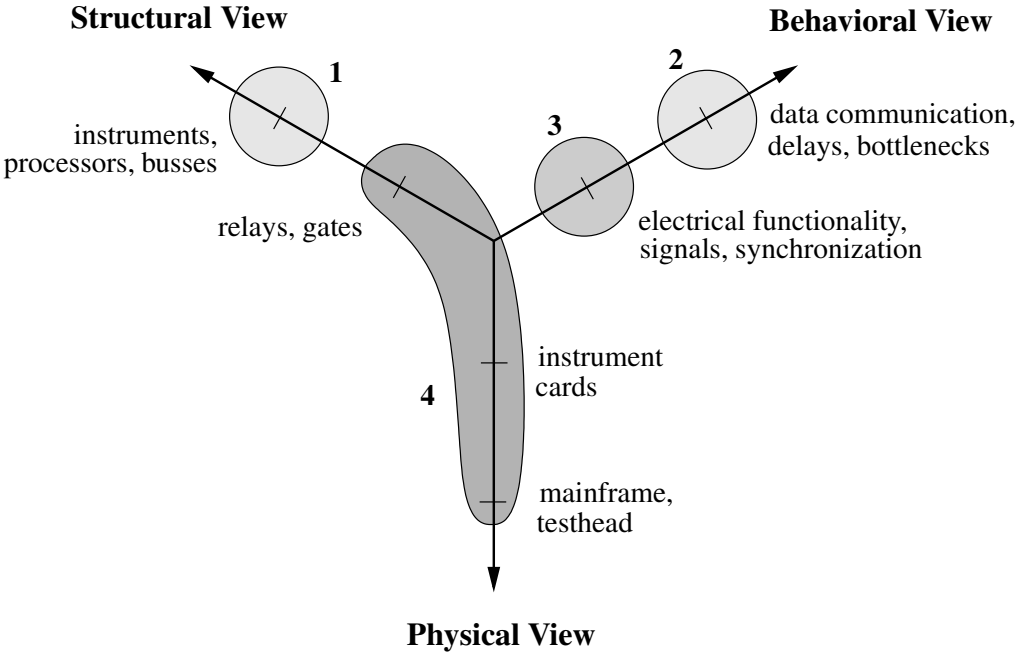
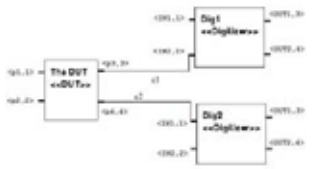
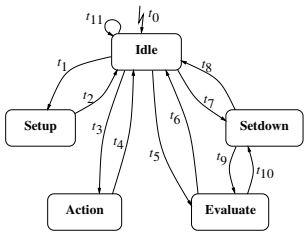
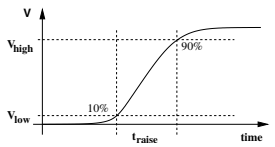


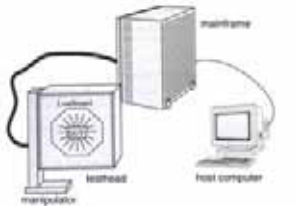
Figure 1.2 Views and abstraction levels - the Y-chart

1.3 The Simulation Levels Used

As combinations of abstraction levels on different system perspectives, four simulation levels are introduced in this work. They were shown in Figure 1.2 and will be explained in the table below.

Sim. Level	Description	Representation
1	It is a structural high level description of the system. On this level, the ATE is a collection of instruments, communication components, devices under test (DUTs) and host computer(s). **Problems**: – to describe the components using adequate properties, – if possible, to use concepts from the object-oriented design, mainly encapsulation and inheritance, – to describe the component ports (interconnection points with other components). **Simulation** on this level is a simple interconnection check: open connections, all connections allowed etc.	

2	It is the behavioral high level description of the system. Here behavioral models (in form of statecharts) are introduced for the hardware components and task graphs are used as test program models. They control the hardware models. Communication between components is seen as data communication. **Problems**: – to define models that describe the functionality of the components simply enough for allowing the simulation of the entire architecture but complex enough to contain all the interesting parts of the problem, – to define test program models, – to connect all the models and simulate them. **Simulation** on this level is a discrete-event simulation, because the models can be defined in form of extended state machines (statecharts) that react at events. The simulator must use these high level models, but also permit, if necessary, the connection of lower level models, via the concept of ”observers” (see [27]). The high level models can be written in any general programming language: C, C++ etc.	
3	It is a behavioral low level (electrical) description of the system. The behavioral models that are used here must take into consideration the electrical properties and transformation of the components, must decide the signal characteristics and constraints. **Problems**: – to define models by using mathematical equations and also finite state machines, – synchronization problems of the used signals. **Simulation** on this level is a mixed-signal analog and digital one. Therefore the models on this level must be described with special hardware programming languages, like for instance `VHDL-AMS`.	

4	Includes low level structural modeling together with physical aspects of the system. **Problems**: – the internal structure of the instruments, – placement problems: were to put the instrument cards: in the main frame or in the testhead (see Section 3.1). The actual modeling on this level is no concern of this work.	

The main concern of this work is the high level structural and behavioral modeling, i.e. the simulation levels 1 and 2, where the following simulation goals can be attained:

– getting a coarse evaluation of the execution time of a test program on a given architecture, before the actual construction of the test system;

– obtaining the component load and therefore having an overview of the components that are mostly used or the components that are not at all used; thus getting the bottlenecks in the data communication.

In order to evaluate an architecture, key test cases for different measuring requirements must be used: mixed-signal stimuli with complex wave forms, analysis of DUTs analog signals with modern digital signal processing methods. A test program and a DUT must thus be also taken into consideration when evaluating an architecture.

1.4 Used Hardware and Software Models

Orthogonal to the abstraction levels and views the models used for describing hardware and software can be, in general, classified as follows.

- ❑ control dominant (or state oriented)
- ❑ data flow dominant (or activity oriented)
- ❑ heterogeneous (control plus data oriented)
- ❑ time oriented
- ❑ structure oriented

Some of these models are based on the Petri net model (see Appendix A). A very good introduction to Petri nets and the modeling of systems using Petri nets is given by Peterson in [55].

Examples of *control dominant models* are the finite state machines (FSMs) and the statecharts, which are state machine extensions. They are normally used for modeling hardware behavior or control (reactive real time systems). The statecharts are used in this work for test system component behavioral modeling and therefore they will be explained in detail in Chapter 4. The finite state machines can be seen as special kinds of Petri net models and therefore are shown in Appendix A.

An example of *data flow models* are the data flow graphs. They are normally used for modeling activities or data dependencies. Throughout this work the activities will be called *tasks* and a particular acyclic kind of a data flow graph, called a *simple task graph*, will be used. Because simple task graphs are used for modeling test steps, they will be presented in Chapter 2, together with concrete tasks used when testing integrated circuits. The data flow graph are shown also in Appendix A, as being another special kind of Petri net models.

A special kind of models, which combine control and data flow, are the *heterogeneous* ones. As an example of these models the so-called *hierarchical task graphs* are shown. They are most adequate for modeling a test program seen as a set of tasks with dependencies between them together with control structures such as loops or branches.

By introducing *time marks* on a data flow or heterogeneous model, the model can be used for the performance analysis of the system. Such time oriented models are called *time marked graphs*. The simple and hierarchical task graphs used are actually such kind of models.

A *structure oriented* model is used for defining the structure of the physical composition of the system (modules and connections). Such a model will be used for representing the structural view of a test system, as it will be shown in Chapter 3.

1.5 Motivation for the Creation of a New Software Environment - TSCE

This work has two goals:

- to *define* high level models for the test program and the hardware components of a test system used when evaluating the system and
- to *implement* a completely new software environment for the manipulation of the introduced models and evaluating an architecture.

The new software environment must allow the communication between abstraction levels and also the simulation on the high level. The environment is called the "*Test System Component Modeling and Simulation Environment*" (TSCE) and is defined by the following concepts:

1. In order to model an architecture, individual components, including instruments and communication, must be modeled in a structured way, using properties which help in

evaluating the component characteristics. Simulation levels are connected to the component properties. Communication between simulation levels is allowed.

2. The encapsulation and inheritance concepts from object-oriented design are used to create a hierarchy of models. To use a model, it must be instantiated.

3. The software environment allows, in a graphical way, via a *Graphical User Interface*, the component definition and instantiation and the creation of connections, thus enabling the building of architectures.

4. A time evaluation of a constructed architecture takes into consideration the changes within component models, and thus behavioral models are defined and attached to the components.

5. The architecture evaluation cannot be done without a test program and thus test program modeling on a higher abstraction level is also realized.

6. A simulator maps the software models of the test program onto the hardware ones of the components. On the high abstraction level, used for component modeling, the use of a discrete-event simulator is sufficient. The output of the simulator provides the evaluation of the execution time and the component load.

In the frame of this work a discrete-event simulator was written for the following reasons:

– the simulator had to be controlled by the test program model, that generates the events in the system that are further processed by the models,

– the simulator needs to connect models on several (at least two) abstraction levels: the simulation and evaluation of a complete architecture is done based on high level models but the property values used on this level must be computed using low level models and the concept of ”observer”, as it is described by Gentner in [27],

– the simulator works with a hierarchy of models, using models that inherit properties and behavior from other models.

1.6 Overview of TSCE

The created software environment, TSCE, is developed with object-oriented concepts and the *Unified Modeling Language*. TSCE contains several packages:

types	–	contains the classes for defining component models with properties and port definitions, the component models being called device types. This module implements also the model hierarchy.
elements	–	contains the classes for defining instances of components, of ports and connections between port instances.

`graphic`	–	is the module that contains the graphic extensions of the models and the *Graphical User Interface*.
`model`	–	is the module for defining behavioral models of components in the form of statecharts (extended finite state machine models). **Remark**: The actions performed by the models are provided as C action functions contained in dynamic link libraries, therefore they can be compiled and modified separately from the `TSCE` main part.
`program`	–	is the module for defining test program models in the form of simple and hierarchical task graphs.
`simulator`	–	implements the self written discrete-event simulator and the event classes.
`xml`	–	is the module that contains the `XML` parsers for reading and saving component models, architectures and programs from `XML` files.

1.7 Structure of this Work

This work is structured in the following way.
Chapter 2 presents the simple and hierarchical task graphs and illustrates how, throughout this work, a test program is represented using these data structures. This representation allows the modeling of a test program as tasks to be performed, with dependencies and possible control nodes, and is used in this work to evaluate the duration of a test program.

The used structural component modeling is explained and presented in Chapter 3, Section 3.3. Here device types are introduced which, using generic properties and ports, are able to represent an arbitrary test system component. Inheritence concepts are applied when defining a device type, thus allowing the creation of a device type hierarchy, having as goal the inheritence of properties along the hierarchy.
The use of device types via their instances, called device elements, in order to create different test system architectures is presented also in Chapter 3, Section 3.4.

In Chapter 4 behavioral models for test system components are shown based on statecharts (finite state machine extensions). It is shown how component behavioral models can be defined using the software environment designed, each behavioral model being attached to a device type. Some model examples are introduced, used for general test system instruments, such as Digitizers, Generators and DSPs, or for a Bus communication component.

Chapter 5 presents the `TSCE` simulation environment and the discrete-event simulator programed together with simulation examples. Some task graphs are used and it is shown how the simulation takes place, controlled by a task graph and using the behavioral models introduced in the previous chapters.

Chapter 6 contains possible extensions of the introduced models in order to realize an automatic mapping onto existent test system resources.

Conclusion and further developments are presented in Chapter 7.

Chapter 2

Test Program Modeling

2.1 What is a Test Program?

2.1.1 Generalities

As defined by Miegler in [48], a *test program*, for the test of mixed analog-digital integrated circuits, is a textual description in the form of a program code plus signal description of all the test steps that must be performed by an Automated Test Equipment (`ATE`) for a Device Under Test (DUT).

A *test step* performs usually a particular measurement on a particular DUT and documents the result. The set of all the test steps, describing all the necessary measurements, is the test program.

Following the results of the test, a classification of the DUTs is performed, that is called *binning*. The ”good” DUTs are separated from the ”bad” ones. For one DUT this can be expressed by a `PASS`/`FAIL` variable.

The sequential arrangement of all the test steps to be performed on a DUT is called a *test sequence*.

The repetition of a test sequence for all the DUTs to be measured is called a *test loop*.

2.1.2 Construction of a Test Step

In the following section, the construction of an individual test step will be more closely presented. The description is given by Miegler in [48] and by Beyer in [6]. A test step consists of the following:

- *definition of the used signals*
  The digital and analog signals used must be defined in a specific form that is required