Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.

http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.

http://www.ub.tuwien.ac.at/eng

TU WIEN

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Systematic Testing of Analog-Mixed Signal Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Mario Heindl, BSc.
Matrikelnummer 1125055

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Mitwirkung: Univ.Lektor Dr. Dejan Nickovic

Wien, 15. Februar 2017

Mario Heindl	Radu Grosu

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Systematic Testing of Analog-Mixed Signal Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Mario Heindl, BSc.
Registration Number 1125055

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Assistance: Univ.Lektor Dr. Dejan Nickovic

Vienna, 15th February, 2017

Mario Heindl	Radu Grosu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Mario Heindl, BSc.
Wiener Strasse 23, 2120 Wolkersdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Februar 2017

Mario Heindl

Danksagung

An dieser Stelle möchte ich mich bei meinen Betreuern, Prof. Radu Grosu und Dr. Dejan Nickovic, für ihre Unterstützung bedanken. Sämtliche Diskussionen trugen dazu bei, diese Masterarbeit Schritt für Schritt zu verbessern. Dejans Rat zur Implementierung sowie seine Feedbacks hatten großen Anteil daran diese Arbeit zu vervollständigen. Ebenso war Radus Feedback ein wichtiger Teil, um die Arbeit in dieser Qualität fertigzustellen.

Ebenso danken möchte ich Dieter Haerle vom Kompetenzzentrum für Automobil- und Industrieelektronik für all die Zeit die er aufgebracht hat, um mich beim Erreichen der Ziele dieser Arbeit zu unterstützen. Ohne die gemeinsamen Diskussionen, Debugging Sessions und die Koordination wäre es nicht möglich gewesen die hier vorgestellten Lösungen zu implementieren. Weiters danke ich auch Stephen Sabanal von Infineon Technologies für seinen Support betreffend des ATE Tests, der mir beim Vertsändnis und der Modellierung sehr geholfen hat. Vielen Dank auch an Hermann Tauber von Infineon Technologies, ohne dessen Erweiterungen im bestehenden GPIF Fileformat es nicht möglich gewesen wäre diese Masterarbeit in der bestehenden Form fertigzustellen.

Mein Dank gilt auch meinen Eltern, Andrea Heindl und Wolfgang Loho, die mich bereits mein ganzes Leben lang unterstützen und die mir dieses Studium ermöglichten. Auch bin ich meiner Freundin Alexandra Leukauf dankbar, die mich all die Zeit unterstützt und motiviert hat.

Acknowledgements

I would like to express my gratitude to Prof. Radu Grosu and Dr. Dejan Nickovic for supervising me throughout this thesis. Discussions with them improved this thesis step by step. The practical support from Dejan and his reviews helped me quite a lot to finish this thesis. Also Radu's feedbacks were important to finish this thesis with this quality.

Furthermore I want to thank Dieter Haerle from the Kompetenzzentrum für Automobil- und Industrieelektronik (KAI) for all the discussions, debugging sessions and the coordination during the practical part of this thesis. He spent a lot of time in supervising me to achieve all the targets. I also want to thank Stephen Sabanal from Infineon Technologies for his support regarding the ATE Test which helped me quite a lot to understand and model the simulation environment. Many thanks also to Hermann Tauber from Infineon Technologies. Without his extensions in the existing GPIF file format, it would not have been possible to finish the practical part of this thesis in that kind of way.

Thanks also to my parents Andrea Heindl and Wolfgang Loho for supporting me my whole life. They enabled the possibility of studying for me. I am also grateful for the support of my girlfriend, Alexandra Leukauf, who supported and motivated me all the time.

Kurzfassung

Ein wichtiger Schritt im Design-Flow von Analog Mixed-Signal Systemen ist die Post-Silicon Verifikation. Sie dient dazu, um Random Manufacturing Defects zu erkennen und kostenintensive Field Returns vom Kunden zu verhindern. Diese Art von Test wird durch das Automatic Test Equipment (ATE) unterstützt. Ein ATE dient dazu, um analoge sowie digitale Stimuli anzuwenden sowie Messungen durchzuführen anhand derer verifiziert wird, ob das System die Spezifikationsanforderungen erfüllt.

Die heute am meisten verbreitete Testmethode zum Verifizieren von Analog Mixed-Signal Systemen ist Parameter Oriented Testing (POT). Die größte Schwäche dieser Methode liegt darin, dass keinerlei Aussage über die Testcoverage bezüglich Random Manufacturing Defects getroffen werden kann. Eine Alternative, um eine höhere Coverage zu erreichen ist Defect Oriented Testing (DOT). Dazu ist es notwendig, die aktuelle Coverage der existierenden Tests zu analysieren und zu messen.

Im Rahmen dieser Arbeit wurde ein Framwork zur Unterstützung der Analyse der Testcoverage entwickelt. Dazu wird ein beliebiges ATE Testprogramm übersetzt und in einer Simulationsumgebung ausgeführt. In dieser Simulationsumgebung ist es möglich Random Manufacturing Defects zu injezieren und die Effektivität der aktuellen Tests zu messen.

In einem ersten Schritt werden die Teile des ATE und die verwendeten Testinstrumente vorgestellt, bevor die zu modellierenden Funktionalitäten identifiziert werden. Aufbauend auf diesen Informationen gibt diese Arbeit einen Überblick darüber wie diese modelliert werden können. Zusätzlich wird eine Methode vorgestellt, um ein existierendes, allgemeines ATE Testprogramm automatisch in ein Format zu übersetzen, das von der Simulationsumgebung importiert und ausgeführt werden kann. Diese Ergebnisse ermöglichen es, ein ATE Testprogramm mit minimaler manueller Interaktion und einem großen Anteil an Wiederverwendung von vorhandenen Informationen im Rahmen einer Simulation auszuführen. Um Aussagen über die Simulationszeit und die Funktionalität der vorgestellten Methode treffen zu können, wurde der gesamte Prozess anhand eines existierenden Chips, zur Verfügung gestellt von Infineon Technologies, verifiziert.

Abstract

Post-silicon verification consists of testing and detecting random manufacturing defects just before its delivery to the customer. This is an important verification step that allows preventing costly field returns. This type of tests are executed by the *automatic test equipment* (ATE), which provides both ,analog and digital stimuli, and measurement units to check whether the design meets its specification requirements.

Nowadays, *parameter oriented testing* (POT) is the predominant testing method used by industry for the verification of analog and mixed-signal designs. The main drawback of POT is the relative lack of coverage information that it provides with respect to random manufacturing defects, e.g. opens and shorts caused by particles during the manufacturing process. More recently, *defect oriented testing* (DOT) has been proposed as an alternative testing method that can achieve higher test coverage than POT. In order to achieve this goal, it is essential to be able to analyze and measure the coverage of existing test suites.

In this thesis, we develop a framework that enables and facilitates the coverage analysis of ATE tests by allowing translation of arbitrary ATE test programs into a given simulation environment. The simulation environment allows fault injection that mimics manufacturing defects and thus can be used to assess the effectiveness of the existing ATE test programs.

We first introduce the ATE environment and its instruments. We then identify the ATE functionality that needs to be supported in the simulation environment, and we propose a methodology for modelling it. Furthermore, we develop a procedure for automatically translating general ATE test programs into a format, which can be imported into and executed from a simulation environment. The resulting outcomes enable applying ATE test programs during the simulation runs with a minimal manual interaction, thus considerably increasing automation and reuse in the testing effort. We implement the results and evaluate them on a real industrial chip, provided by Infineon Technologies to ensure the correct functionality of the translation and simulation process as well as for getting an idea about the needed simulation time for a whole ATE test.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation 1
1.2 Related Work 5
1.3 Contribution of this work 7

2 Background to automatic test equipment (ATE) test programs **11**
2.1 Requirements for an ATE test 12
2.2 General setup 12
2.3 Common test cases in industry 16

3 Setting up the simulation environment **21**
3.1 Modelling of an ATE test 22
3.2 ATE instrument model 26

4 Translation of an ATE test program **37**
4.1 Identification of needed functionalities 38
4.2 General test program language 46
4.3 Translation process 47
4.4 Code injection 48
4.5 Parsing of the intermediate file 50
4.6 Creating the limit checker 51
4.7 Example translation for the continuity test 55

5 Evaluation **61**
5.1 Setup 61
5.2 Simulation time 63
5.3 Fault detection 65

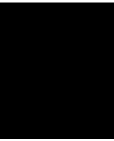
6 Conclusion and future work **69**

7 Appendix **73**

List of Figures **77**

List of Tables **78**

Bibliography **83**

CHAPTER 1 

Introduction

1.1 Motivation

Nowadays, the importance of analog mixed-signal (AMS) designs in system on chips (SoC) is constantly increasing, resulting in elaborated interactions between heterogeneous components and consequently a higher complexity of the designs. Since reliability and robustness have become more and more important for the customers, especially in safety critical applications, testing AMS designs turns out to be a non trivial challenge for the industry. From the industry's point of view there are certain requirements regarding testing methodologies: since testing time contributes significantly to the manufacturing costs, a major target is to keep this time as low as possible while keeping a high test coverage. Whereas in the domain of digital design well-known ATE methodologies, like scan-chains, are used, there is no such systematic approach for AMS designs.

Figure 1.2 gives an overview of a simplified AMS design flow, based on [BSED07]. As one can see, the design is divided into an analog as well as a digital part, which are then combined in the *mixed-signal design flow*. First of all, the mixed-signal specification is divided into an analog and a digital specification, such that both parts can be developed independently from each other. In the analog design flow, shown on the right hand side, the design engineers have to define several blocks, which are then combined for the complete analog design. In parallel, the test engineers define the test benches, used for the verification of the analog part. After a verification step, the model is calibrated, such that a precise model of the device is available in a simulation environment. When the verification was successful and the model is complete, physical development of the analog part starts. The design of the digital parts of an AMS design is similar: the digital components are designed and tested based on a verification suite. When the design is complete, physical manufacturing can start. Both parts are combined in the mixed-signal design flow. Based on the specification, a verification plan is developed, describing all the scenarios and test cases, which need to be verified.

The verification takes place at two different levels in this design flow which are highlighted in Figure 1.2. During the *Simulation Verification* the test benches are executed in a simulation run, for verifying that all functional requirements of the specification are met. Note, that in [RGA02] it is shown how simulation runs might influence time-to-market in a positive way (Figure 1.1). In contrast to the traditional approach, verification can be started earlier which yields in much lower time-to-market since the debugging time after the first silicon is reduced significantly.

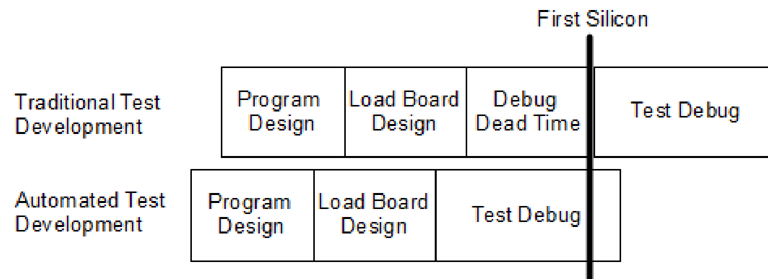


Figure 1.1: Time-to-market improvement by ATE

Caused by high simulation times, this is a common bottleneck in the design flow for AMS systems. For this purpose often simplifications are used to speed up the simulation runs. After all the test benches have been executed in the simulation and the functional verification was successful, manufacturing of the chip starts.

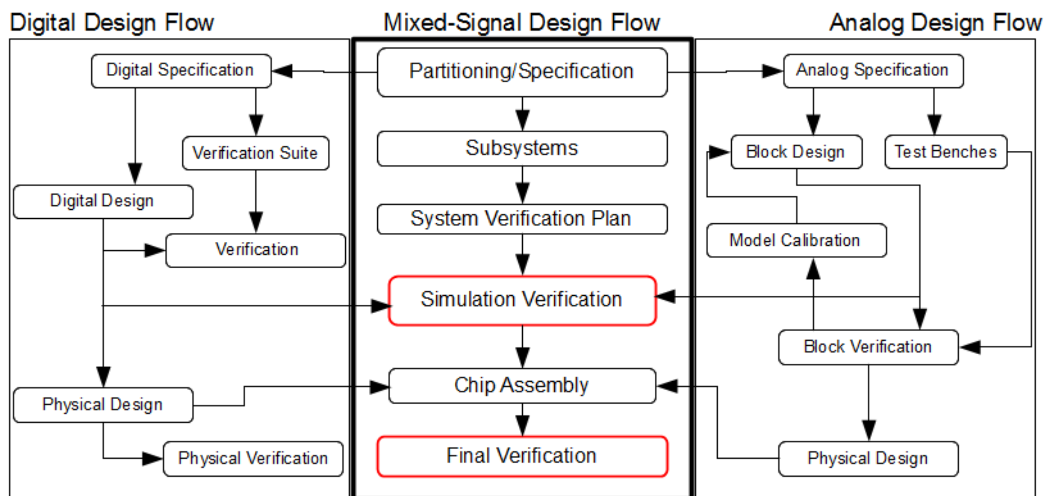
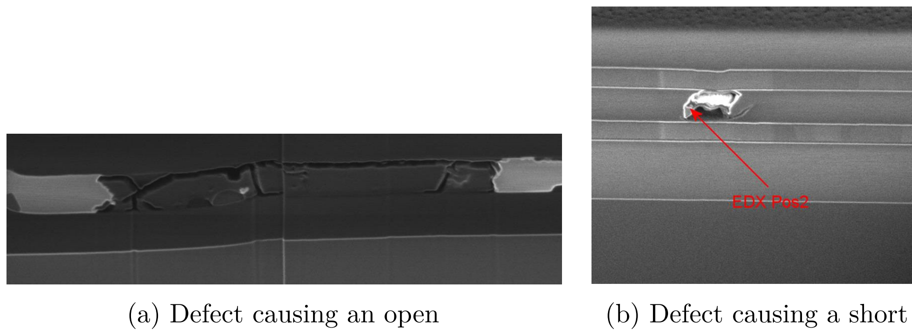


Figure 1.2: Analog Mixed-Signal Design Flow [BSED07]

Although the functional behaviour has been verified well during the simulation phase, it cannot be guaranteed that no faults will occur during the manufacturing process. The reason are the so called *random manufacturing defects*. This kind of defects is caused



(a) Defect causing an open (b) Defect causing a short

Figure 1.3: Random Manufacturing Defects

for instance by particles during the manufacturing process, yielding in an open or a short. Figure 1.3 visualizes these two kinds of defects. On the left hand side, one can see an open, where the wires are not connected any more, whereas on the right hand side, a short between the metal layers is shown - two layers, which are not supposed to be connected, are connected now.

For this *final verification* an ATE is used, which provides different test instruments, which are able to force a specific voltage, current as well as wave forms as inputs to the device under test (DUT). Furthermore, these instruments have the possibility of measuring analog signals and verify if their value lies within the user-specified limits. In Figure 1.4 a schematic of the setup of an ATE test is shown. The DUT is connected to the instruments of the ATE via the *load board*, which has the purpose of realizing dynamic connections. For instance, relays on the load board are used for connecting various test instruments to a special pin of the DUT during different tests. Apart from the hardware, the ATE consists of a test program, describing the test run (e.g. when relays shall be opened/closed, actual pin levels, current or voltage to apply). Moreover, measurements performed by the ATE hardware can be read and post-processed to check if the recorded values lie within specified limits or not. This is referred to as *parameter oriented testing* (POT).

The main problem of a POT test is the unknown test coverage. It can never be ensured, that all possible defects of a device are detected by the test. Therefore, whenever a *random manufacturing defect* occurs that is not covered by the POT test, this yields in a field return by the customer which is very critical in semiconductor industry. Currently, companies have a big interest to get a quantitative number concerning their test coverage in order to improve it. Other drawbacks of a POT test are:

1. **Test time:** This unit is a significant factor regarding manufacturing costs. Therefore it is necessary to keep test time as low as possible. With the current approach of testing, single test cases might be redundant and could be dropped out - there is a high potential for improving test times.

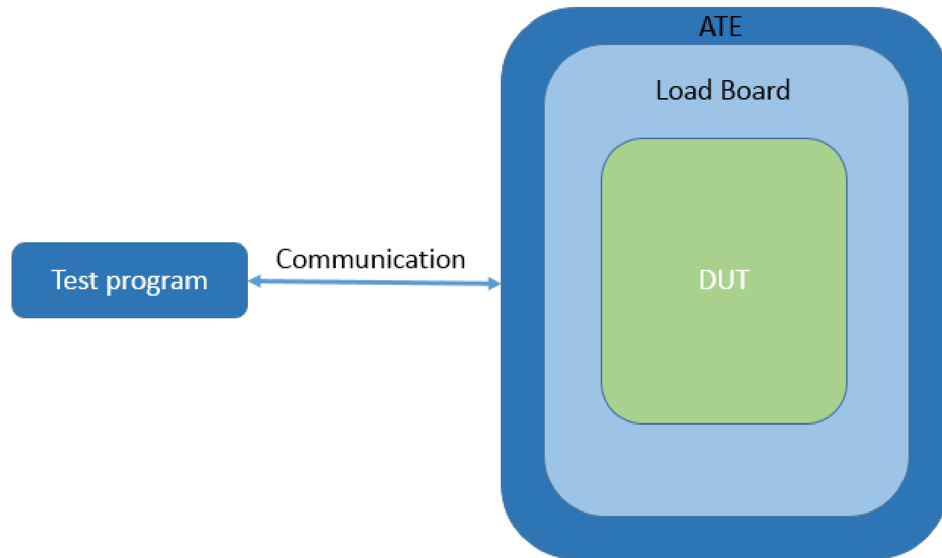


Figure 1.4: Schematic of the test environment

2. **Complexity:** The quality of the test run mainly depends on the experience of the test engineer. Designing test cases is a complex task caused by the number of interactions in AMS designs and the lack of insight into the different tests.

3. **Bottleneck in the design process:** Designing the load board and implementing the test program are time consuming tasks. Since the test engineers have to work with prototypes, development of the test program starts in a late phase of the design process since there is no possibility for testing the ATE test program earlier.

Caused by the lack of insight into the ATE test because of limited measurement resources, these challenges cannot be addressed at the stage of *Final Verification* in the design flow. It is necessary to go back one step to the *Mixed-Mode System Verification*. At this level, a testing approach which has already been proposed in the 90's in [SGOT98] or [KTH+11] can be used: the defect oriented test (DOT). Some industrial experience of DOT is given in [Xin98]. It has been shown that this kind of test indeed comes up with higher test coverage.

The idea of a DOT is to inject a specific fault into a model of the DUT and check with a simulation of the ATE test program, whether this fault would be detected or not. A schematic of the work flow for DOT is shown in Figure 1.5. On the left hand side the faults are identified and extracted such that they can be injected into the simulation model of the DUT while on the right hand side the ATE test patterns are executed within a simulation environment of the ATE. During the Analog Fault Simulation, it is checked whether an injected fault has been detected by the simulation or not. With the

advantages of getting an optimized test run and having a quantitative number of the test coverage (as has been shown in [Xin98]) this testing method has high potential for being used in industry, but still there are different challenges like high simulation times and the translation from an ATE test into some kind of format which can be understood by the simulation environment.

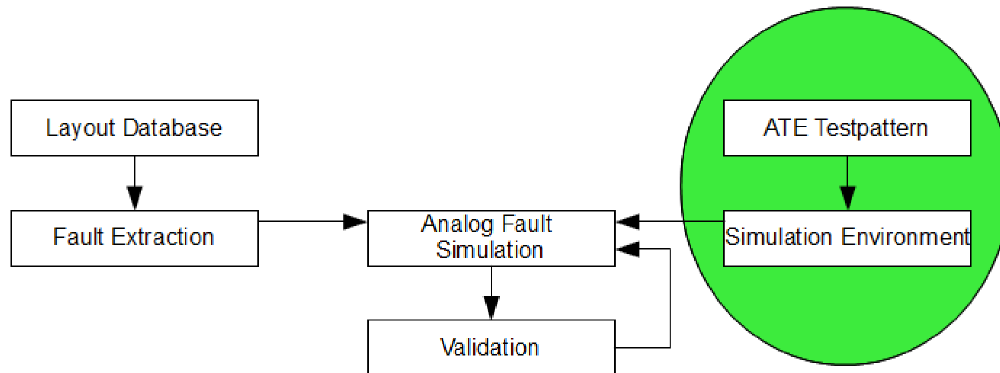


Figure 1.5: Flow diagram of DOT

This thesis focusses on the second of these challenges. We show, how to set up a methodology for translating the ATE test program in such a way that it can be simulated in a pre-silicon simulation environment.

1.2 Related Work

Since DOT testing is not a very new idea, there have already been several proposals of how to verify the correct behaviour of a device in the context of a simulation environment.

One of the big topics which is actually dealt with in research is *formal verification*. [ZT09] gives an overview about different approaches of *formal verification*, like *equivalence checking*, *model checking* or *deductive verification methods*.

Equivalence checking is based on two models, one of the specification and another one for the implementation of the AMS design. The question which has to be answered during the check is whether these models are equal with respect to some notion of equivalence or not. At this point, the approach of *equivalence checking* becomes challenging: how should equivalence between the models be defined? An example is given in [BHA95] where it is proposed to use the transfer function of the models to define their equality. Since the calculation of the transfer function in AMS designs is non trivial, this technique has been restricted to linear circuits in the content of their work. As one can see, *equivalence checking* is a complex approach, which needs a lot of computation time and effort, yielding in rare usage in industry.

Another approach which was dealt with is *deductive verification* [Fil11] based on inference rules. Tools were used to perform *equivalence checking* based on such rules for instance in

[GV99]. The problems occurring regarding *deductive verification* are quite similar to the challenges faced with *equivalence checking*: the models have to be linearized to reduce the complexity and the computation effort is much too high.

The third kind of *formal verification* method to be mentioned is *model checking* [CGP99]. Here, starting from a specific state, all possible signal sequences are checked whether they lead to a bad state or not. For this decision, the user first has to define the specification with a temporal logic formula. This translation is a common bottleneck during *model checking* since it cannot be done automatically yet. Secondly, the *state explosion problem* yields in high computational cost causing that *model checking* still needs an insufficient amount of time.

As one can see, all those verification methods have several challenges in common: computation time, complexity and strong restrictions to the class of circuits that can be addressed. Therefore, unfortunately, they are insufficient for industrial practice. Instead of such an exhaustive methodology, which is able to detect all possible faults, an incomplete approach is the common way for testing. Based on a set of test cases, derived from the specification at the beginning of the design flow, the correct functionality of the device is verified - with the disadvantage of not being able to detect every single fault.

Defect oriented testing techniques, presented in [KTH+11], [Xin98] and [BXvK+99] all have in common the usage of a DOTSS (defect oriented test simulation system) tool. This tool requires the manual definition of stimuli, the test bench and the limits and supports only the testing of shorts. This lack of an automatic translation yields in high costs.

An approach which can also be used for setting up a simulation environment for AMS tests is proposed in [Lan90]. In the content of this work, a *test pattern specification language* is proposed for translating stimuli in the other way round. The design engineer has to define the test run with this special language which can then be translated into a format supported by the tester. Although the translation could be also done in the other direction, the problem with this approach is the lack of the possibility of injecting and identifying faults in the simulation environment resulting in the problem of not being usable for DOT.

Anther methodology has already been proposed in [KRT02], where a *test setup simulation* was introduced which translates an ATE test program from a specific format into a general file format called GPIF. Furthermore the instruments were modelled to simulate the given test program out of this GPIF file. Also in [BK92], there was a focus on this approach. They introduced a tool called DANTES, which provides a simulation environment as well as a database to share information between the design and the test engineer. The problem of setting up a simulation environment and sharing information was already addressed in [Web89]. All these proposed solutions have one big drawback in common: a lot of manual interaction is needed for the test setup - there is no automatic translation between the design and the test stage in the design flow. Moreover, they neglect the automatic verification (check of certain limits) and are focussed on debugging

a test program with less accuracy which is not sufficient for a DOT test. In other works, e.g. [RKR03], there was just a view at modelling a specific part of the ATE test - the *load board.*

[DG04] for instance gives a more formal approach of defining an ATE test concept. This approach on the one hand shall solve the problem of the manual interaction in the solutions mentioned before. Anyway, the focus of this work also lies on the debugging of a test program instead of the fault injection, caused by the lack of accuracy.

The challenge of defining a usable format through all the design stages has also been tackled by the recently formed Accellera Proposed Working Group with the aim of defining a standard for so called portable stimuli. A portable stimulus is defined once and can be reused through all stages in the design flow. Furthermore it should be not just reusable within the design process, but also between different tools of different vendors. Members of this working group are big companies regarding AMS designs, like Cadence, Mentor Graphics or Synopsis.

The abtsract language for defining the stimuli shall be independent of all the currently used languages. The existence of reusable stimuli would certainly enable a speed up of the verification time and increase the robustness of the whole design process [TSR+09]. In the last update of the working group, a presentation was given where in Figure 1.6 a rough overview of the usage of this abstract language is given. The target is to define an Abstract Portable Stimulus Model which can be parsed into different languages used for verification or testing with one single tool. Therefore this language can be used through the whole design process.

As one can see, the setup of the simulation environment and translation of a test program is not a trivial task. In today's solutions the user has to define test bench, limits and stimuli manually since there is no common format which can be used in all phases of the design process. Although defining standards for portable stimuli has already started, currently there is no possibility for automatically translating a general ATE test program into a format which can be executed in a simulation environment.

1.3 Contribution of this work

Although, the challenge of setting up the simulation environment of an ATE and sharing test information has already been addressed, there is currently no approach that can be used in a fully automatic way. Together with the lack of accuracy, this approaches are insufficient for executing a DOT test. Therefore, we address exactly this problem in the context of this thesis.

We develop a methodology and implement a framework for setting up the simulation environment for ATE tests and translating generic ATE test programs into a format that can be simulated. Since during the design flow of AMS designs several languages (e.g. SystemVerilog, SystemC, Verilog-AMS or VHDL) are used, there is no possibility for reusing the information which was generated at a specific design level. So, there

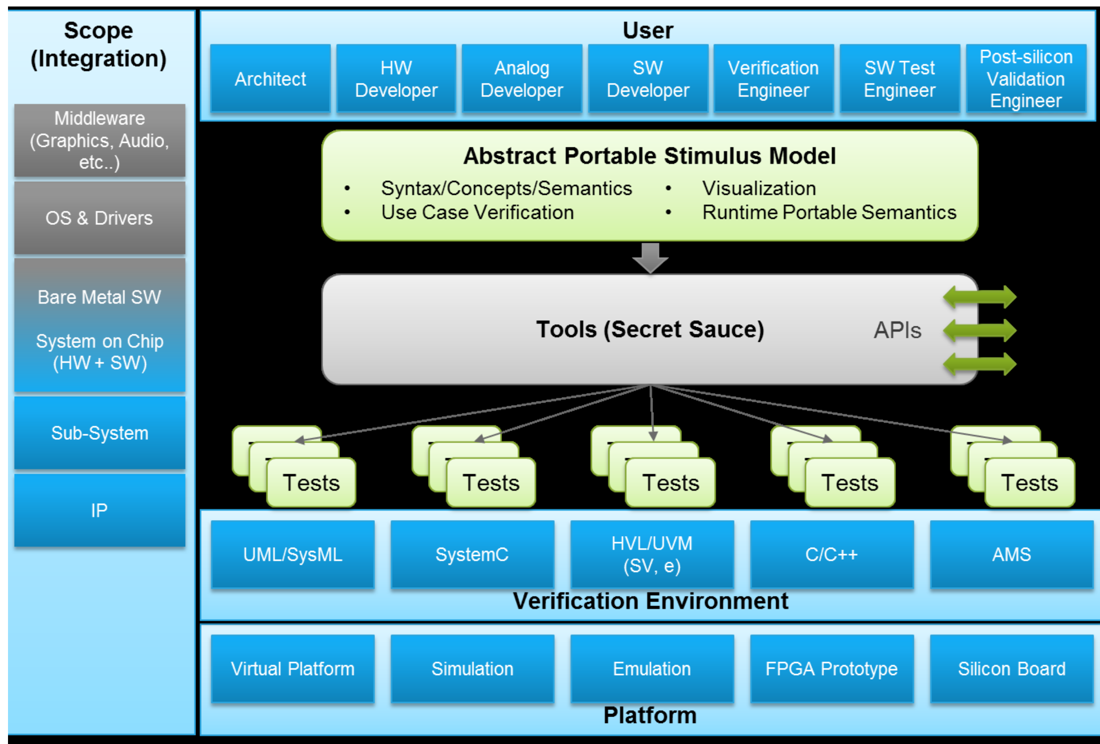


Figure 1.6: Overview of an abstract language for portable stimuli [Gro16]

are completely different languages used for defining the ATE test program as well as defining a simulation run. Reimplementing test benches in different languages used during the design flow contributes significantly to the time-to-market as well as in the design costs. Therefore it is indispensable to support an automatic translation mechanism to support reusing information, which was defined at a specific state. In this thesis, we study the practical challenges of DOT and more specifically, we address the problem of systematically translating an ATE test environment into a simulation environment. The novel contributions of this thesis can be summarized as follows:

1. **Modelling of arbitrary ATE environments:** the thesis provides a general simulation environment that can be used independently from the chosen ATE. The proposed methodology can be used for extending the functionality of the models with minimal effort.

2. **Automatic translation of ATE tests into simulation test runs:** the translation of an ATE test is fully automated, therefore the information from the final verification is reused to configure the simulation environment.

3. **Translation of limit checkers:** we develop a methodology for manual translation of limit checkers from the ATE to the simulation environment and propose a sketch of a procedure that allows automation of this process.

4. **Implementation and the evaluation of the framework:** we implemented all the results presented in this thesis and applied them to a real industrial chip using a real ATE setup.

The main outcome of this thesis is a framework for systematic translation of ATE test programs in the simulation environment. This framework allows in particular the quality analysis of the existing ATE test programs in the context of DOT. We demonstrated its applicability on a real industrial example, making it possible to simulate an entire ATE test for the first time inside Infineon.

This thesis is organized as follows. In Chapter 2, the ATE test will be considered from a general point of view to get an idea what exactly has to be supported by a simulation environment and which constructs are used in such an ATE test program. Afterwards, in Chapter 3, a closer look at the different test instruments and their content shall be given to show how a simulation environment can be set up. Combined with the translation process given in Chapter 4, the simulation process can be executed, which will then be used for evaluating the results on a specific chip from the area of automotive applications in Chapter 5.

CHAPTER 2

Background to ATE test programs

In this chapter, we introduce ATE testing. It starts with a short overview of the requirements for an ATE test, before we continue with the general setup of such a test. In the third part of this chapter, a short overview about common test cases will be shown. As a conclusion, we introduce the test case, which will be used as leading example throughout this thesis. Please note that this chapter is based on [RGA02] and focusses on the part of *final verification* in the design flow as shown in Figure 2.1.

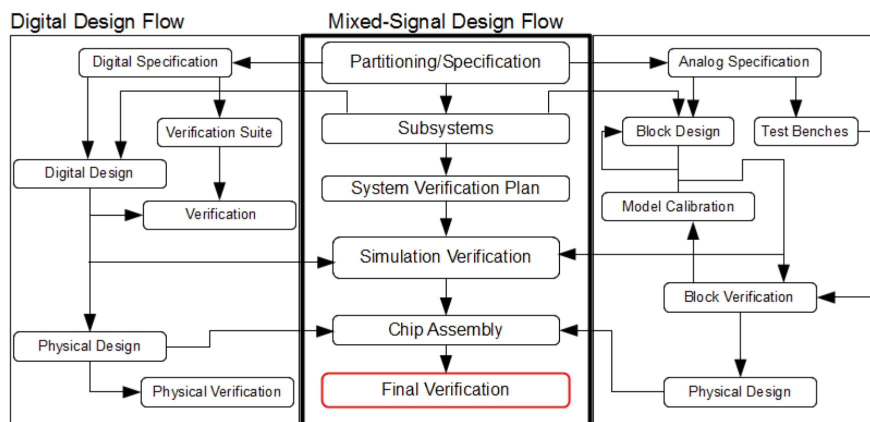


Figure 2.1: Analog Mixed-Signal Design Flow focussed on final verification [BSED07]

2.1 Requirements for an ATE test

Caused by the *random manufacturing defects* it is indispensable to perform a last check of the devices during the *final verification*. Since high test times yield in higher manufacturing costs, it is desired by the industry to keep the duration of the testing activities as low as possible, while the quality of the product has to stay the same. To ensure that these basic requirements are met, test benches based on the specification are developed early in the design flow. There are several targets for them.

Of course, the main purpose of a test is to ensure the quality of the product, which is done by checking whether all the specification requirements are met or not. Faults need to be detected as early as possible to decrease testing costs. Furthermore, it is possible to give quantitative information about the device, like rise times for digital signals. Apart from the checks on the device itself, the test is able to verify the quality of the design process, such that it can be improved for later production.

As one can see, the test benches have an important role in the design process and their development becomes more and more important with the growing complexity of the circuit, especially in AMS designs. Therefore, also requirements for the single test benches should be kept in mind as given in [BSED07]:

1. Scalability: The input stimuli of different technologies are defined as parameter,such that they can be changed easily for other technologies.
2. Configurability: It has to be possible to define all the test bench variables through parameters in a main control file.
3. Automated Result Analysis: The verification of a correct test run has to be performed automatically, without manual interaction.

Nowadays, it is desired to test a high number of devices simultaneously such that the test is automatically executed and controlled by a computer. This is exactly what an ATE is used for: execute and evaluate the test benches at high speed without loss of accuracy of the test. For this purpose, the ATE has to provide several programmable instruments with desired accuracy such that they can be used in a dynamic way.

Furthermore, for verifying the correctness of analog and digital signals, checking waveforms and performing timing measurements, the ATE has to provide measurement units which are able to support these measurements. The general setup of an ATE test will be given in the next section.

2.2 General setup

An ATE test consists of several parts, introduced in this section. The general setup of such a test is given in Figure 2.2. As shown, the DUT is the central element, a chip

consisting of analog as well as digital inputs/outputs. The inputs of the chip are shown on the left hand side of the figure, whereas its outputs can be found on the right hand side. It is the test engineer's task to setup different stimuli as inputs for this DUT and measuring the outputs for verifying its correct functionality. The ATE's objective is to provide the necessary test instruments for achieving these targets. In general, one can distinguish between digital and analog test instruments. Main tasks which have to be supported by the analog instruments are:

1. Provide voltage source with current-clamp
2. Provide current source with voltage-clamp
3. Source an arbitrary waveform
4. Measure voltage
5. Measure current
6. Measure timing information
7. Capture waveform

Whereas the digital instruments have to support:

1. Source digital waveform
2. Measure and verify digital signals

The connection between the instruments of the ATE and the DUT is done by the *load board*. Its purpose it to provide dynamic connections such that the test environment can be set up dynamically for different tests (e.g. open and close relays to switch between different test instruments as inputs for the DUT or provide higher current). Note that the design of the load board is a time consuming task, since a test program combines several tests and therefore different connections might be needed for them. During a simulation run, the correct functionality of the load board should be verified for performing debugging in an early phase of the design flow.

Additionally to the hardware part, the tester is able to understand a special programming language used for describing the test flow. It is possible for the test engineer to open/close relays, connect/disconnect instruments, configure the sourced signals etc. during the test. Furthermore, different test patterns are executed. A *test pattern* describes the sequence of the test vectors, consisting of analog and digital signals, which shall be applied to the DUT. Furthermore, the points in time when strobes should be performed are specified (they are similar to procedures used in imperative programming languages). A strobe

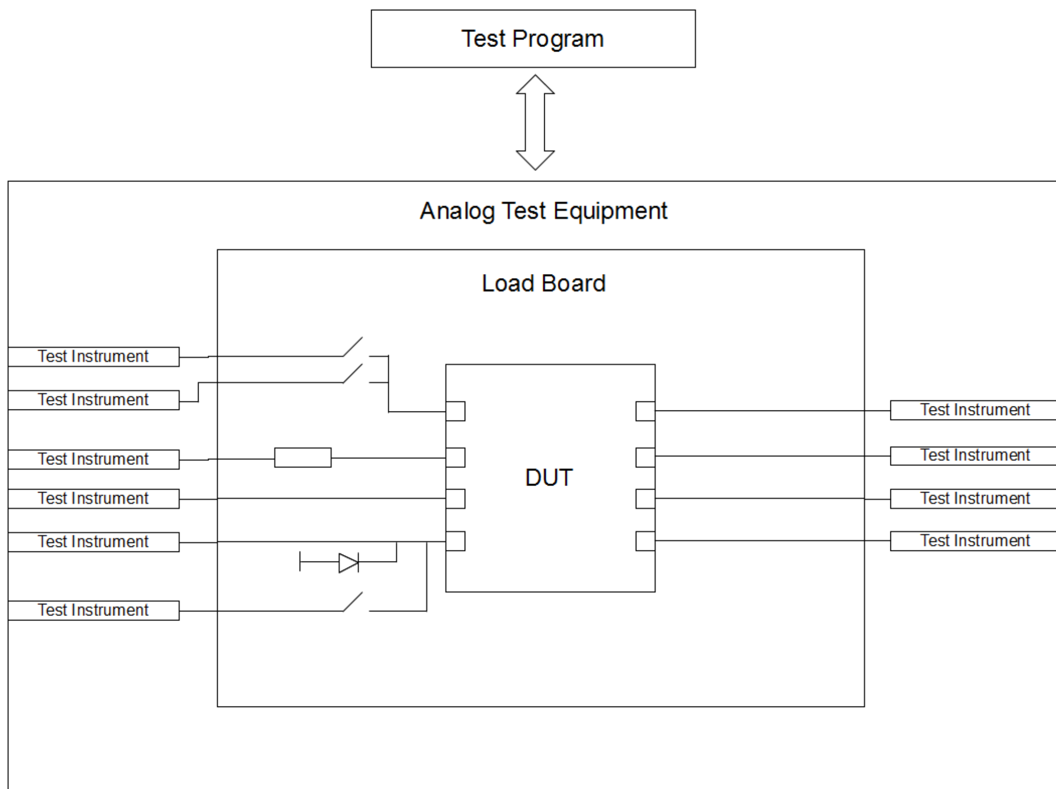


Figure 2.2: Hardware setup of an ATE test

is simply a measurement of an analog or a digital signal within the execution of a test pattern.

Figure 2.3 shows how the software interface of an ATE test is organized. The *program flow* exactly describes the actions performed during the test run and determines the sequence in which the test patterns have to be executed. Therefore, different inputs, defined by the test engineer, have to be set up. First of all, the user has to define in which sequence the patterns shall be executed and the limits which should be used for the test cases. Furthermore, for sourcing/measuring digital signals, the pin levels have to be specified. Moreover, the user has to create and store the arbitrary waveforms which shall be used during the execution. The program flow and the test patterns are normally executed in parallel and can be synchronized with several flags. For instance it is possible to close a relay before continuing with the pattern execution. To read the measurements performed during a test run and check them with the limits, the hardware supports an interface for communicating with the test program.

A typical test setup for AMS designs is shown in Figure 2.4. Analog signals are applied to the analog part of the device, whereas the specified digital signals are applied to the digital part. So, first the test engineers specify the setup of the *load board*, before

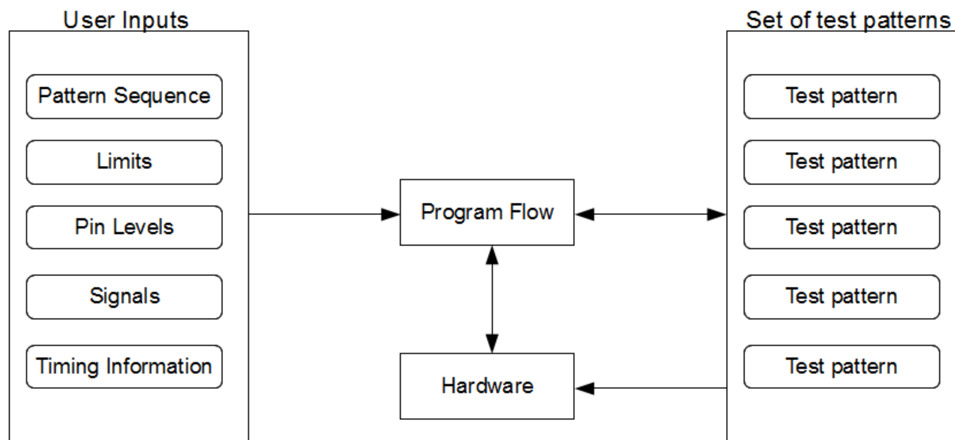


Figure 2.3: Software setup of an ATE test

executing the test pattern. During the execution, several strobes are performed and stored in the memory of the test instruments. After the test pattern has finished, the measurements can be read by the test program. When needed, the test engineer can also do post calculations in the context of the test program. Last but not least, the results are compared with the specified limits and automatically verified in the evaluation block.

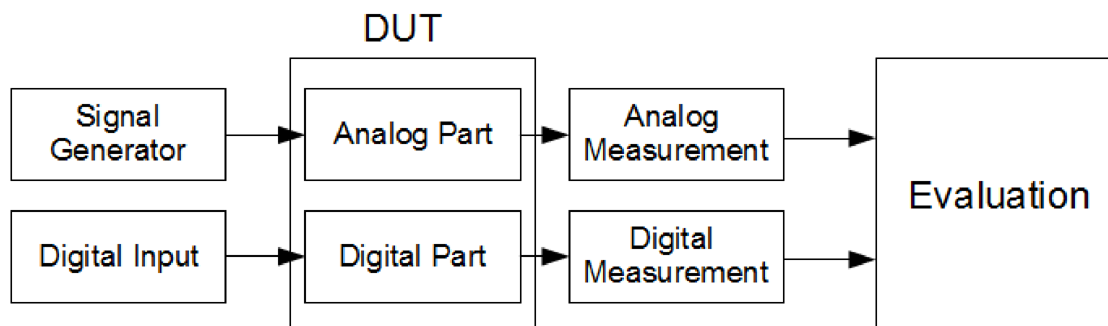


Figure 2.4: ATE test flow

As one can see, setting up an ATE test is a complex task and combines a lot of information. Therefore it is necessary to start developing the test program early in the design flow such that it can already be tested in a simulation environment to ensure its functionality. The ability of automatically translating the specific test program into a format which can be imported into the simulation environment would be a big improvement in the development of an ATE test, which would save time and cost.

2.3 Common test cases in industry

Since test time has a significant contribution to the time-to-market and manufacturing costs, it is strongly desired to keep test times as low as possible. Therefore ordering of the tests is a crucial point to ensure an early detection of a defect during the test. Note: with DOT this ordering can be optimized automatically based on the simulation results.

In general, an ATE test needs to cover a lot of possible faults. Therefore, different test categories have been developed which are quite common in industry. The first test case which is normally executed is the so called *continuity test* - it is responsible to verify whether the connection between the chip and the *load board* is done properly. Afterwards, the parameters given in the specification (like leakage current or power consumption) are verified. Other categories, which need to be covered are digital tests (testing the digital functionality based on truth tables from the specification) and timing tests. A timing test verifies the quality of the signals (e.g. rise times for digital signals). Last but not least, also the combination of digital and analog parts of the DUT needs to be verified - this is done during a mixed-signal test. A more detailed view at AMS testing is given in [Wan06].

During the test, several test cases chosen from these categories are executed (e.g. during the parametric test, the power supply is checked as well as the leakage current). In the remaining parts of this work, it shall be focussed on the *continuity test* as example.

2.3.1 Example Continuity Test

Since the continuity test is one of the most common test cases used in industry, we use it as our illustrating example throughout this thesis. Therefore, a short introduction of this kind of test will be given here. As has already been mentioned, the purpose of this kind of test is to figure out if the connection between the chip and the *load board* is done properly.

Figure 2.5 gives an overview of the functionality of a continuity test. First of all a single pin is selected which shall be tested for opens or shorts. Normally, a pin is used as input path to some circuit, which is protected by the *protection ESD diodes*. Exactly these diodes are used for the continuity test. The corresponding test instruments are connected to the pins of the DUT. Every instrument acts as a voltage source, sourcing 0V. The instrument connected to the pin which shall be checked is configured to sink current. The current flows through the protection diode to the test instrument - therefore the voltage measured by the instrument should be in a certain area around a diode voltage of 0.7V. Otherwise there exists an open/short between two pins.

We use this simple kind of test throughout this thesis as leading example. What remains to show is the setup of the test program for running a continuity test. As ATE we used Teradyne's Flex Tester. Figure 2.6 depicts the parts a test program consists of. The communication between tester and software is done by some code, written in *Visual Basic*. This program is split up into several specific procedures, where every procedure

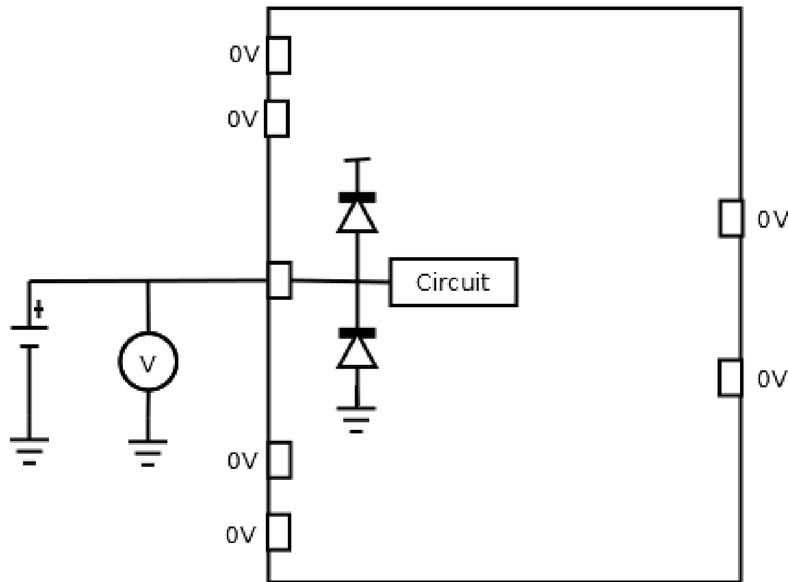


Figure 2.5: Continuity test

represents a single test. Within this code, the test engineer has the ability to fully control the different instruments of the ATE, read the measurements or run conventional program code like calculations or loops. Furthermore it is possible to define single test patterns. A test pattern is similar to a procedure - it contains a set of commands, which are executed in parallel to the Visual Basic code. Also in the content of the test pattern, program constructs like loops or other conditional statements are available. To ensure a deterministic test, the user can synchronize the pattern and the Visual Basic codes via specific statements (e.g. wait). Whereas the Visual Basic program has a bidirectional communication to the ATE (instruments can be configured and information like measurements can be read), a pattern can just configure the instruments, which makes sense because after reading a measurement the user might perform post calculations. We are having a closer look at the detailed configuration parameters in Chapter 4.

What remains is the configuration of such a test program. The test engineer might want to control the Visual Basic code and the patterns to be executed from a *graphical user interface*. This configuration is done via specific *Microsoft Excel Sheets*, where the essential ones are shown in Figure 2.6. For example, in the *Test Instances Sheet*, the test flow is specified - here the user can exactly define the test cases (procedures in the code) to run. The sheets describing the instruments of the ATE are the *Pin Map* (specifies the instrument and relay names, which will be used throughout the program) and the *Channel Map*, which maps the names of the *Pin Map* to the physical identification of the ATE instruments (they are normally identified by unique numbers). Whenever a function for configuring an instrument is called, the name specified in the *Pin Map* is used as identification. Note, that also a grouping of several instruments is possible.

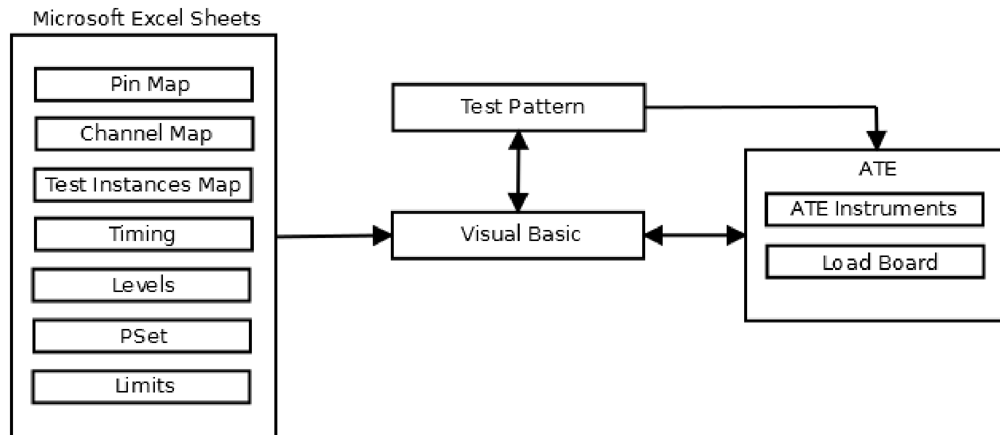


Figure 2.6: Setup of the Teradyne Flex Tester

The *Timing Sheet* as well as the *Level Sheet* define the parameters, needed during a program run, like which voltage values indicates a logic 1, when we want to make a digital measurement or for how long a signal is active. A set of these parameters is available for every instrument - these sets are defined in these two sheets. As already mentioned, during a test run, different parameters are applied to single instruments. Instead of configuring one by one, Teradyne also supports configuring an instrument with a parameter set (PSet), containing a group of configuration parameters. They are defined in the *PSet Sheet*, but in contrast to the other sheets, this one is not mandatory.

Last but not least we have the *Limit Sheet*. Here the test engineer can define the limits, to verify the correct behaviour of the DUT. The limits are defined for every test separately in a list. Whenever a check is executed in the test program, the next values are taken from the list and checked with the desired value.

After this short introduction to Terdyne's Flex tester, we present now a small example of the continuity test how it is used for an Infineon chip as DUT. We neglect all the sheets, since they are very specific and not necessary for the next parts. Just keep in mind that the timings, levels and parameters are configured in the context of these sheets. Note that they can also be configured within the *Visual Basic* code.

Listing 2.1: Setup of a continuity test

```
Call Relay_On("OUTxF_k", 0.02) 'maintain 10ms

Call Alarm_On("Vs_dc30,Vdd_dc30,IS_dc30,all_dc90_hi_a", False)

TheHdw.DCVI.pins("Vs_dc30,IS_dc30").Connect (tlDCVIConnectDefault)

TheHdw.Wait (resource_relay_settling)
TheHdw.DCVI.pins("Vs_dc30").PSets("DC30v_0VR30V_100mAR200m_measV").Apply
TheHdw.DCVI.pins("Vs_dc30").Gate = True
TheHdw.DCVI.pins("Vs_dc30").Disconnect (tlDCVIConnectHighGuard)

TheHdw.Wait (0.001) 'maintain 1ms
TheHdw.DCVI.pins("all_dc90_lo_a").Connect (tlDCVIConnectLowForce)
```

```
TheHdw.Wait (0.006) 'maintain 6ms

If quad220_device = True Then
    TheHdw.DCVI.pins("all_dc90_hi_a").Connect (tlDCVIConnectDefault)
    TheHdw.Wait (0.006) 'maintain 6ms
Else
    TheHdw.DCVI.pins("all_dc90_Quad").Connect (tlDCVIConnectDefault)
    TheHdw.Wait (0.006) 'maintain 6ms
    If quad040_device = True Then
        Call Relay_On("SO_OUT5_k", 0.002)
        TheHdw.DCVI.pins("OUT0_dc30,SO_OUT5_dc30").Connect (tlDCVIConnectHighSense)
        TheHdw.DCVI.pins("OUT0_dc30,SO_OUT5_dc30").Connect (tlDCVIConnectHighForce)
        TheHdw.Wait (resource_relay_settling)
        TheHdw.DCVI.pins("OUT0_dc30,SO_OUT5_dc30").LocalKelvin = False
        TheHdw.DCVI.pins("OUT0_dc30,SO_OUT5_dc30").Disconnect (tlDCVIConnectHighGuard)
    End If
End If
TheHdw.PPMU.pins("all_hsd").Connect

TheHdw.DCVI.pins("Vs_dc30,IS_dc30,all_dc90_lo_a").LocalKelvin = False

TheHdw.Wait (resource_relay_settling)

TheHdw.PPMU.pins("all_hsd").ForceV 0, 200 * uA, 200 * uA, -200 * uA
TheHdw.Wait (0.001)


Call TheHdw.Digital.Patgen.Continue(0, cpuA) 'VBT_code

TheHdw.DCVI.pins("Vdd_dc30").PSets("DC30v_0VR30V_100mAR200m_measI").Apply
TheHdw.Wait (0.001)
TheHdw.DCVI.pins("Vdd_dc30").Gate = True
TheHdw.Wait (0.001)
TheHdw.DCVI.pins("Vdd_dc30").Connect (tlDCVIConnectDefault)
TheHdw.Wait (resource_relay_settling)
TheHdw.DCVI.pins("Vdd_dc30").LocalKelvin = False
TheHdw.Wait (resource_relay_settling)

If quad220_device = True Then
    TheHdw.Digital.Patterns.Pat(Continuity.pat").Start␣("Continuity_Quad220_Initial")
End␣If
If␣quad040_device␣=␣True␣Then
␣␣␣␣TheHdw.Digital.Patterns.Pat(Continuity.pat").Start ("Continuity_Quad040_Initial")
End If
If penta_device = True Then
    Then TheHdw.Digital.Patterns.Pat(Continuity.pat").Start␣("Continuity_Penta_Initial")
End␣If
If␣hexa_device␣=␣True␣Then
␣␣␣␣Then␣TheHdw.Digital.Patterns.Pat(Continuity.pat").Start ("Continuity_Hexa_Initial")
End If

/* Note: from here the pattern and the VBA code are executed in parallel */
/* We jump directly to the limit checks */

gnd_vs = TheHdw.DCVI.pins("VS_DC30").Meter.Read(tlNoStrobe, 5, -1, Average)

/* The voltage has to be less than -0.7V */
datalog(gnd_vs)
```

Listing 2.1 illustrates an example of the program code for the setup of a continuity test. This code snipped is used for configuring the relays of the load board, connect the needed instruments and set the voltage of all instruments to 0V as common for a continuity test. As we can see, with conditional statements several types of DUTs are distinguished (e.g. devices with different channels). In the end the program chooses the pattern to run depending on the device type.

Listing 2.2: Pattern for the continuity test

```
:vec_nr  :patexe              :ws  :stimuli          :pset            :ms
    1,   -,                     7,        -,      d137 d258 d142 d6 d6 d6 d6 . .,
    2,   $repeat=100,           7,        -,      -,
    3,   $repeat=3000,          7,        -,      -,
    4,   -,                     8,        -,      -, . . $Enable_Alarm . . . . . .
    5,   $repeat=5,             8,        -,      -, . . $Strobe . . . . . .
    6,   -,                     8,        -,      -, . . $Disable_Alarm . . . . . .
    7,   -,                     7,        -,      . . d140 . . . . . . .,
    8,   $repeat=3000,          7,        -,      -,
    9,   -,                     8,        -,      -, . . $Enable_Alarm . . . . . .
   10,   $repeat=5,             8,        -,      -, . . $Strobe . . . . . .
   11,   -,                     8,        -,      -, . . $Disable_Alarm . . . . . .
   12,   -,                     7,        -,      . . d137 . . . . . . .,
   13,   $repeat=500,           7,        -,      -,
   14,   -,                     7,        -,      d139 d141 d155 d9 d9 d9 d9 . .,
   15,   $repeat=500,           7,        -,      -,
```

In Listing 2.2 we see an extraction of the pattern in GPIF format [KRT02] as it is called in the context of the code. Here, every line describes a test vector, where every line consists of several columns. The first column just describes the line number (which can be used for jumps inside the pattern), whereas in the second column simple programming constructs can be used (e.g. repeat, call, jump). The code is followed by a timing information, represented by a number. This number identifies the timing defined in the *Timing Sheet*. This timing information determines the duration of such a line. After this duration the next line is executed. With this information it is possible to satisfy timing constraints like rise times. The next column describes a digital waveform - a list of digital instruments is defined at the beginning of a pattern. The digital waveform assigns a logical output or expected value to every of these instruments. Afterwards, the analog waveform is defined, just with the difference of applying a PSet instead of a logic value. A dot means "don't change". The last column can be used for executing so called microcodes for every instrument. Here for example, a filter can be enabled for an instruments or strobes can be made to be read afterwards in the Visual Basic code.

In Chapter 4 we show the parts and commands used in this program from a more abstract perspective and use it to clarify which steps have to be taken until we are able to run it in the simulation environment we propose in the next chapter.

CHAPTER 3

Setting up the simulation environment

In this chapter, we develop a model of an ATE test for the simulation environment. Although there are already test benches available in the simulation environment, they are not useful as model for the ATE test, since they are not using the same instruments with the same properties and the load board might be neglected. Therefore it is necessary to model the different resources, available at an ATE to run the test in a simulation equivalent to the real test. We first present the overall description of a test environment model using a top-down approach. We then instantiate this model to a concrete example of an Infineon chip. We implement this concrete example, using the Cadence Virtuoso tool and simulate it with Spectre. The modelling activity presented in this chapter is part of the mixed-model verification part of the design flow, as illustrated in Figure 3.1.

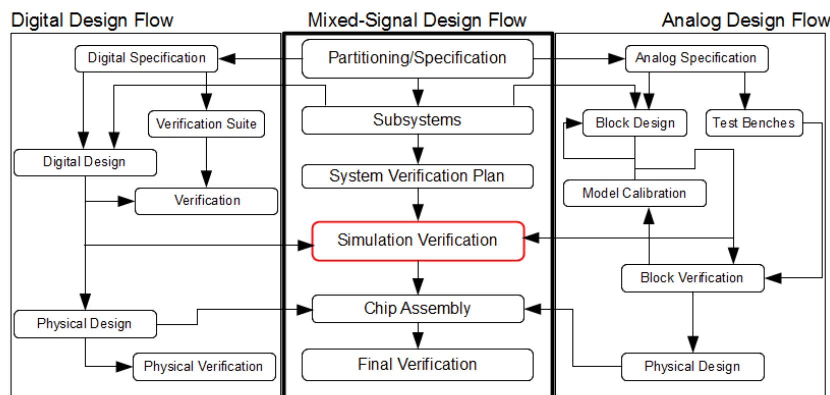


Figure 3.1: Modelling part in the design flow [BSED07]

3.1 Modelling of an ATE test

In this section, we present a step-by-step overview of the ATE test model that is not specific to concrete instrument types. Whereas in the ATE test several kinds of resources are available (e.g. high current/low current or analog/digital instruments), we model all of them with one single model. We illustrate the modelling steps and the use of the resulting model with an example from the automotive field, a chip provided by Infineon Technologies. This chip plays the role of the DUT in our presented scenario.

Figure 3.2 depicts the top level setup of the hardware of an ATE test. The ATE typically consists of several kinds of test instruments connected to the *load board* and provides different functionalities. As one can see, we are using the same model for every instrument, since a general schematic is used for all types of test instruments. They are connected to the DUT via the load board. Consider the pinning block in Figure 3.2. This block includes the detailed connection of the DUT to the load board. It is used to introduce force and sense lines for the design.

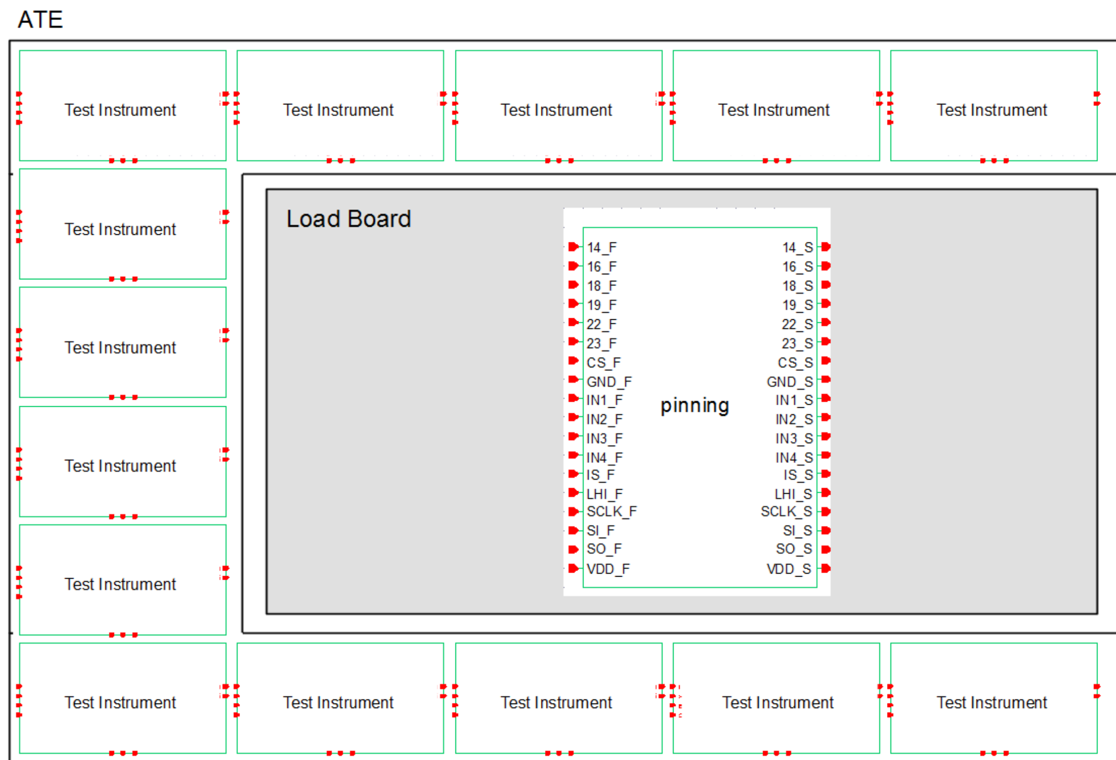


Figure 3.2: Top-Level setup of the simulation environment

We note that the *load board* is a very special part of an ATE test. Its configuration depends on the available resources (instruments) of the specific ATE and the tests to be executed on the DUT. Therefore, the load board is specific to the design and its test configuration. Due to this specificity, we do not detail the model of the load board in

this thesis. One should just keep in mind that this part has to be remodelled whenever a new load board is used in the test. Note that the problem of modelling the *load board* was also addressed in [RKR03].

There is just one part of the load board, which can be modelled generally: the connections to the DUT. During the test, two kinds of connections are used - a force line and a sense line. The force line is responsible for sourcing a specified voltage, current or waveform, the signal is driven by the source of the instrument. In contrast to the force line, the sense line is used for measuring voltage. The distinction between these kind of connections is necessary, since we want to measure voltage directly at the DUT according to the *kelvin measurement principle*. We note that the sense line is shorted with the force line, in order to measure voltages inside the test instrument later on. This shorting is referred to as *kelvin connection*.

The wires used for connecting the power supply of the DUT to the load board are shown in Figure 3.3. On the left hand side, one can see the input pins to which several test instruments can be directly connected. Sometimes it is necessary to use an implicit wiring, where connections are realized via *ConnectionByName*. Moreover, Cadence supports global wires, available through all design hierarchies. They are identified by a "!" at the end of the wire name. Therefore, the input pins are connected to the global wire *vsub!*, which is the power supply of the model of the chip. Moreover, the output pin, VS_S is shorted with *vsub!* for measuring the voltage. We can see that these two wires are not directly shorted, because of a specific tool feature. Cadence Virtuoso does not allow a direct connection between two or more input pins. One solution to overcome this issue is to use low ohmic resistors for these connection, with the disadvantage of introducing more complexity to the overall design, yielding in higher computational effort during the simulation. Therefore we used the IProbe, which is normally used for current measurements. In the circuit they act like a resistor of 0Ω, but do not increase the computational effort.

The force and the sense lines of the remaining input pins of the chip can be connected directly, since no global wires are used. Similarly to the previous case, we see on the left hand side of Figure 3.4 the input pins, also known as force lines, whereas on the right hand side, the sense lines have a direct connection to the output pins.

With these connections, we can both force current or voltage as input to the DUT pins and measure voltage directly from the sense lines, shorted with the force lines. We recall that in this thesis we ignore the load board - hence we directly connect the DUT pins to the instrument models. In reality, these connections are implemented via the manually modelled load board.

This concludes the presentation of the high level model of the ATE test environment. In the remainder of this chapter, we focus on the specific models of the test instruments. Figure 3.5 depicts a conceptional schematic of a test instrument, whereas Figure 3.6 shows the block diagram of a test instrument. We start with a description of the static interface (the pins) in a test instrument. Consider the right hand side of Figure 3.6.

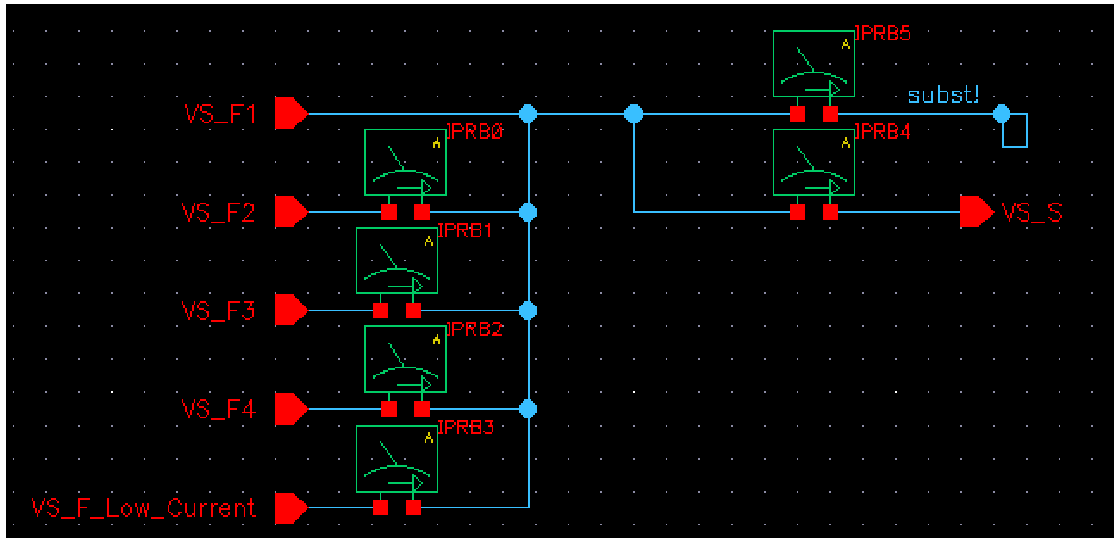


Figure 3.3: Connection between DUT supply and load board

The pins *channel_A_hi* and *channel_B_hi* act as high side connections of the source included in the test instruments. With these channels, it is possible for the test engineer to realize two different connections for the high side and select the active one in the context of the test program. At most one of these channels is connected at any time, never both. A third output which might be used is the so called *dibaccess*. Via two switches this output pin can be either connected to the sense or the force line. The *dibaccess* can be used for implementing an additional connection, like a flow from the force line to ground.

On the bottom side of the block, we have the low force connections *channel_A_lo* and *channel_B_lo*. Similar to the high force line, one can multiplex between these low force connections. Moreover the channels of the high and the low interface can be mixed, e.g. on the high side *channel_B_hi* might be used, while on the low side *channel_A_lo* is connected. Furthermore, on the low side we have modelled a *ground* pin, which is used for voltage sources and the switches inside the block. In contrast to the other channels, ground is connected all the time.

Furthermore, we have the input pins for measuring voltage on the left hand side of the instrument - *sense_A_hi*, *sense_A_lo*, *sense_B_hi* and *sense_B_lo*. Similarly to the source, the voltmeter needs a high as well as a low side connection. Also here, the test engineer can switch between two channels for both sides. Again, it is possible to mix the channels of the high and the low sense interface. For our DUT, just one of the sense channels (*sense_A_hi*) is used. It is either connected to ground or floating for the power supply.

Apart from the pins used for driving and measuring signals, a test instrument can perform two evaluations on the measured voltage: verify the logical value or performing a timing

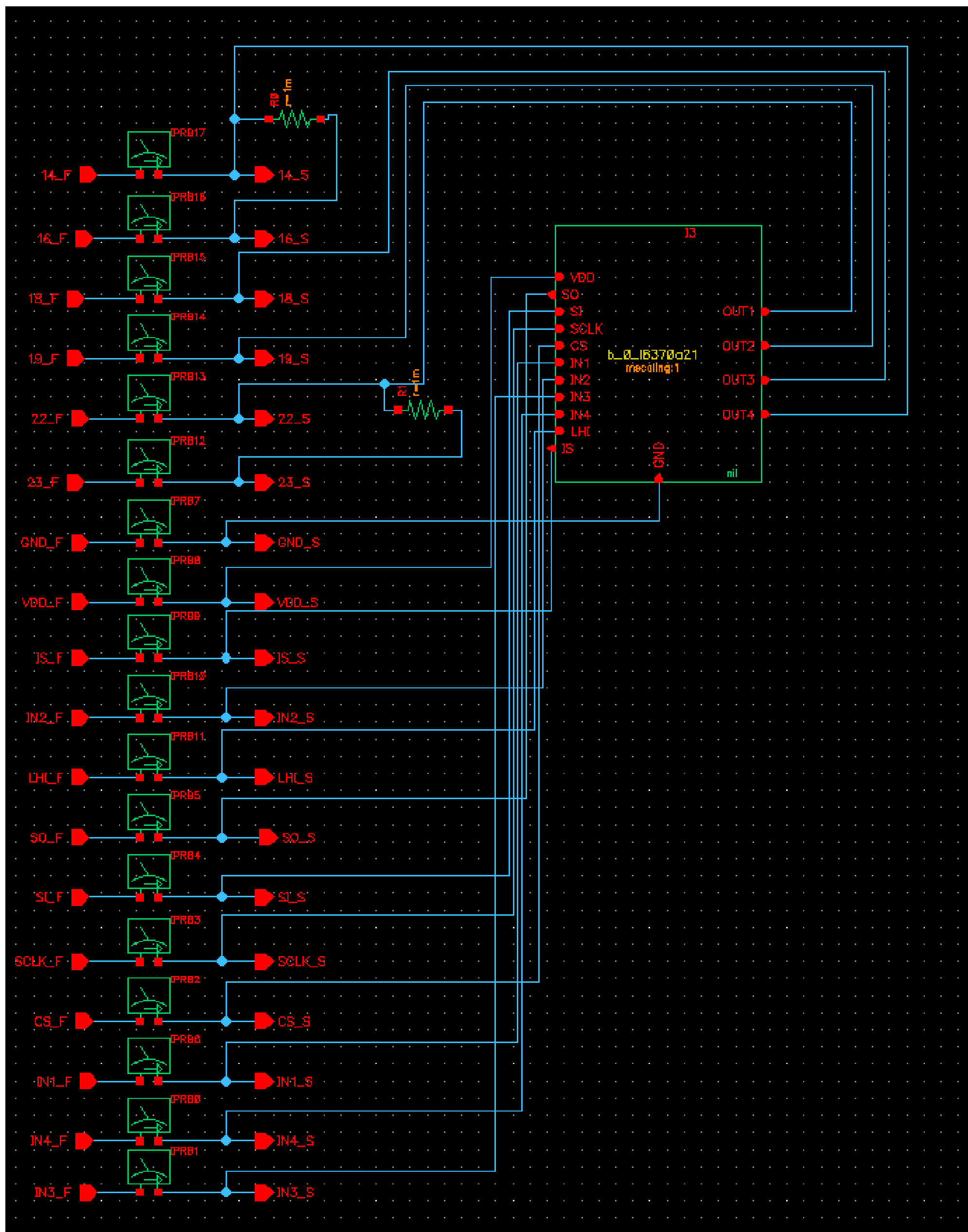


Figure 3.4: Connection between DUT and load board

measurement. Since all these informations are needed for limit checking, we use additional output pins for the corresponding values. They source the measured value as voltage such that the limit checker can use them as input for post calculations and verification.

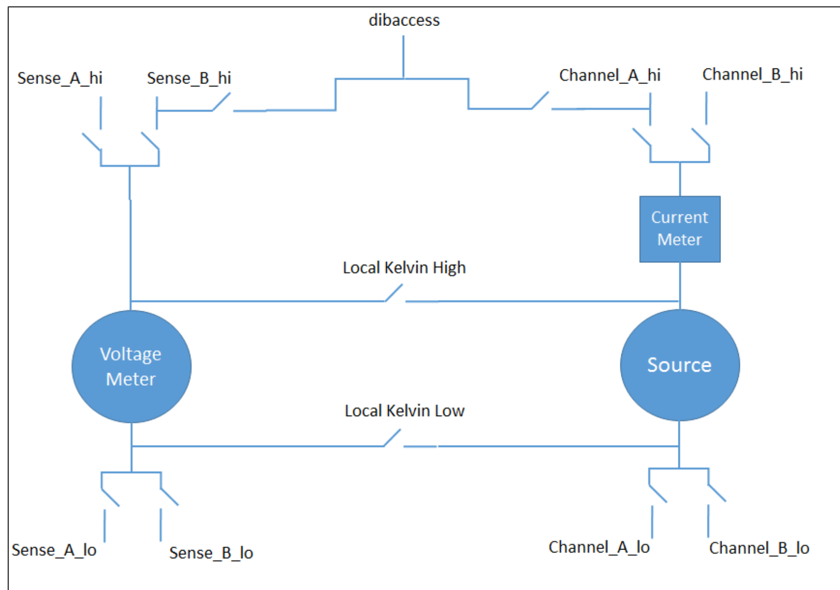


Figure 3.5: Schematic of a test instrument

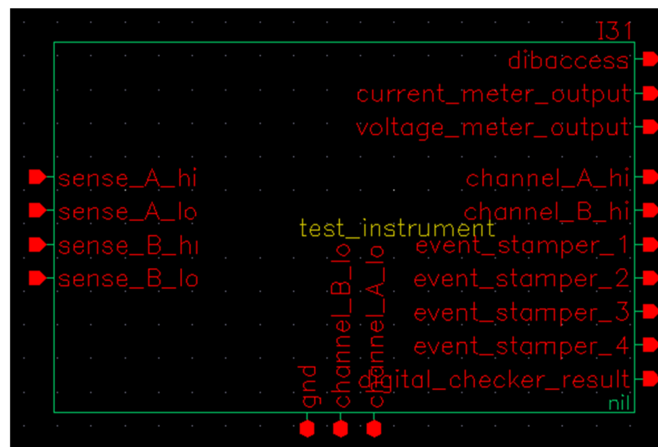


Figure 3.6: Pinning of a test instrument

3.2 ATE instrument model

We now present the actual models of the test instruments. Normally one can distinguish between different kinds of test instruments. In general there are analog and digital instruments, whereas one can further divide these categories. For instance, Teradyne's Flex tester distinguishes between grounded and floating analog instruments, high/low current/voltage instruments, instruments with timing measurement capabilities and so on. Since this is caused by the available resources of such a tester (increasing costs with increasing number of complex instruments), this is just an issue of the tester hardware. In

the context of a simulation environment it is not necessary to take care of this hardware specialities - therefore we designed a general test instrument which combines all these features, to reduce the complexity of the modelling work and ensure that changes can be made easily (e.g. when a test instrument of a special type is replaced in the ATE test by another one).

The overview of a generic test instrument model is shown in Figure 3.9. The central element is the block in the middle (1), containing the source which can dynamically switch between a current or a voltage source. The high pin is located at the right side of the block. It is connected through an ampere meter (an *iprobe* called CURRENT_METER (2)) to two switches: one connected to *channel_A_hi* and another one connected to *channel_B_hi* (7). At every point in time, at most one of these switches is closed, the other one has to be open. They are controlled by a voltage piecewise linear source (VPWL) - 5V indicate a closed switch (low resistance), whereas 0V indicate an open switch (high resistance). Also on the low side, two switches act as a multiplexer for selecting the desired connection from the available ones.

Apart from this forcing part of the instrument, the sensing input is responsible for measuring voltage at a specific point in the hardware (current is measured directly at the force line with low resistance). A voltage controlled voltage source (VCVS) is used for acting as a voltmeter (3). The two switches between the high input of the voltmeter and *sense_A_hi* and *sense_B_hi* are acting as a multiplexer such that at every point in time just one of these lines is used as input. The low sense lines are connected in the same way. In our example, just channel A is used on the low sides (for the force as well as for the sense line). For most of the instruments, the low force lines are connected to ground. The test instruments connected to the power supply need to be floating - therefore their low force/sense lines are connected to the output ports of the DUT. The voltmeter measures the voltage between the high and the low sense input and sources the measured voltage to the net *vmeas_out*, such that it can be used for timing measurements or the verification of a logical value. Current metering works in a similar way: a current controlled voltage source (CCVS) (4) uses the measurement of CURRENT_METER as input and sources the corresponding value. How the voltages representing the actual voltage/current are sourced throughout the output ports is shown in Figure 3.7, such that they can act as input for the limit checker.

We can see, that there are two additional switches - one for directly connecting the high force line with the high input of the voltmeter and another one for connecting the low force line with the low input of the voltmeter. These are so called *local kelvin connections*(5). These connections are useful to have a reference even when no sense line connection is active. The main intention of the *local kelvin* is to have a reference for the control loop used in the test instrument. Moreover it is also possible to force voltage or current over these sense lines - one has to connect the *local kelvin connection*, disconnect both force lines and connect the desired sense line. This might be useful in some cases (e.g. providing higher current in that way because of a limited number of available resources).

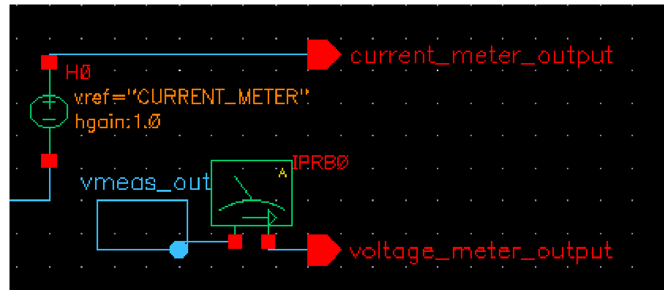


Figure 3.7: Meter outputs

On the right hand side of the model, we have two additional blocks - one is responsible for performing timing measurements (8), whereas the other one verifies a desired logical value (9). Both use the measured voltage from *vmeas_out* as input. Additionally for both evaluations the information of the pin levels (voltage in high (VIH) and voltage in low (VIL)) is needed. Since the levels can also change dynamically during the execution, they are controlled externally. Therefore one VPWL is used for sourcing the actual value of VIH and another one for VIL (6) - the outputs are sourced to the nets called *VIL* and *VIH*.

In the following sections, we have a closer look at the blocks used inside a test instrument to get a better understanding of their functionality and configuration.

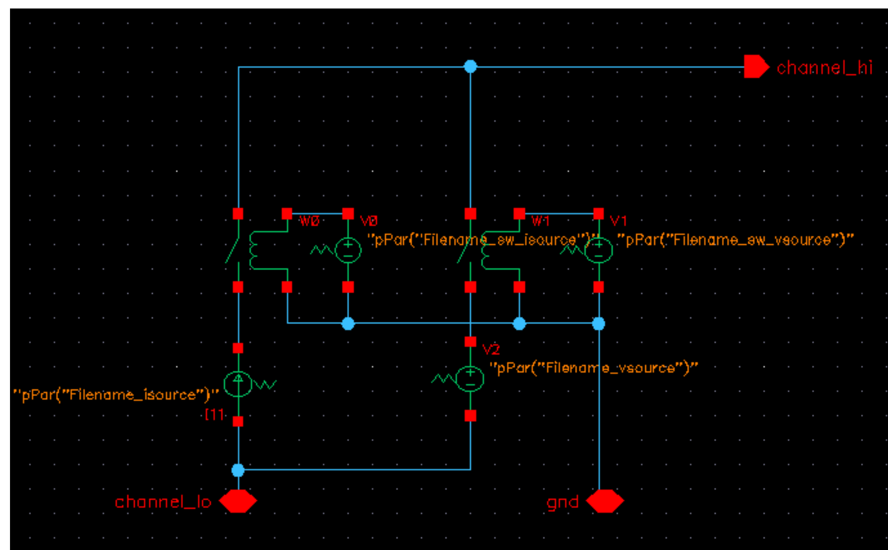


Figure 3.8: ATE test instrument channel

3.2.1 The source

This section provides a closer look at the used source within an instrument. It has already been mentioned, that an instrument can either act as a voltage or current source. The mode of the source might change dynamically during the execution of the ATE test. As shown in Figure 3.8, this block contains a VPWL as well as an current piecewise linear source (IPWL). Two switches are used for the dynamic mode change - at every point in time, exactly one of the switches is closed. One needs to take special care of the current source: whenever the current source is used, it needs to be ensured that the switch is closed, otherwise this behaviour would result in high voltage and undesired behaviour. The signal is sourced at the output pin *channel_hi*, while the pin *channel_lo* is used as reference (it can either be grounded or floating). The *gnd* pin is just used for the controlling voltage sources and the switches. We use this configuration also for digital signals - here just the voltage source is active with the voltage corresponding to the desired logical value.

The central element in the whole model is the VPWL. Such a source is configured by a pair consisting of a point in time (in seconds) and a voltage value (in Volts). The source linearly interpolates the voltage values between these two points in time. Since the classical VPWL has an upper bound on the number of points which can be defined and cannot be controlled from external, an extension of this kind of source must be used: the voltage piecewise linear source - file controlled (VPWLF). In general the VPWLF supports the same features as the classical VPWL with the difference that it reads the pairs from an external file with a much higher upper limit.

An example input file for an VPWLF is shown below. While the output of the source is shown in Figure 3.10. As one can see, the voltage is configured to stay at 0V until 5ms, then it starts rising linearly to 5V for 2ms, before it is linearly decreasing to 0V again from 9ms until 10ms. The configuration for the current source (current piecewise linear source - file controlled (IPWLF)) works the same.

```
0 0
0.005 0
0.007 5
0.009 5
0.010 0
```

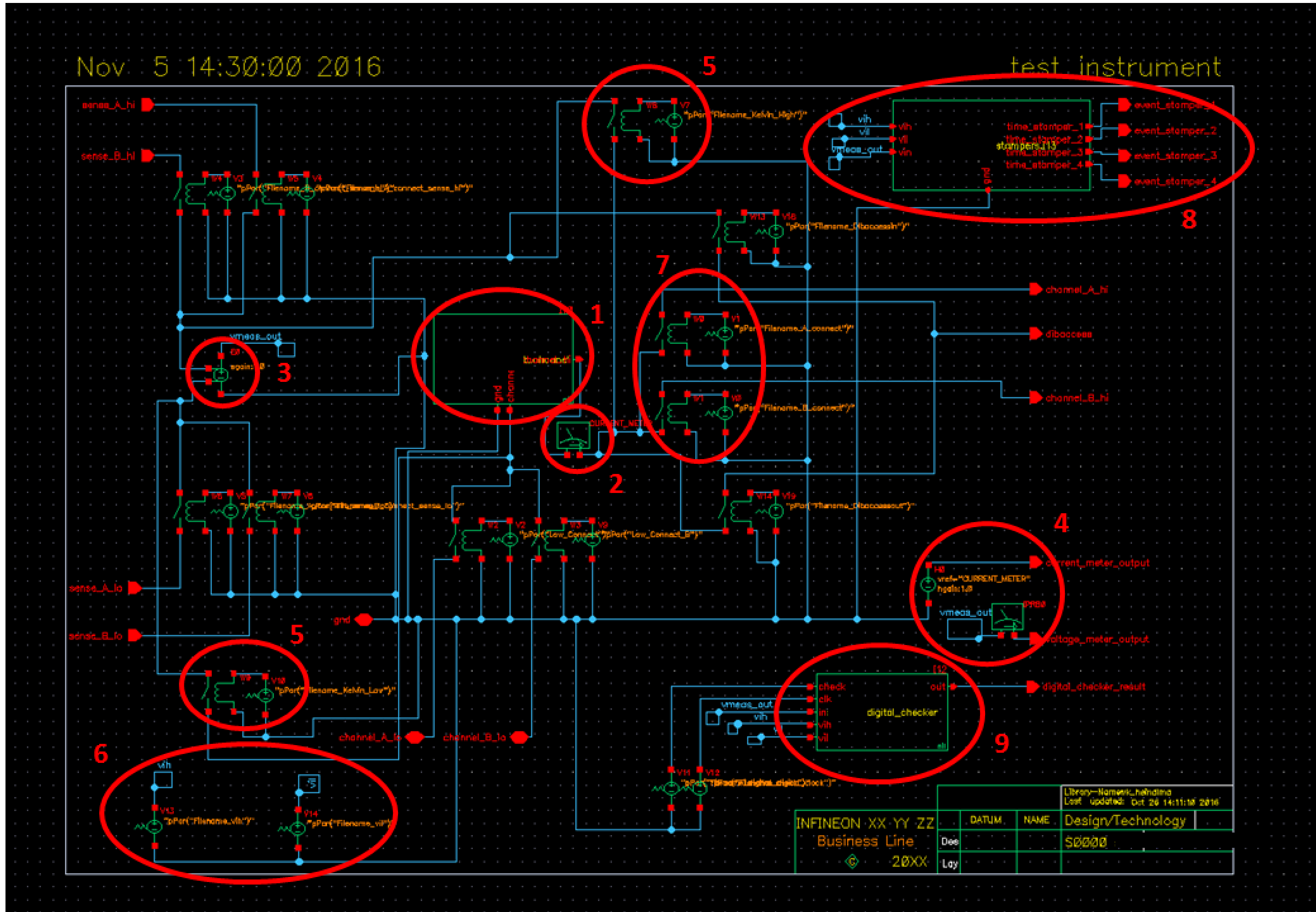


Figure 3.9: ATE test instrument

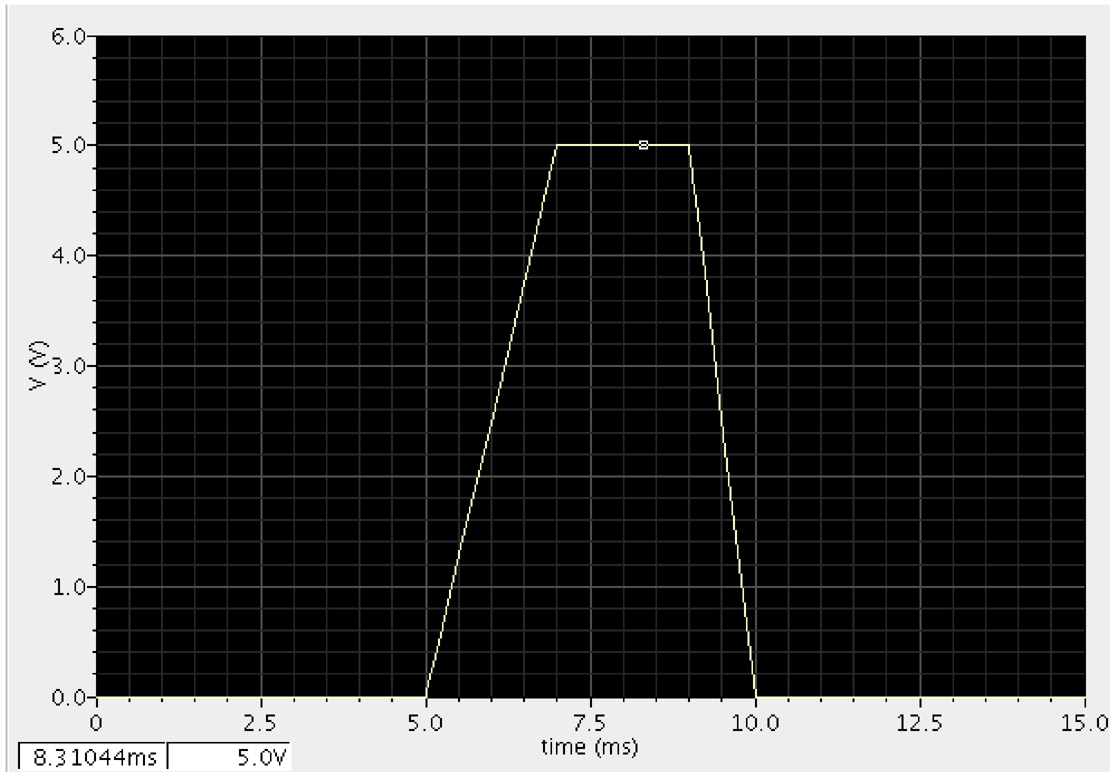


Figure 3.10: Example definition of a waveform

At the end of this chapter we would like to mention an additional feature supported by ATE testers: the voltage/current clamp. For example, the test engineer has the possibility to define a current clamp for a voltage source. When the current reaches this clamping value, the voltage source takes the role of a current source, forcing the clamping value as current. For a current source with a voltage clamp it works symmetrically. Modelling this feature together with dynamic clamps usually yields in convergence problems during the simulation and at the moment there is no known solution to this issue in spectre. Therefore we neglect this feature and assume that an excess of the clamp just occurs in the case of a fault.

In the next sections we have a closer look at two evaluation blocks used during simulation.

3.2.2 Digital checker

In AMS designs, we need to verify both, analog and digital behaviours. Whereas the sampled value of voltage or current is sourced directly out of the pins *current_meter_output* and *voltage_meter_output*, the digital value is checked first inside the digital checker which is shown in 3.11 as it is used in the test instrument. Within the ATE test program, the test engineer can specify the expected logic value at some point in time. This expected logic value is sourced by a VPWLF - 5V correspond to an expected logic high value,

whereas 0V mean that a logic zero is expected. The nets *VIH* and *VIL* are used as input for the actual pin levels. For the information whenever a check shall be performed, an additional signal is used: at every (rising or falling) edge, the digital checker verifies the input voltage (*vmeas_out)* with the expected value according to the pin levels. As long as no fault occurred, the output *out* is set to 5V. In case the expected and measured value differ it is set to 0V. This output is written to *digital_checker_result* such that it can be verified in the limit checker later on.

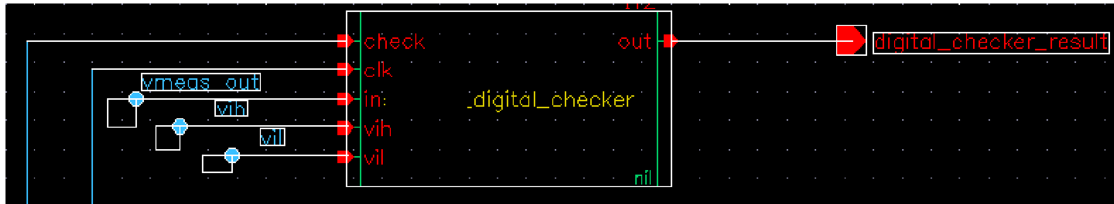


Figure 3.11: Digital checker block

Figure 3.12 depicts an example of the usage of a digital checker during the simulation. The first signal *clk* is used for triggering the measurements, whereas *check* denotes the expected value. In the third line, one can see the signals *vmeas_out*, *VIH* and *VIL*, whereas in the last line we have the output *digital_checker_result*. We note, that at every event (edge) of the clock, the signal *vmeas_out* corresponds to the expected logic value, therefore the output remains at 5V.

Figure 3.12: Example for the functionality of the digital checker

In our environment we implemented the digital checker in Verilog-A - the corresponding code an be found in the appendix.

3.2.3 Event stampers

Another measurement used during a test is the timing measurement. It is necessary to ensure that rise/fall times meet specification requirements of the device. Every test instrument consists of four event stampers, combined in the block *event_stampers* shown in Figure 3.13. It takes the actual levels *VIL* and *VIH* as well as the measured voltage *vmeas_out* as input. An event stamper is triggered by an event (e.g. leaving the low value). Whenever this event occurs, the time is measured and written to the output port. The outputs of the block are the times measured by every event stamper.

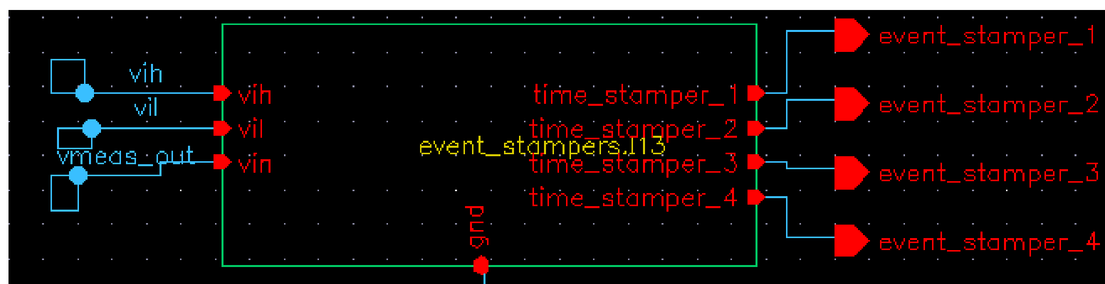


Figure 3.13: Event stamper block of the instrument

Figure 3.14 shows how the four event stampers are connected to the input/output pins. Every event stamper uses the levels and the measured voltage as input. Additionally we need to specify which event shall trigger a measurement. Here we distinguish between *leaving low*, *leaving high*, *reaching low* and *reaching high* events. These events are discussed in more detail in the next chapter. Furthermore, an event stamper block needs to be enabled - otherwise no measurements are done. The events of interest are configured via the voltage of a VPWLF where every voltage level indicates an event. Every stamper can be triggered on a different event. To enable the desired block, the VPWLF of the enable signal needs to be set to 5V. When this source is set to 6V a measurement is performed immediately without waiting for an event. In the case an event happens more often, always the last time is sourced at the output.

An example of such a timing measurement is shown in Figure 3.15. As can be seen, the digital voltages *VIH* and *VIL* are set to 2.3V and 13V. The voltage indicating the trigger event is about 1V which means we are waiting for a *leaving low event*. When the enable signal is set to high, these values are stored and the stamper waits for the corresponding event. When the input signal reaches the value of *VIL*, event_time is set to the actual time - in this example the event occurs at approximately 4.05ms, therefore the output voltage of the stamper is set to 4.05mV.

We implemented the event stamper in in Verilog-A, the corresponding code can be found in the appendix.

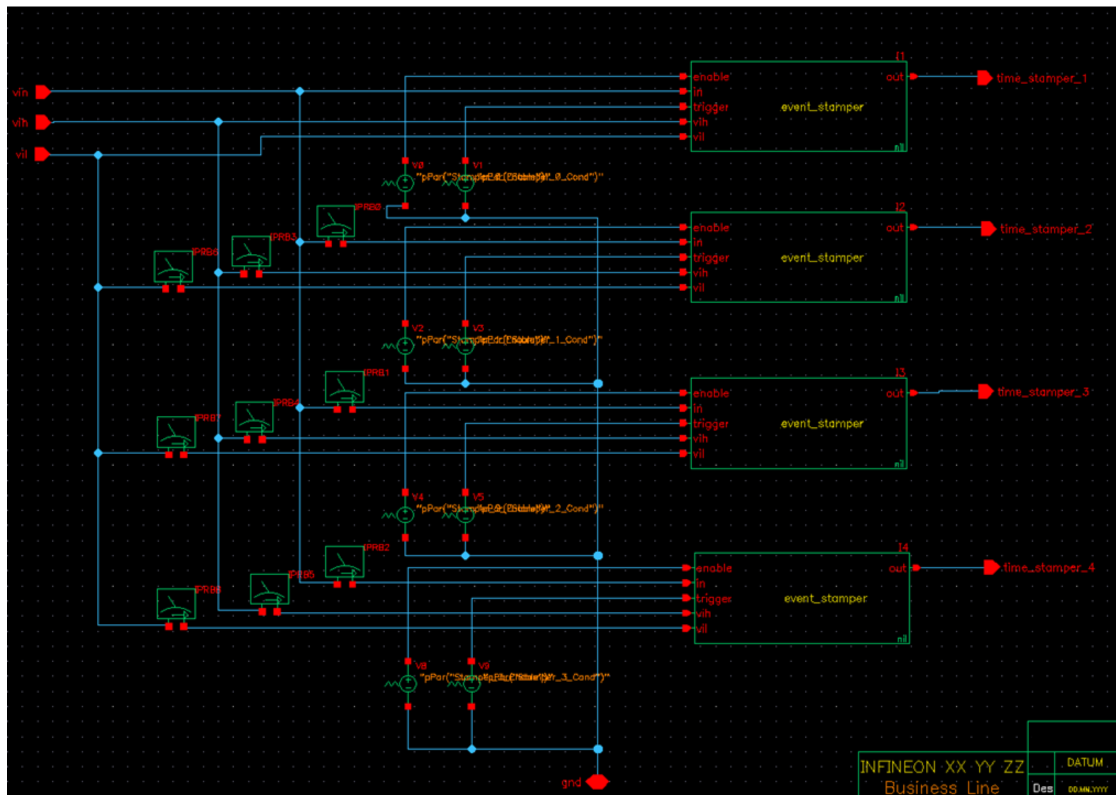


Figure 3.14: Connections of the four event stampers

3.2.4 Limit checker

Now that we are able to measure voltage, current, time and verify the correctness of logical values, we need to combine all those information for evaluating the correct functionality of the DUT. Since the limit checker is generated during the translation of an ATE test program, we just discuss its connections and high-level functionality here.

The limit checker takes all the measurement signals as inputs. Similar to the digital checker, it is triggered by a clock where every edge indicates an event. Whenever such an event occurs, either the value of a signal is measured and stored within an array or post calculations/checks are performed. Its output is 5V as long as no fault occurred and all checks were successful, in the case of an error the output voltage is set to 0V. The digital checks are not event driven, at the moment one digital checker falls down to 0V, the limit checker immediately triggers an error.

In this kind of way, the limit checker knows about every signal from every instrument, therefore also dependencies of measurements can be checked. When there are no dependencies between test instruments one might also implement the limit checker within the test instrument.

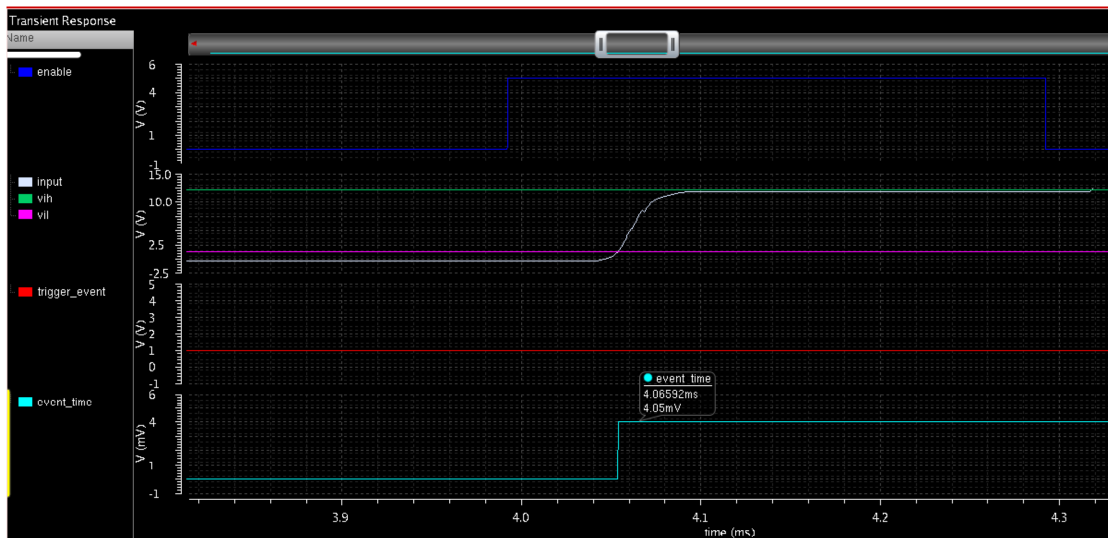


Figure 3.15: Example of an event stamper

With this discussion, we conclude the modelling part of the ATE environment. In the next chapter we show how to translate an ATE test program such that we can simulate it in this environment and verify the results automatically.

CHAPTER 4

Translation of an ATE test program

In the previous chapters, we presented the functionality of ATE tests and we showed how to model it in a simulation environment. Next, we develop a procedure for translating ATE test programs into test descriptions that can be executed in the simulation environment in the content of this chapter. We first identify the functionality that is needed in order to successfully port ATE test programs into the simulation environment. We then propose a general language in which we can express such functionality and propose the concrete translation from ATE to simulation test programs.

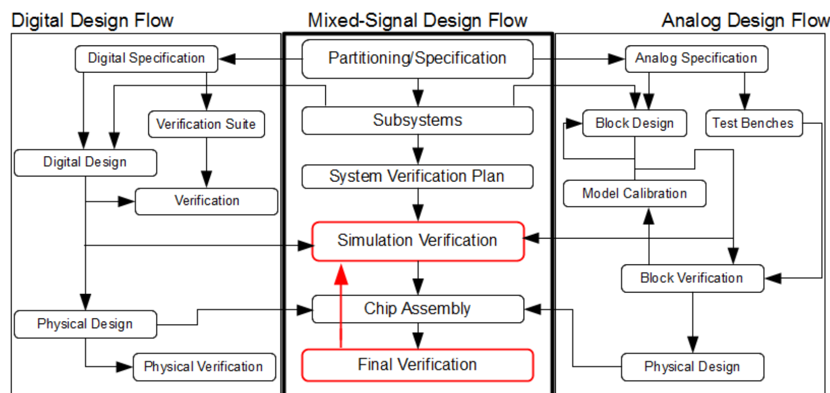


Figure 4.1: Translaton process in the design flow [BSED07]

4.1 Identification of needed functionalities

In this section, we identify the core functionality of ATE test programs that is independent from a specific ATE and is needed for their accurate translation into a simulation environment.

Figure 4.2 depicts the abstract view of the main components in an ATE test. The ATE hardware consists of a load board and the test instruments. The test program must be able to control different functionalities of these hardware components. A load board contains relays that are dynamically configured at runtime. They are responsible for dynamically setting up the DUT connections. Note that the remaining components of the load board are passive (e.g. resistors or capacities with static values), therefore they will not be discussed in this section.

The ATE test instruments are configurable by test engineers. We distinguish between analog, digital and general ATE functionality in the context of AMS design. In the previous section we showed how to model test instruments with such heterogeneous functionality.

Furthermore, communication between the ATE hardware and the test program plays an important role and must be considered. This communication typically has the following form: the test program provides a sequence of input values to the ATE hardware, one by one. In every step, the test program sends an input value, waits for the ATE to process the data and acknowledge with a special event before it moves to the next step. This functionality is represented by the Communication block in Figure 4.2.

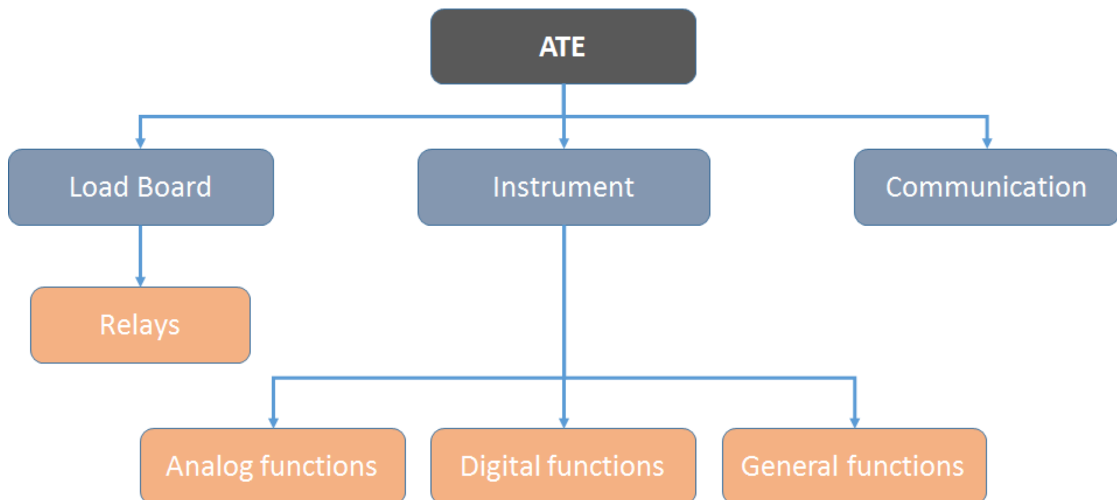


Figure 4.2: Abstract schematic of ATE

We now present specific aspects of the elements identified in Figure 4.2 and have a closer look at their provided functionalities in the sense of an abstract language.

The load board relays are controlled by an impulse (input voltage), which can have two states: on (5V) or off (0V). Note that by default, the impulse of every relay is in the off state, whereas the relay is either open or closed in this state, depending on the design of the load board. Whenever the impulse changes, the state of the relay is updated accordingly.

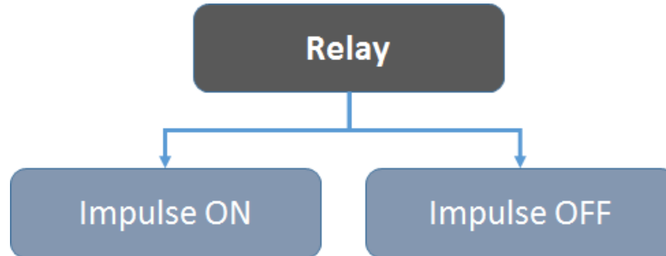


Figure 4.3: Relay functionalities

It follows that a test program can control a relay by switching the impulse on and off. This action is done by the two commands Relay_On and Relay_Off as shown in Table 4.1.

Description	Command
Turn on the impulse, which controls the relay	Relay_On
Turn off the impulse, which controls the relay	Relay_Off

Table 4.1: Relay functionalities and commands

We now focus on the control of the test instruments by a test program. We start by presenting general functionalities that are common to every test instrument, regardless of whether it is digital or analog. All test instruments (a conceptional overview was given in Figure 3.5) have high and low connections (for both, the force and the sense line) as well as a local Kelvin connection to connect the sense and the force line through a switch. An overview of these functionalities is given in Figure 4.4. The Levels block in Figure 4.4 defines the pin levels, which might dynamically change during the test run. They are used for specifying the voltage levels high and low. Furthermore they are important during a timing test for distinguishing when a specific logic value has changed. The timing itself is used for both, analog and digital signals: for an analog signal it indicates how long an output is active (e.g. set output voltage to 5V for 5us), whereas in the digital part the timing determines the length of the signals as well as the point in time when a signal has to be strobed.

Table 4.2 summarizes the commands used to connect and disconnect test instruments. Note that we will treat the commands used for defining the pin levels and the timing will be treated separately.

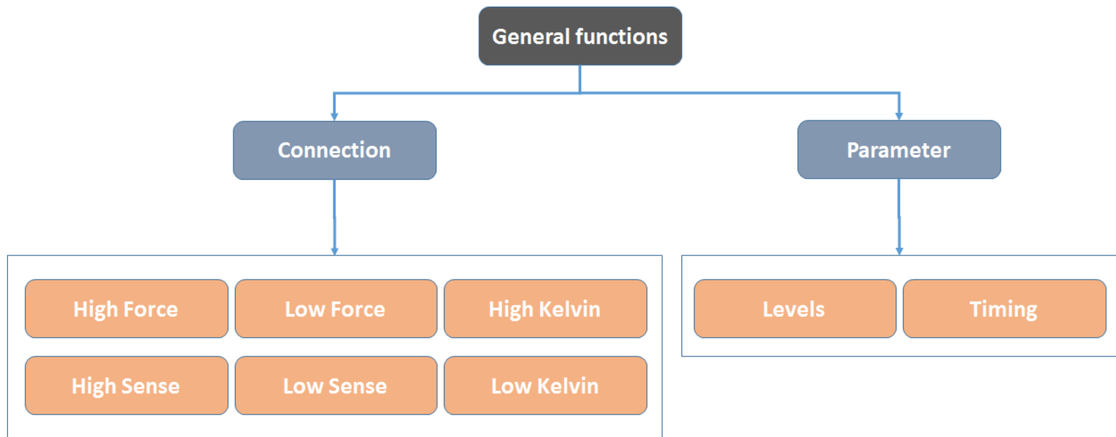


Figure 4.4: Overview about the basic general functions of an ATE

Description	Command
Connect the high side of the force line	ConnectHighForce
Connect the low side of the force line	ConnectLowForce
Disconnect the high side of the force line	DisconnectHighForce
Disconnect the low side of the force line	DisconnectLowForce
Connect the high side of the sense line	ConnectHighSense
Connect the low side of the sense line	ConnectLowSense
Disconnect the high side of the sense line	DisconnectHighSense
Disconnect the low side of the sense line	DisconnectLowSense
Close the high kelvin relay	ConnectHighKelvin
Close the low kelvin relay	ConnectLowKelvin
Open the high kelvin relay	DisconnectHighKelvin
Open the low kelvin relay	DisconnectLowKelvin

Table 4.2: Commands available on the force line

The level parameters, shown in Figure 4.5, are used at the input and the output lines in the digital part of a test instrument. The input of a digital instrument is used for specifying an expected logic value at a specified point in time. The digital signal is measured at that time instant and the recorded value is compared to the specified logic value. A difference between the specified and the measured value indicates an error in the DUT. Four different pin levels are configured for every instrument, as summarized in Table 4.3 - voltage out high (VOH), VIH, voltage out low (VOL) and VIL.

The input and the output levels are dynamic and the test engineer can control them in every digital instrument, by using the commands depicted in Table 4.3. Note that

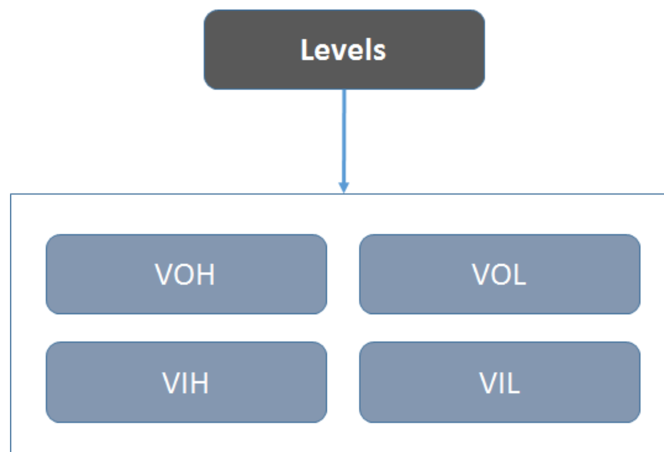


Figure 4.5: Overview of the basic properties concerning the levels

different levels can be applied to different instruments, such that it is possible to perform several digital measurements in parallel.

Description	Command
Minimum voltage for output high	SetVOH
Maximum voltage for output low	SetVOL
Minimum voltage for input high	SetVIH
Maximum voltage for input low	SetVIL

Table 4.3: Commands for configuring the pin levels

The timing block is used in both, analog and digital instruments. On the one hand the timing is used for configuring the duration of a signal (e.g. set the output voltage to 5V for 4us or force a logic 1 for the same time). Furthermore, the test engineer can control the time between the execution of two consecutive commands by setting a delay between them. This delay is determined by the period of the timing configuration. For digital waveforms, two additional informations are needed: the window and the format. The format simply defines the wire code which is used (e.g. non-return to zero or return to zero). The window is used to define an interval over which the digital measurements are made. For example, when the timing is set to 3us and the window to [200us, 250us], then the input signal is measured throughout the interval [3200us, 3250us] and compared with the specified logic value.

We now proceed with the presentation of the commands specific to the analog part of a test instrument. The analog instrument consists of a force and a sense line. As can be seen in Figure 4.7, there is a force line as well as a sense line. In general an analog instrument can be used as voltage or as current source. It has to be able to switch between these modes. Additional to the source mode, a clamp has to be applied (current clamp

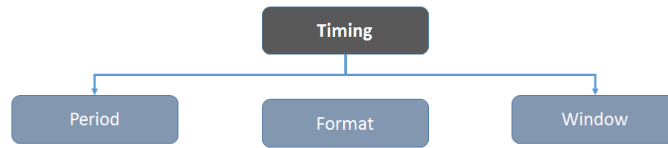


Figure 4.6: Overview about the basic timing functions of an ATE

Description	Command
Set period (corresponds to a wait statement)	Wait
Set wire code (e.g. NRZ)	SetFormat
Set Window within the timing for strobe and check input	SetTimingWindow

Table 4.4: Commands available for the timing parameter

for a voltage source and a voltage clamp for a current source) to limit the corresponding unit. Furthermore it is possible to source a whole waveform (in current or voltage mode). For this functionality, the samples have to be configured together with a sample rate and the mode. Note that the timing is used for defining a sequence of commands. Assume we apply a timing with a period of 3us to an instrument and force 0V. Next we apply a timing with a period of 10us and force a voltage of 10V. This would mean that for 3us a voltage of 0V is forced. After this 3us, the next command with the timing period of 10us is applied, yielding in an output voltage of 10V for the next 10us. This might yield to problems when sourcing a waveform: here the delay between two values is determined by the *sample_rate*. When there is a value applied in the time between two points in time of the wave, this will result in a malformed wave and therefore cause a wrong behaviour. We realize the timing period with a simple wait statement.

The other part of an analog test instrument is the measurement unit. We can distinguish between three units to be measured: current, voltage or time. Time measurements are event-driven: the test program supports different events in which the user might be interested (e.g. reaching the voltage VOH after a logic 0 was applied). We will discuss the concrete functionality and configuration of timing measurements in the next block. The hardware often includes a filter for the input of the sense line that can be enabled or disabled. We do not consider this functionality, since it is not relevant for the simulation.

Table 4.5 gives an overview about the command set, used for configuring the force line of the analog instrument. Since a whole waveform can also be sourced by applying the corresponding current and voltage values combined with a correct timing, we ignore the wave command. The clamp indicates a boundary for the voltage (in case of a current source) or the current (in case of a voltage source). When the corresponding unit reaches this boundary value, an alarm would be raised indicating an error. In some cases, it is allowed to disable this alarm, when it is possible to reach this value. When for instance the current clamp of a voltage source is exceeded, the instrument will switch to a current source, sourcing the value of the current clamp.

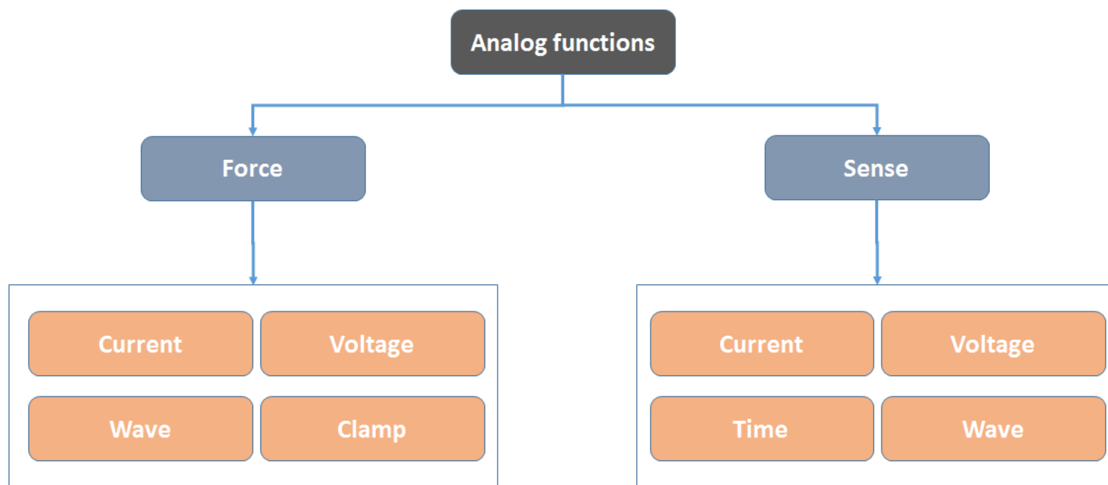


Figure 4.7: Overview about the basic analog functions of an ATE

Description	**Command**
Force current on the force line	ForceCurrent
Force voltage on the force line	ForceVoltage
Force a given waveform (e.g. Ramp) in voltage or current mode	ForceWave
Set voltage or current clamp	SetClamp

Table 4.5: Commands available for the source

The commands for configuring the sense line are summarized in Table 4.6. Note that in some testers, metering current or voltage means that the corresponding value is stored in the memory of the test instrument and has to be read separately. We assume that the value is immediately transferred to the test program. Furthermore, aggregation functions such as averaging multiple samples can be realized with a restricted set of basic commands, hence we ignore them in this thesis.

When the clamping value is set, it is automatically treated in the expected way - when the instrument is configured as voltage source it will be used as current clamp, otherwise as voltage clamp.

When considering timing measurements, some more information needs to be available which is shown in Figure 4.8. Timing measurements are event triggered based on a global clock that starts running at the beginning of the test program. When the timing measurements are enabled, the measurement unit waits, until the configured event occurs. Examples of interesting events during such a timing measurement are shown in Figure 4.8. Whenever such an event happens, the time is stored in the instrument and can be read by the test program. Furthermore we will introduce a command to force a timing measurement, such that timing differences between the current point in time and the

Description	Command
Configure meter to current mode	MeterCurrent
Configure meter to voltage mode	MeterVoltage
Get the time of a measured event (e.g. Leaving Low)	GetEventTime
Set voltage or current clamp	GetWave

Table 4.6: Commands available on the sense line

time of the event can be calculated.

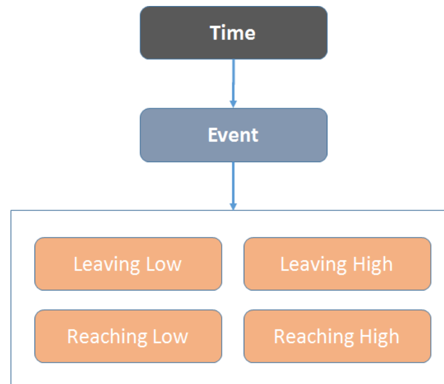


Figure 4.8: Overview about the basic timing functions of an ATE

Table 4.7 gives an overview of the commands, needed for controlling the timing measurement unit. First of all, the event the user is interested in has to be defined. As shown in Figure 4.8 one can distinguish between:

1. Leaving Low: The voltage level of the input signal becomes greater or equal to VIL
2. Reaching Low: The voltage level of the input signal becomes lower or equal to VIL
3. Leaving High: The voltage level of the input signal becomes lower or equal to VIH
4. Reaching High: The voltage level of the input signal becomes greater or equal to VIH

The timing measurement has to be enabled to force the instrument to wait for the desired event. When the event occurs, the actual time of the global clock is stored in the memory and can be read at some point in the test program. We note that one event stamper per instrument may not be sufficient in some cases, hence the test equipment can contain multiple stampers in every instrument as shown in Section 3.2.3. It is necessary to enable and set the events of the stampers separately (they can also be triggered on different events).

Description	Command
Set the event at which time is sampled	SetTriggerEvent
Enable event stamping	EnableStamper

Table 4.7: Commands for configuring the event stampers

The parts which have to be taken into account for the digital part of the test instruments are shown in Figure 4.9. Similar to the analog part, also the digital instrument consists of an input and an output. The output is responsible for sourcing a digital logic value based on the actual levels and the timing. On the other hand, the input part samples a value and performs a check on it. As a consequence, the test engineer is able to define an expected value (high or low) and during the window phase inside the actual timing it is checked whether the input is equal to this value - if not, an error occurs.

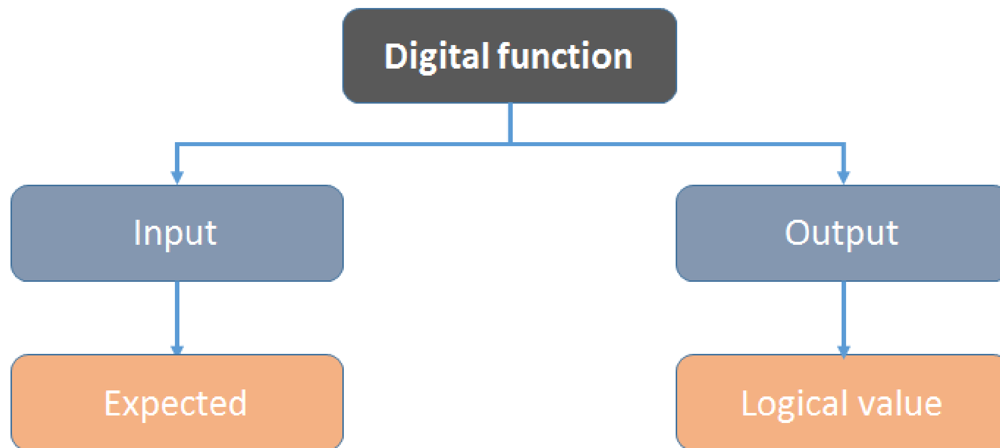


Figure 4.9: Overview about the basic digital functions of an ATE

Table 4.8 shows that two commands are needed to control a digital test instrument: one for defining the expected value, as well as defining the output value. Note, that apart from 0 and 1 also X can be used, meaning *don't care*. When the expected value is X there will just be no check of the input signal and the output of the test instrument is set to high-impedance (current source with 0A). The output will keep its last value.

Description	Command
Set the expected logic value (high or low)	SetExpectedValue
Set the logic value of the output	SetLogicOutput

Table 4.8: Commands available on the digital sense line

Specific testers may often support additional functions to the ones considered in this document. In this thesis, we only consider commands that are relevant for the translation

of ATE test programs into the simulation environment. In addition, ATEs often support test patterns that are executed in parallel with the test program and synchronized for ensuring a deterministic test. The functionalities for the communication between pattern and test program are in the block Communication in Figure 4.2. Since, the pattern is very similar to a function call and only deterministic tests shall be considered here, this functionality is neglected to keep the language simple.

Finally, test programs may support conditional statements in order to accommodate to test device variations.

4.2 General test program language

We assume that ATE test programs are written in a general *Turing Complete* programming language. We do not require any specific syntax, and use pseudo-code where appropriate.

As already mentioned, one part of a test program are the so called test patterns. Such a pattern defines line by line a test vector consisting of the timing to use, the digital configuration (output and expected logic value) as well as the analog stimuli in the form of a voltage/current value or of a waveform description, consisting of a sequence of timestamp/value pairs. Test patterns also use conditional statements and loops. We note that patterns are executed in parallel to the rest of the test program. Despite this parallelism, we assume deterministic tests. This is a reasonable assumption - non-determinism in ATE test execution is an undesirable feature that is avoided in practice by proper synchronization primitives.

During the run of an ATE test program, signals are measured and their result is compared to a bound defining the minimum and maximum allowed value. For automatic detection of a limit check, it is necessary to introduce a separate command. As a consequence, the *CheckLimit* command, parameterized with an input variable and high/low limits, is used for checking the limits during a test execution.

The actual limits used in *CheckLimit* are not always simple constraints fetched from the limit specification table. It turns out that the specified limit values are often pre-processed using different arithmetic operators before being used for the limit checking. We will see in the next section that this preprocessing of specified limits is problematic for the translation of ATE tests to the simulation environment.

The last problem, which remains is how to specify which test instrument shall be configured when calling a specific command. Every instrument has a unique name such that this name is also known by the ATE (the test instruments are uniquely identified by the ATE by their positions). Every command takes as parameter the name of the instrument which shall be configured. Additionally these commands take the necessary information (e.g. high level voltage and low level voltage) as parameters. Furthermore, the question is, when will such a command be applied (e.g. when the sourcing mode changes, the clamping value might not be up-to-date and might cause an error in the test). Therefore we introduce the *Apply* command, which will apply the currently configured

parameters to the instrument identified by the name. So it is possible to send a whole set of changing parameters to the test instruments without introducing secondary effects.

4.3 Translation process

After completing the definition of a general ATE test program language, we now propose an automated procedure for translating ATE test programs into a format supported by simulation environments. We note that a precondition is the support of an offline debugging by the tester, such that it is possible to run the test program before the first silicon (otherwise one loses the advantage of using the translation early in the design flow).

Figure 4.10 depicts a high-level overview of the translation process. The translation procedure takes as input the ATE test program and a configuration file. As we have shown in the section before, the test program consists of several commands, which are relevant for the simulation. In a first step, we need to identify which of these commands are executed during the execution of the ATE test program and print this program flow into an intermediate file. Therefore, after every relevant command, a line is added which prints the command of interest into the intermediate file. This approach is called *code injection*. The adapted test program is then executed in the debugging (offline) mode of the tester. After this execution, we have an intermediate file which contains all the commands which were really executed. As highlighted with the grey box in the figure, the generation of the intermediate format is not part of this thesis and used from [KRT02] with small changes in the format. We use this intermediate file as input for the parser and translate this program flow for the simulation environment.

The points of manual interaction have been highlighted in the figure:

1. ATE test program: The ATE test program has to be written by the test engineer for the *final verification* in the design flow and is taken as input.

2. Debugging run: Debugging has to be executed manually by the user, to write the relevant commands into a file.

3. Configuration file: The user has to provide a configuration file with information about the simulation environment such that it can be configured automatically.

In the next sections, these single parts of the translation flow will be discussed separately in more detail. Note that the limit checker can of course be implemented in an arbitrary language (Verilog-A, VHDL-AMS,...), it just has to be supported by the simulator. In this thesis, we encode the limit checkers as state machines expressed in Verilog-A.

The intermediate file filters the test pattern and contains only statements (sequences of commands) that are relevant for executing the test in the simulation environment. In addition, it contains the set of calculations that are performed during the test run.

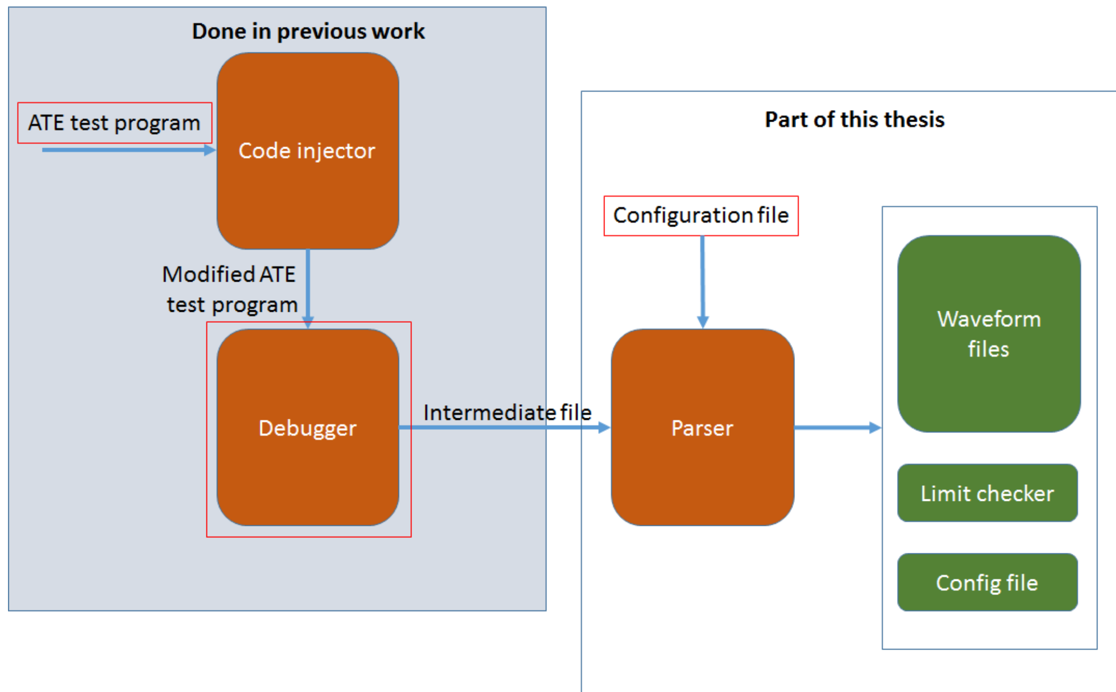


Figure 4.10: Translation process for an ATE test program

The last two pieces of information are necessary for keeping track of how variables are calculated such that post calculations can be simulated in the context of the limit checker.

4.4 Code injection

In many cases, a test program is not just implemented for one single device, but for several versions (e.g. 4 channel and 2 channel devices). Therefore a lot of the test program is similar for these two models and it won't make sense to implement two very similar test programs. Instead, the test engineer uses conditional statements for distinguishing between these two models when it comes to specific parts in the test program. The test engineer is then able to select the model to be checked in the *graphical user interface*.

Translating a test program into a format supported by the simulation environment is challenging to do with a parser, due to the lack of information generated during the test runtime. Another possibility would be to use on-the-fly translation of the test program during its execution with an interpretor. However, this solution may be too expensive for complicated test patterns.

To overcome this challenge, the idea which is introduced within [KRT02] is code injection. We got this generated intermediate file from Infineon Technologies as input for our parser. From the previous section, it is already known, which commands of the tester are relevant for the simulation environment. This knowledge can now be used for doing

some preprocessing, before the test program is executed. The goal is to instantiate a test pattern to a specific configuration, thus getting rid of conditional statements and loops, resulting in a simple sequence of commands that are really executed for that particular configuration.

During the code injection, the preprocessor goes through the files of the test program line by line as shown in Figure 4.11, searching for the relevant commands including tester commands, variable definitions (local and global) as well as variable calculations. Whenever such a line is found, the associated command with its parameters is inserted to the intermediate file. The program simply runs until all lines have been processed.

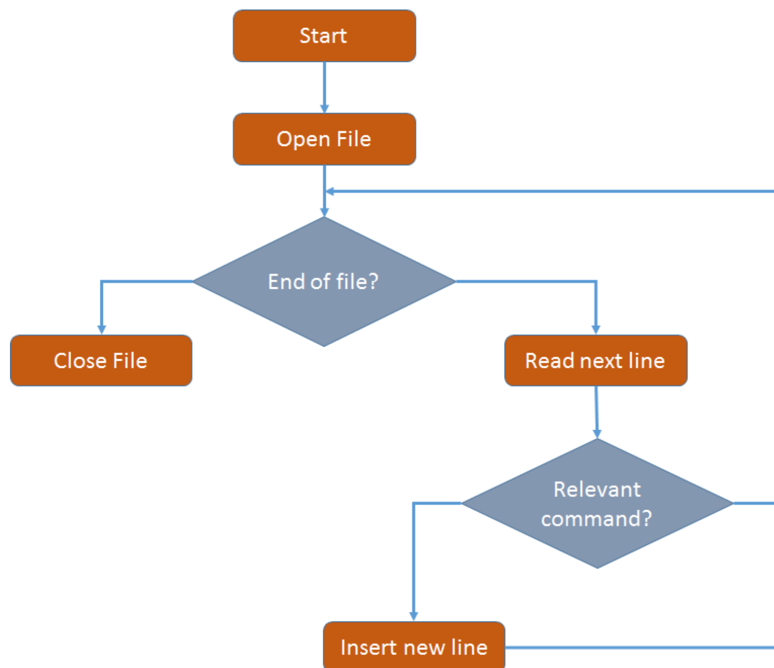


Figure 4.11: Flow diagram for the code injections

After the test program has been preprocessed, the user has to start the offline debugger, which is able to simulate the test program without a device (of course the limit checks will fail without responses from the DUT, so they have to be ignored). During this run, also the newly added output commands will be executed leading to an intermediate file including all the information of interest which was executed. Note: since different testers of course use different languages for configuration, one has to change the commands of interest! They should be mapped to the general language introduced in the previous sections such that the parser, which will be used in the next step needs not to be adapted.

In order to improve the reusability of this method, the intermediate file uses a general encoding that is not specific to particular equipment or simulation environments.

We finally note that many test equipments support a network interface for the communi-

cation between ATE and test program over which the commands are sent. Therefore, an improvement against this static code injection would be to read the executed commands over the network interface and process them according to their meaning.

4.5 Parsing of the intermediate file

After filtering the test pattern and generating the intermediate file, we need to parse it and adapt it to the simulation environment by:

1. Provide the configuration files for the test instruments in the simulation environment.
2. Provide a state machine in a chosen language for the limit checker.
3. Provide a configuration file for setting all the parameters in the simulation environment automatically.

We remind the reader that every instance of a test instrument or a relay is uniquely defined in the simulation environment by its name, as discussed in Section 3. For configuring such a test instrument, the path of the used input file is taken as parameter. Normally, the user would have to set all the filenames in the simulation environment manually. For preventing this issue, many simulation programs provide a scripting language (e.g. SKILL in Cadence) such that these parameters can be set automatically, preventing typos and other undesired faults. The needed input files are created during the translation process by the parser automatically. For relating a file to the corresponding instances, the user has to provide this information in a configuration file: a mapping between instrument names used in the test program to the instance which corresponds to this instrument in the simulation environment. The parser therefore automatically creates a script, which can be imported by the simulation program. This has to be done for the instruments as well as for the relays.

We next continue with the generation of the input files for the simulation environment. The target is to translate the entire command sequence used in the test program to simple waveforms which can then be sourced by the test instruments. Furthermore it has to be possible to switch between the current and the voltage source and configure the switches which are used in the model. So let's first give an overview about the different files, which have to be generated for the simulation environment.

We use the file format we introduced in Chapter 3 for controlling the whole simulation environment from external, including all the voltage supplies for controlling the switches and used voltage/current source for the instrument. For configuring a relay, just one file is necessary - the file defining the waveform for the voltage source which is controlling the switch (0V means open, 5V is closed). For an instrument a set of files must be generated, controlling all the parts of the instrument, including:

- Voltage Source
- Current Source
- Switch of the voltage source
- Switch of the current source
- Switch for channel A high (force)
- Switch for channel B high (force)
- Switch for channel A low (force)
- Switch for channel B low (force)
- Switch for channel A high (sense)
- Switch for channel B high (sense)
- Switch for channel A low (sense)
- Switch for channel B low (sense)
- Switch for high kelvin
- Switch for low kelvin
- Expected value for digital check
- Trigger signal for digital check
- Voltage for logic high
- Voltage for logic low
- Enable signals for all event stampers
- Trigger events for all event stampers

In the target path (which is specified in the configuration file), a folder structure must contain a folder for every test instrument including these files and a separate folder for the relays. The purpose of the parser is to perform the translation between the intermediate file and these set of files automatically, such that all the instruments are configured.

Therefore in a first step all the files are generated out of the list of instruments given in the configuration file (through the instance mapping) and initialized with the first line which is mandatory: 0 0. Afterwards parsing starts, where every single command, which is of interest, is mapped to one of these files with the correct wave information.

Note that all the commands take as parameter the instrument on which they shall be applied. The user can define a whole block of commands (consisting of a timing, sourcing value etc.) followed by an apply command when the last block shall be applied to the instrument. After an apply command, the parser executes the commands (including keeping track of the time).

4.6 Creating the limit checker

In this section we have a closer look at the construction of the *limit checker*. This block is responsible for verifying the correct behaviour of the DUT based on the measurements from the test instruments. The *limit checker* must execute all the post calculations from the ATE test program. As we mentioned in Chapter 2, the limit values are defined in a *Limit Sheet*. This sheet consists of a list for every test, where an element of this list consists of a higher limit value as well as a lower limit value. During execution of the ATE test program, the *CheckLimit* procedure is called, using a variable and the limits as parameter. Whenever this procedure is called, we compare the value with the next

element from the list. We use this information to design a *finite state machine*(FSM), which performs the limit checking.

The first step is to define trigger points for this state machine, similar to a clock signal in synchronous designs. In our abstract language, several commands (e.g. MeterCurrent or MeterVoltage) are available for performing a measurement and getting the result. Whenever we execute one of these commands, new calculations are executed - therefore we use these command set to define the trigger points (events) of the limit checker. An event is either a rising or a falling edge. Every time, a command of this command set is recognized, we generate such an event (according to the previous event). For this signal, we use the same file format as introduced in combination with a VPWL.

Assume the test engineer executed a MeterCurrent command at some point in time and stores its value in a variable. This means that we need to insert a new state in the limit checker in which the same measurement is executed and stored in a variable during the simulation. Furthermore, we must keep track about these variables. Whenever we detect such a measurement, the corresponding variable name is stored in a data structure (e.g. Hash Map). At the moment, the parser recognizes a new calculation, it checks whether the used variables occur in the Hash Map or not. Variables, which do not occur in the Hash Map have a deterministic value since they are not related to any measurements (we exclude the case of randomness because we assume that the test is deterministic). Therefore, this kind of variables is replaced by their value.

The other case is to find a calculation, which uses such a measurement variable found in the Hash Map. This means, its value is non deterministic. Therefore, we need to add this calculation to the state machine to the actual state - the deterministic values are again replaced. Moreover, the new variable is added to the Hash Map.

When we encounter a *CheckLimit* command, the check of the variable taken as parameter is added to the actual state. Note that for the moment we neglect the occurrence of global variables. When they are introduced, we need to use one map for local and one for global variables to distinguish between them. Also procedure calls must be taken into account within a calculation - here the procedure call simply has to be replaced with the return value during the calculation.

A flow diagram of the whole code injection flow with a limit checker included is illustrated in Figure 4.12. We now distinguish between two different files: one is used for implementing the FSM (e.g. in Verilog-A) and another one acts as the intermediate file as before. Furthermore, calculations are added to the set of relevant commands - whenever a calculation is recognized, we check, whether it is relevant to be added to the state machine or not. Note that calculations are always added to the actual state. Measurement commands are handled in the same way, additionally they just trigger a new event for the state machine. All the other commands are handled in the same way as before and will be written into the intermediate file.

What remains to show is the state machine itself which is used for the limit checker.

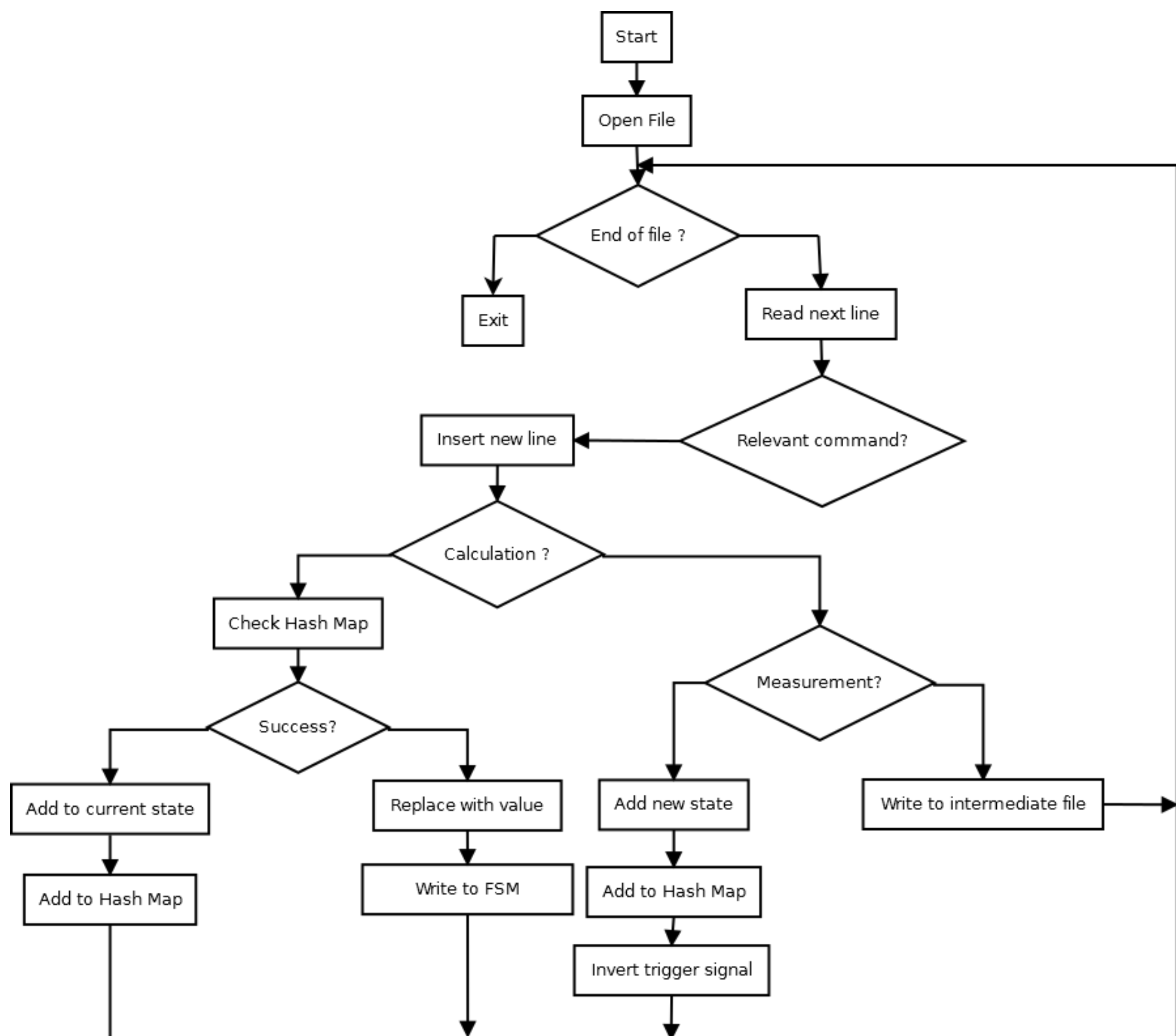


Figure 4.12: Flow diagram with limit checker included

Figure 4.13 gives a rough overview of how the limit checker is supposed to work. The checker uses all the measurements done by the instruments as inputs, including voltage, current and timing measurements. Whenever a clock event occurs (either a rising or a falling edge), the desired input is read and stored in a variable. Since all the post calculations from the code, occurring after the measurement, are included in that state, they are immediately executed. In the case that also a CheckLimit command has to be executed before the next measurement command, it is added to the same state. Therefore, after a clock event, three steps are executed: reading the measurement(always), post calculation and limit check (both optional).

Now we extend this simple state machine as illustrated in Figure 4.14. Since after

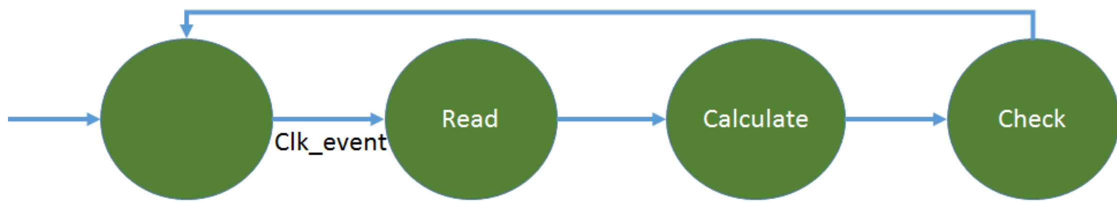


Figure 4.13: Schematic of the functionality of the limit checker

every measurement different post calculation steps might be executed, we need to keep track about this measurements - this is done by the new state introduced by every measurement command. Therefore a counter indicating the current state is increased after every clock event, yielding in the possibility of varying the post calculation steps after every measurement.

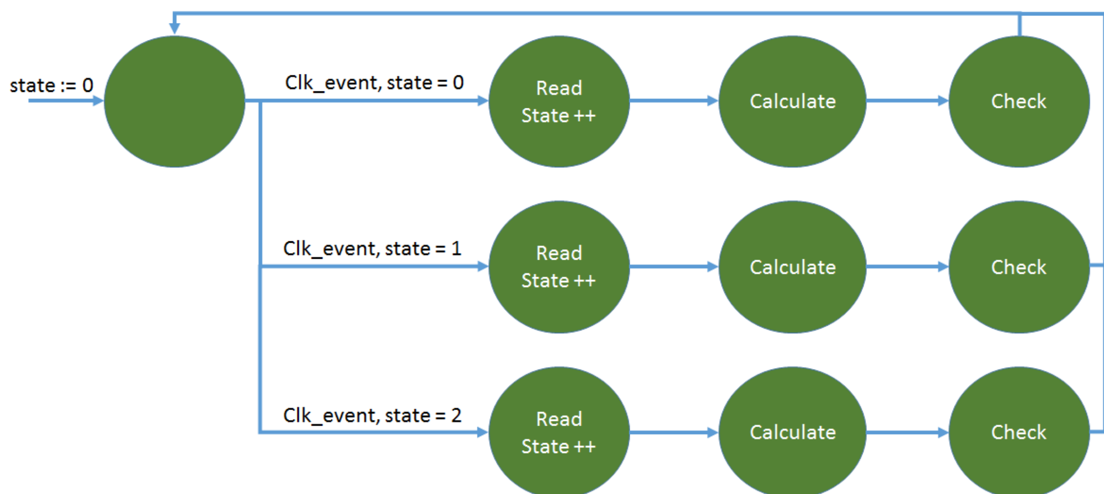


Figure 4.14: Schematic of the state machine for the limit checker

Note that the limit checker has also to distinguish between the test cases, since different test cases use different limit sets. To overcome this issue, another input, indicating the currently executed test case has to be added.

What should be mentioned is that in several programming languages in which an ATE test program might be implemented, the module size can be restricted. This could lead to problems with the limit checker, since we must keep track on every single variable and calculation, yielding in a lot of code to inject. These injections might result in an excess of the module size - the program cannot be executed any more. This should be kept in mind during implementation such that the calculations are outsourced into several modules. Note that another possibility for automatically creating the limit checker is the usage of *signal temporal logic* as introduced in [JBG+15]. With some modifications, it would be possible to implement a similar solution in Verilog-A.

4.7 Example translation for the continuity test

In this section we apply the proposed approach to the code of the continuity test shown in Chapter 2. We do not show the step of the code injection, instead we start with the output of the debugging phase. Recall that in a first step the relevant commands are identified inside the code and an output line is inserted next to them, before we run the new code in the debugging mode of the tester.

The code from the intermediate file is shown in Listing 4.1. At first, all the instruments and relays are connected. The wait statements are needed for ensuring that all the signals are set before applying new waveforms. Note that also in the simulation environment such waiting times are needed, insufficient delays might yield in convergence problems.

Listing 4.1: Setup of a continuity test

```
Call Relay_On("OUTxF_k")

ConnectHighForce("Vs_dc30")
ConnectHighSense("Vs_dc30")

ConnectHighForce("IS_dc30")
ConnectHighSense("IS_dc30")

ForceVoltage("Vs_dc30", 0)
SetClamp("Vs_dc30", 0.1)
Apply("Vs_dc30")

ConnectLowForce("all_dc90_lo_a")
ConnectHighForce("all_dc90_hi_a")
ConnectHighSense("all_dc90_hi_a")
ConnectLowSense("all_dc90_hi_a")

ConnectHighForce("all_hsd")
ConnectHighSense("all_hsd")

DisconnectHighKelvin("Vs_dc30,IS_dc30,all_dc90_lo_a")
DisconnectLowKelvin("Vs_dc30,IS_dc30,all_dc90_lo_a")

ForceVoltage("all_hsd", 0)

ForceVoltage("Vdd_dc30", 0)
SetClamp("Vdd_dc30", 0.1)
Apply("Vdd_dc30")

ConnectHighForce("Vdd_dc30")
ConnectHighSense("Vdd_dc30")

DisconnectHighKelvin("Vdd_dc30")

ForceVoltage("IS_DC30",0.0)
SetClamp("IS_DC30",0.0002)
ForceVoltage("VDD_DC30",-0.1)
SetClamp("VDD_DC30",0.1)
ForceCurrent("Vs_DC30",-0.001)
SetClamp("Vs_DC30",-1)
ForceVoltage("Vs_1Alo",0)
SetClamp("Vs_1Alo",0.0002)
ForceVoltage("Vs_2Alo",0)
SetClamp("Vs_2Alo",0.0002)
ForceVoltage("Vs_3Alo",0)
SetClamp("Vs_3Alo",0.0002)
ForceVoltage("Vs_4Alo",0)
SetClamp("Vs_4Alo",0.0002)
Wait(0.000001)

Wait(0.000001) # Repeat this line 100 times
```

```
Wait(0.000001) # Repeat this line 3000 times

Wait(0.000001)
Wait(0.000001) # Repeat this line 3000 times

# Repeat these lines 5 times
MeterVoltage("Vs_DC30")
Wait(0.000010)

ForceCurrent("Vs_DC30", −0.00005)
SetClamp("Vs_DC30", −1)
Wait(0.000001)
Wait(0.000001) # Repeat this line 3000 times

# Repeat these lines 5 times
MeterVoltage("Vs_DC30")
Wait(0.000010)

SetVoltage("Vs_DC30", 0)
Wait(0.000001)
Wait(0.000001) # Repeat this line 500 times

ForceCurrent("IS_DC30",0.0001)
SetClamp("IS_DC30", 1)
ForceCurrent("VDD_DC30",−0.0001)
SetClamp("VDD_DC30",−1)
ForceVoltage("Vs_DC30",0)
SetClamp("Vs_DC30",0.1)
ForceCurrent("Vs_1Alo" 0.00005)
SetClamp("Vs_1Alo", 1)
ForceCurrent("Vs_2Alo" 0.00005)
SetClamp("Vs_2Alo", 1)
ForceCurrent("Vs_3Alo" 0.00005)
SetClamp("Vs_3Alo", 1)
ForceCurrent("Vs_4Alo" 0.00005)
SetClamp("Vs_4Alo", 1)

Wait(0.000001) # Repeat this line 500 times

gnd_vs = (MeterVoltage(VS_DC30) +
          MeterVoltage(VS_DC30) +
          MeterVoltage(VS_DC30) +
          MeterVoltage(VS_DC30) +
          MeterVoltage(VS_DC30)) / 5;

CheckLimit(VS_DC30, −0.7, 1000);
```

Now that we have got the intermediate file in our general language, we can translate it into the input format (waveforms of the VPWLFs) of the simulation environment - they are generated by the parser for the intermediate file. Here, we simply focus on the instrument VS_DC30 and show the configuration files for it. The others are generated in the same way. Note that the files of VS_DC30 which are not shown here, are set to 0V at every point in time.

Here we show, how the files generated by the parser look like for this instrument. They exactly define the waveforms used for the connections, the sources and for the limit checker. The connections within the instrument VS_DC30 are shown in the Listings 4.2, 4.3 and 4.4. During the continuity test, the instrument acts as a current source - the current is shown in Listing 4.5.

Listing 4.2: Switch of the current source

```
0 0
9.0618e−5 0
9.0619e−5 0
```

```
9.062e-5 0
9.067e-5 0
0.000109832 0
0.000110832 5
0.003291856 5
0.003292856 5
0.006362871 5
0.006363871 0
```

Listing 4.3: Channel A connect

```
0 0
8.97e-5 0
8.98e-5 5
```

Listing 4.4: Sense AHI connet

```
0 0
8.99e-5 0
9e-5 5
```

Listing 4.5: Current source

```
0 0
9.0618e-5 0
9.0619e-5 0
9.062e-5 0
9.067e-5 0
0.000109832 0
0.000110832 -0.001
0.003291856 -0.001
0.003292856 -0.00005
0.006362871 -0.00005
0.006363871 0
0.006863877 0
0.006864877 0
```

During the translation, also the limit checker for the continuity test is generated. The corresponding Verilog-A code is shown in Listing 4.6. Note that we show just the state machine. Every time the trigger signal crosses the value 2.5V, an event is recognized and calculations, measurements or checks are performed. In case of an error (e.g. the average of the measured voltage is less than -0.7V), the output voltage is set to 0V.

Listing 4.6: Limit checker code

```
  @(initial_step) begin
    output_value = 5;
  end

  @(cross(V(clk) - 2.5, 0)) begin
    case(substate)
     0:
               buffer_vs_dc30[0] = V(vs_dc30_voltage);
     1:
               buffer_vs_dc30[1] = V(vs_dc30_voltage);
          2:
               buffer_vs_dc30[2] = V(vs_dc30_voltage);
          3:
               buffer_vs_dc30[3] = V(vs_dc30_voltage);
          4:
               buffer_vs_dc30[4] = V(vs_dc30_voltage);

                    gnd_vs = (buffer_vs_dc30[0] +
                                        buffer_vs_dc30[1] +
```

```
                                    buffer_vs_dc30[2] +
                                    buffer_vs_dc30[3] +
                                    buffer_vs_dc30[4]) / 5;

                    if (gnd_vs < −0.7) output_value = 0;
                    end
            substate = substate + 1;
            endcase
    end

    V(out) <+ output_value;
```

When we run this simulation, the corresponding signals are sourced and the functionality is automatically verified. The waveforms for the connections of VS_DC30 are shown in Figure 4.15 whereas the sourced current is illustrated in Figure 4.16 together with the resulting voltage.

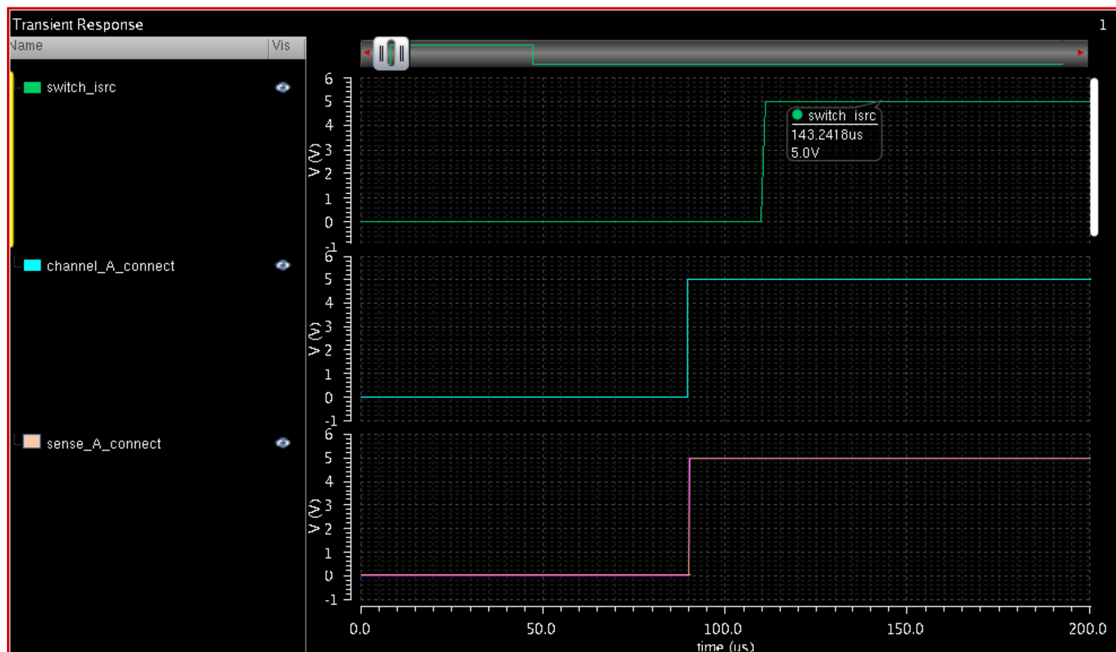


Figure 4.15: Connections of VS_DC30 for the continuity test

In Figure 4.17 we show the trigger signal for the continuity test together with the measured voltage (from the voltmeter of VS_DC30). The voltage is about -520mV at every measurement, therefore the behaviour of the DUT is correct and no fault was recognized.

This example just gives a small overview of the translation process for mainly analog signals. In more complicated test cases (e.g. *timing test*) also digital signals/checks as well as timing checks are implemented. They can be read and checked by the limit checker in the same way. Just keep in mind that digital signals significantly increase the simulation time due to the number of transitions as we show in the evaluation section.

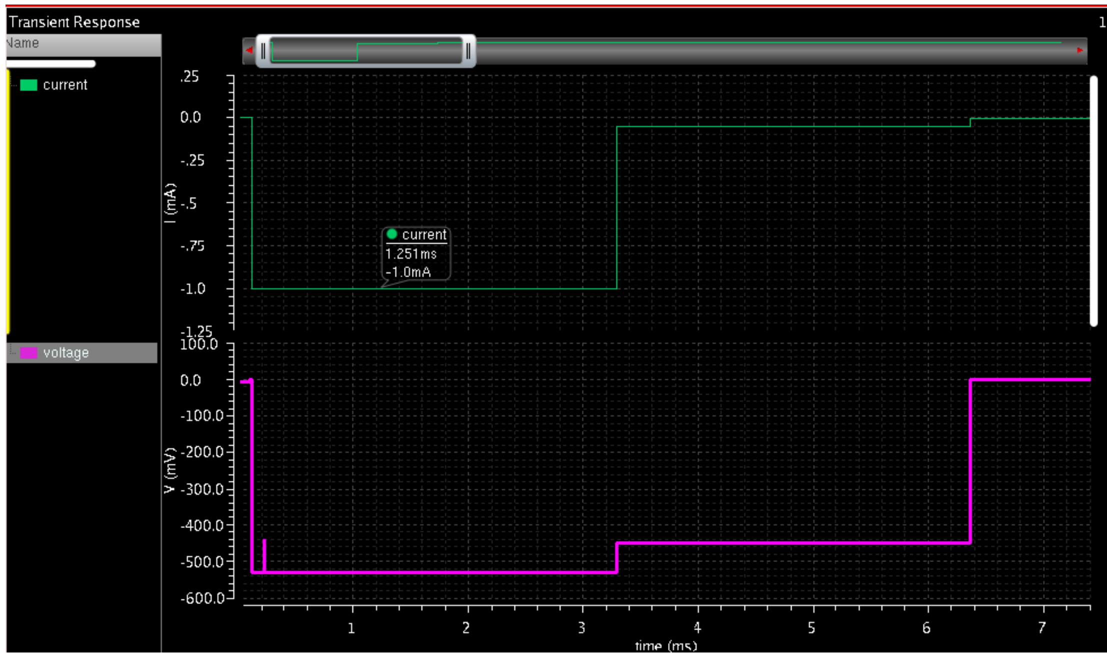


Figure 4.16: Forced signals by VS_DC30 for the continuity test

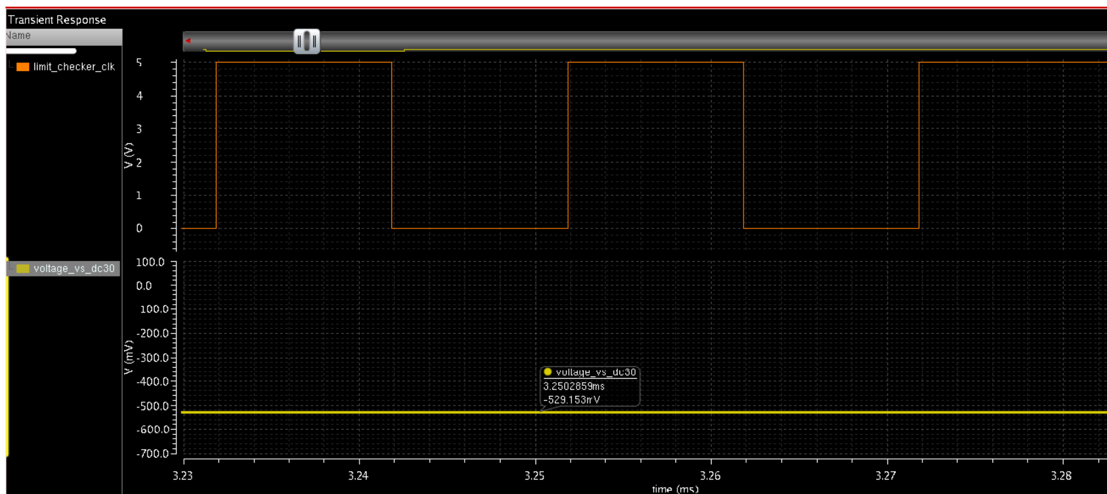


Figure 4.17: Limit checker of the continuity test

Furthermore it needs to be mentioned that the limit checker was generated manually by the approach introduced in this thesis. Caused by the limited module size of *Visual Basic* it is not possible to use the *code injection* for variables and calculations. A possibility to overcome this issue, would either be the usage of the network interface of an ATE tester to figure out which commands are executed or setting up an interpreter for *Visual Basic*.

With this example, we conclude the translation of an ATE test program. In the next

chapter we show the evaluation of this approach.

CHAPTER 5

Evaluation

In the previous chapters, we gave an overview of how to parse an ATE test program such that we can execute it in our modelled simulation environment. What remains is the evaluation concerning *simulation time* and *fault detection*. In the first section, we describe the concrete test setup and which tests were used for the evaluation. Afterwards we continue with the evaluation of the simulation times for the single tests, before we conclude with a concrete fault injection which needs to be detected by the test setup.

5.1 Setup

The top-level schematic we used for the evaluation is shown in Figure 5.1. In the middle, one can see the block, containing the *load board* which is connected to the different test instruments around it. Although we modelled one general instrument, some of them are mainly used for analog signals while other ones are used for digital signals or timing measurements, depending on how they are used in the test program. The tester used for the *final verification* is Teradyne's Flex Tester. On the right hand side of the schematic, we have the limit checker with the needed input signals - they are connected via *connection by name*. A detailed description of the models is given in Chapter 3. Note that the top-level schematic as well as the load board need to be exchanged for other versions of the DUT. The DUT used in our simulation is an Infineon chip. We used *spectre* for the simulation. Furthermore, caused by the lack of large module sizes in *visual basic*, the code injection is very limited, therefore we needed to implement the limit checker manually in Verilog-A, following the description in this thesis.

As in the previous chapters, we neglect the concrete modelling of the load board since it does not influence the simulation time in a relevant manner. In general, the whole ATE setup, does not influence the simulation time significantly. Table 5.1 gives an overview of the tests we used for the verification. Note that just an excerpt from the whole ATE test program is shown. We divided this extraction into several independent parts. The

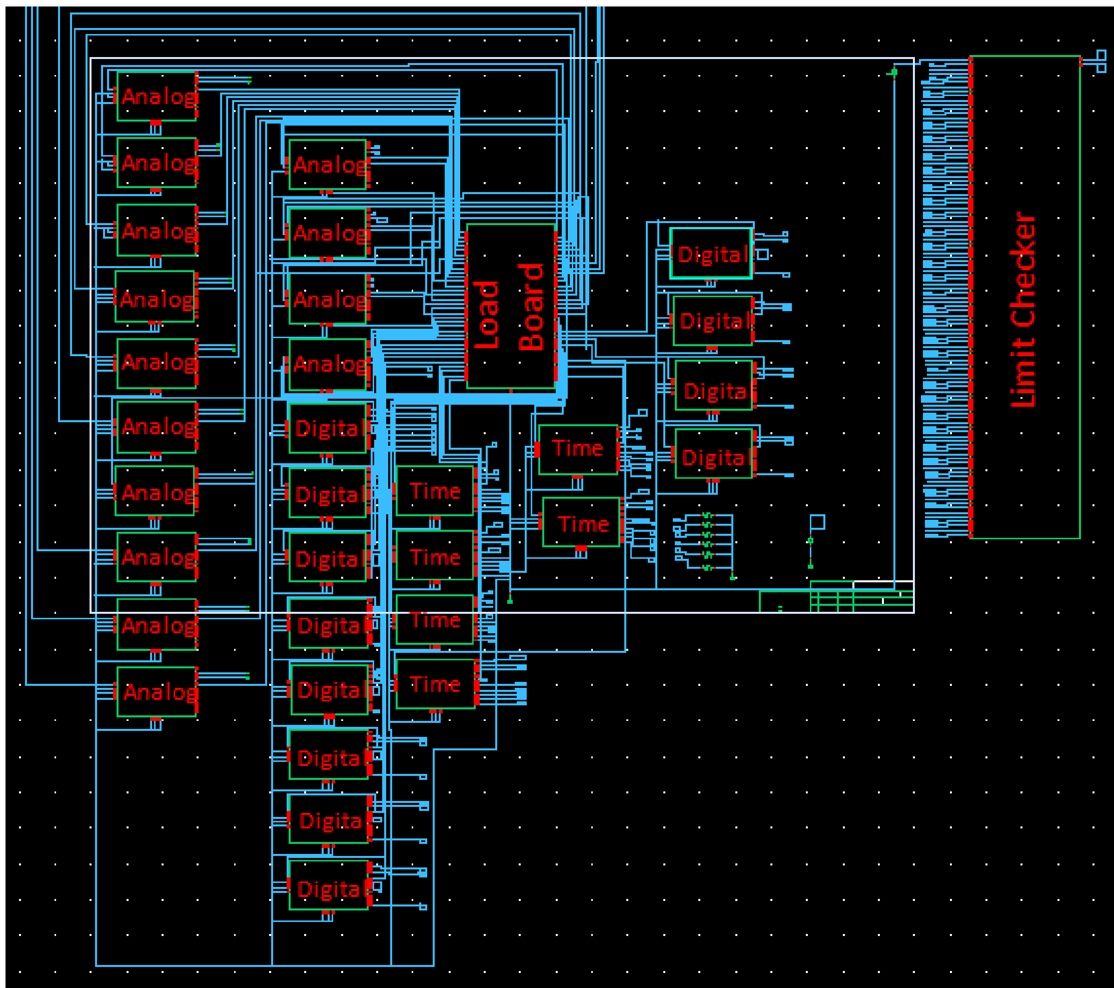


Figure 5.1: ATE test setup

columns give the number of analog and digital transitions applied by our test instruments during the given part. Furthermore we can have a look at the execution time of a specific part needed on the real tester, which also influences the simulation time. Last but not least, we note that the digital transitions are dominating in most of the parts. In the next chapter we have a closer look at the simulation times of the different parts to get an idea which factors increase the simulation time.

We note that Parts 1, 2 and 5 don't contain so many digital transitions as the parts 3 and 4. Therefore we expect lower simulation times for these mainly analog parts of the test program.

In general there might be dependencies between the introduced parts, yielding in the necessity of simulating the whole ATE test (e.g. normally we would have to simulate these parts in the given order). This is caused by the optimal setup used for ATE testing

Part	Digital Transitions	Analog Transitions	Duration
Part 1	0	84	30ms
Part 2	783	318	90ms
Part 3	9840	89	90ms
Part 4	3540	310	60ms
Part 5	1843	50	50ms

Table 5.1: Overview about the independent parts

- instead of disconnecting and reconnecting the same instrument between two parts, it simply stays connected to be used in the next parts. Furthermore, in the case of DOT, the given extraction of the ATE test must be simulated for every single fault, resulting in insufficient high simulation times (as we see in the next section). To overcome this issue, we decoupled the parts from each other manually, such that we can run them in parallel and also in a different order which has also a high potential to decrease the simulation time for a DOT test. Therefore the given parts are treated separately in the following sections.

5.2 Simulation time

When having a look at DOT testing, simulation time becomes a crucial aspect. In this section, we have a closer look at the single parts of the extraction of our ATE test and how they influence the simulation time. Note that the environment used for modelling the ATE does not influence the simulation time significantly. It is determined by the complexity of the chip in our example. Of course, when the *load board* becomes more complex (e.g. another DUT is used), it will contribute a significant amount of simulation time. Note that we spent no efforts in optimizing the simulation time - this is not part of this thesis. Furthermore no models of the function blocks of the DUT were used.

Figure 5.2 depicts the simulation times (in hours) of every independent part used in our setup. As we see, *part 3* needs a lot more simulation time than the mixed-signal or pure analog parts. We suppose that this is caused by the long run time of this part (90ms) together with a high number of digital transitions which seems to slow down the simulation time. In contrast to *part 3*, *part 1* just contains analog transitions together with a low run time on the ATE, yielding in fast simulation times. We also note that *part 2* contains less digital transitions than *part 5*, but has a higher simulation time. We explain this by the usage of an oscillator during *part 5*, which causes a huge amount of transitions and slows down the simulation time. All in all, we note that the number of digital transitions as well as the run time on the real ATE are the properties that influence the simulation time.

In Figure 5.3 we depict the contribution of every single part to the whole simulation

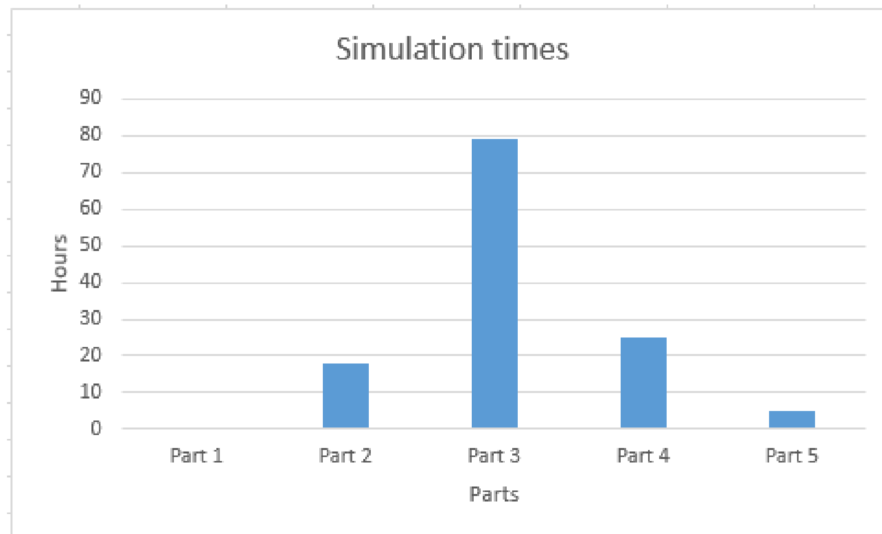


Figure 5.2: Simulation times

time (in percent). The overall simulation time is about 127 hours, where more than 60% of this time are consumed by *part 3*. Also here, we can see that the number of digital transitions provide a significant contribution to the simulation time.

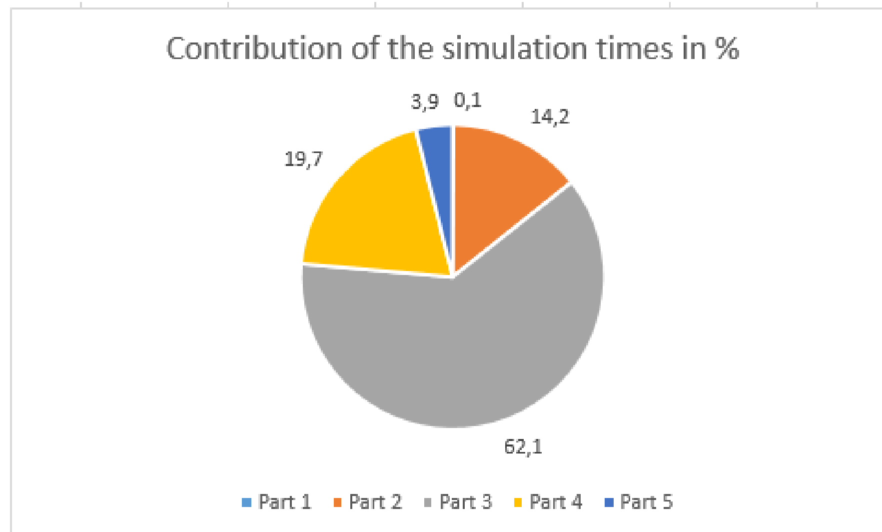


Figure 5.3: Distribution of the test time in percent

All in all we note that a simulation time of 127 hours for one execution of this extraction of the ATE test is too much for being used in a DOT test (a calculation gave an indication that approximately 40 years would be needed to check every single fault). This is one part, which needs to be extended in future work such that it is possible to use DOT.

For example, NXP already introduced an approach for reducing the simulation time in [KTH+11]. They run the whole simulation just one time to get a reference simulation. Afterwards, the ATE test is just simulated in a boundary around the strobes - this approach provides a big improvement concerning simulation time, while still most of the faults are detected. Similar approaches for improving this time are proposed in [HDT+11] or [FGK01].

5.3 Fault detection

Now that we got an idea about the needed simulation time, it remains to show that an injected fault will also be detected. In this section, we inject a fault into the chip which is surely detected by the *continuity test* on the ATE. The fault is an open at the pin IN4.

First of all, the expected behaviour of the DUT without faults is shown in Figure 5.4. We have already seen, that the voltage measured at IN4 has to be in a certain boundary around 700mV during the *continuity test* (while a given current is sourced on that pin). Since this is a correct behaviour, we notice that the limit checker's output stays at 5V for the whole test.

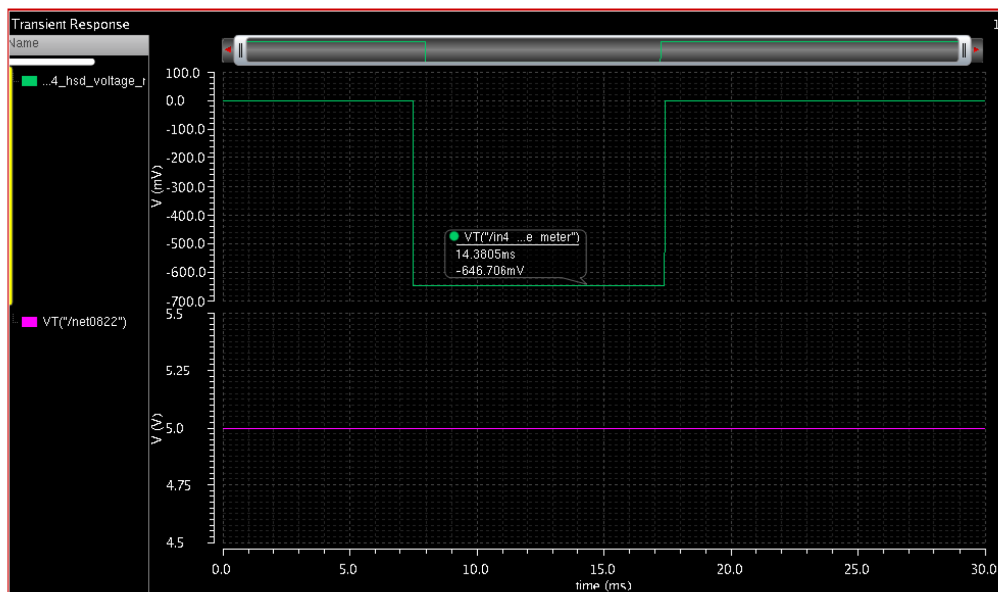


Figure 5.4: Expected behaviour for the IN4 pin

Next, we insert an open in the pinning between the pin IN4 and the *load board* as can be seen in Figure 5.5. Caused by this open, the current is not able to flow into the pin IN4 any more. This is the same behaviour as when there was an open between the pin and the circuit inside the chip it is connected to.

When we simulate the ATE test with this injected fault, we get the results as shown in Figure 5.6. On can see that the voltage now lies in the area of MV. This is caused by

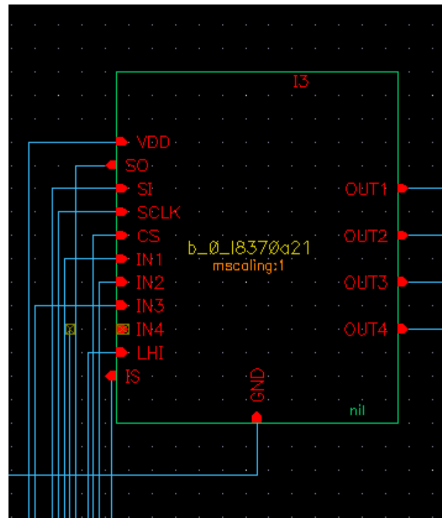


Figure 5.5: Fault in the pinning of the chip

the current which can not flow somewhere. Note that on the real device, the chip or the tester instruments might be destroyed by this behaviour - therefore the test engineer can use a voltage/current clamp as we mentioned in Chapter 3. Since we did not implement this feature, the voltage reaches this high value.

Furthermore, the graphic shows the output of the limit checker. It is 5V at the beginning of the test. After approximately 27ms, the test is finished and the necessary strobes are available - at this point in time the limit checks are performed. The figure shows that the output of the limit checker switches from 5V to 0V indicating a failure. In a DOT test this would stop the test since the fault was successfully detected.

This example acts as our *proof of concept* to show that the ATE test was successfully translated and that the models of the simulation environment as well as the limit checker are working correctly. For the moment, the limit checker was implemented manually in Verilog-A - in future works this needs also to be done automatically, especially for more complex post calculations.

In summary, we saw that the framework introduced in this thesis is working quite well and able to detect injected faults. Compared to previous activities, where the focus lied on the debugging of the test program, this solution can additionally be used for DOT testing because of the exact modelling of the simulation (no models or other simplifications were used). Future works need to focus on the reduction of the simulation time to improve this approach such that it can be used for DOT testing. But with this extension, the proposed framework is a good candidate to be used in industry.

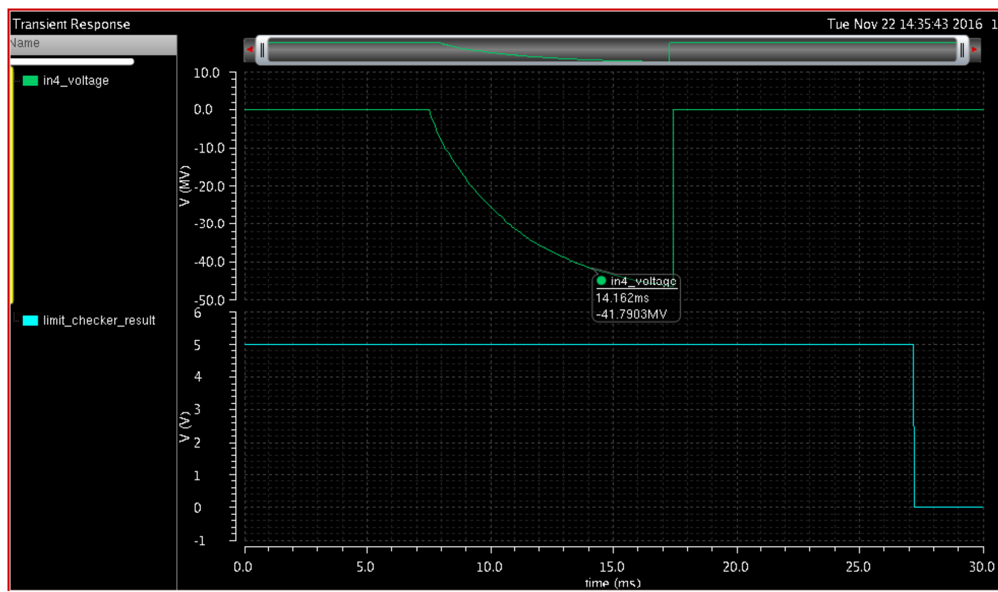


Figure 5.6: Faulty behaviour for the IN4 pin

CHAPTER 6

Conclusion and future work

Caused by *random manufacturing defects* during the production of AMS devices, it is necessary to perform a last verification before the chip is transferred to the customer. To address this problem, the common approach in industry is the usage of an ATE - several instruments provide different stimuli for the DUT and it is verified via strobes whether the device meets all the specification requirements or not.

Although this is a well known methodology for the final verification, drawbacks like uncertain information about the test coverage arise. Therefore it is desired to use the so called DOT. To get better knowledge about the test coverage, it is necessary to simulate a DOT. A DOT simulation can be divided into two parts: translating the ATE test setup for being executed in a simulation environment and the fault injection. In this thesis, we address the first one of these challenges. This challenge (especially the modelling part) was already topic of several publications. What all of them have quite in common is the focus on the debugging of a test program instead of DOT. Therefore they use less accuracy and just small parts of the test program were simulated instead the whole one. Also a working group for defining a portable stimulus has been evolved, but there are no results available yet. To overcome these issues, we provide a generic framework which exactly tackles these challenges and can be used for debugging an ATE test program as well as for verifying a DOT.

Our first step was to figure out the core functionalities and common test cases in industry to get a better knowledge of the required configurations for the model and which properties can be neglected. In general we noticed that the needed behaviours of the ATE are the ability to force and measure voltage/current and digital signals. Furthermore the functionality of measuring time in an event driven way is needed. As an example we used the continuity test throughout this thesis to show the concrete functionality of our models and the translation.

With the knowledge of the needed functionalities of an ATE we modelled the hardware setup of the whole test, such that it can be controlled externally via files. Although in most testers different test instruments with different capabilities (e.g. digital/analog instruments, instruments with event stampers or high voltage/current instruments) are available, we designed one schematic of a general instrument which combines all of these functionalities. Because it is a very specific part of an ATE test, we neglected the *load board* in the context of this work, just note that it must be redesigned when another DUT is used.

Given the simulation model of a general ATE, it is necessary to translate the finished test program such that it can be understood by the environment. In our case translating means to generate waveform files for all the switches and sources used in the model. Therefore we identified the core functionalities which need to be controlled from external in the context of a test program written in an arbitrary programming language. We use code injection to generate a general intermediate file from the specific ATE test program. This approach ensures that just the necessary information needs to be parsed. From this intermediate file, a parser generates the waveform files, the limit checker as well as a script file to configure the file paths used in the simulation environment.

In the evaluation section, we gave an overview about the simulation times needed for verifying the functionality of an Infineon chip with a concrete test setup and provided a proof of concept with a concrete fault injection. Here we saw that it is indeed possible to translate an ATE test program and verify the correct functionality of a DUT with our approach. But there is still room for improvements in future works.

Currently the simulation time is much too high. One simulation of the introduced parts of an ATE test needs about 127 hours. This would have to be executed for every possible fault, which might be quite a lot. NXP described one possible solution for improving the simulation time while keeping the fault detection intend. With this approach it is possible to simulate only the time around a strobe instead of the whole test program. Also the ordering of test cases in dependence of the injected fault can improve the simulation time.

Another problem which occurred throughout the thesis was the modelling of a voltage/current clamp for the sources. In some test programs, the test engineer might run a voltage source into the current limitation such that it acts like a current source. Modelling this special kind of source becomes challenging when the clamping values are dynamic, i.e. they can be arbitrarily changed during the program execution. We tried several solutions we found and also asked the modelling experts for help without any success - there is currently no way to model this special kind of source without getting convergence problems during the simulation. Anyway, this case could simply be avoided by the test engineer by using a current source instead of a voltage source and let it run into the current limitation.

Although the simulation time is quite high for the moment, our approach is working and can be used for the translation of an ATE test program into the given simulation

environment. Since this methodology is very generic, it can be easily adjusted for different ATE testers. Therefore the proposed framework can be used in industry for the translation part of DOT tests and furthermore to debug the ATE test program in an early stage (pre-silicon) of the design flow (also when POT is used). This will give a quite good estimation of the test coverage, which can be used to ensure the quality of the product.

CHAPTER 7

Appendix

Within this chapter, one can find the code we used for the *digital checker* and the *event stamper* written in Verilog-A. Listing 7.1 shows how we implemented the digital checker. The output of the checker is set to 5V at the initial state. Whenever the clock signal crosses 2.5V a new event is generated. In that case, the signal *check* indicates, whether a logic high or low is expected. When a fault (mismatch) occurred, the output of the checker is set to 0V and remains at this value.

Listing 7.1: Code of the digital checker

```
// VerilogA for wk_heindlma, digital_checker, veriloga

`include "constants.vams"
`include "disciplines.vams"

module digital_checker(clk,check,vih,vil,in,out);
input clk,vih,vil,in,check;
output out;
electrical clk,vih,vil,in,out,check;

real output_value;

analog begin

  @(initial_step)
    output_value = 5;

  @(cross(V(clk)-2.5,0)) begin
    if (output_value >= 2.5) begin
      if (V(check) >= 2.5) begin
        if (V(in) < V(vih))
          output_value = 0;
      end else begin
        if (V(in) > V(vil))
          output_value = 0;
      end
    end
  end

  V(out) <+ output_value;

end
```

```
endmodule
```

The used implementation of the event stampers is shown in Listing 7.2. The output is set to 0V in the initial state. When the enable signal crosses 2.5V with a rising transition, the values of VIL and VIH as well as the trigger event are stored in the corresponding variables. Whenever one possible event occurs, it is checked whether this event matches the trigger event - in that case the actual time is measured and written to the output. Note that an enable signal of 6V indicates an immediate time measurement without waiting for an event.

Listing 7.2: Code of the event stamper

```
// VerilogA for wk_heindlma, event_stamper, veriloga

`include "constants.vams"
`include "disciplines.vams"

module event_stamper(enable,vih,vil,trigger_event,in,out);
input enable, vih, vil, trigger_event, in;
output out;
electrical enable, vih, vil, trigger_event, in, out;

real output_value;
real triggering_event;
real vih_tmp, vil_tmp;

analog begin
  @(initial_step) begin
    output_value = 0;
    vih_tmp = 0;
    vil_tmp = 0;
    triggering_event = 0;
    output_value = 0;
  end

  @(cross(V(enable)-5.5,+1)) begin
    output_value = $abstime;
  end

  @(cross(V(enable)-2.5,+1)) begin
    triggering_event = V(trigger_event);
    vih_tmp = V(vih);
    vil_tmp = V(vil);
  end

  @(cross(V(enable)-2.5,-1)) begin
    triggering_event = 0;
  end

  @(cross(V(in)-vih_tmp,+1)) begin
    if(triggering_event == 4) begin
      output_value = $abstime;
    end
  end

  @(cross(V(in)-vih_tmp,-1)) begin
    if(triggering_event == 3) begin
      output_value = $abstime;
    end
  end

  @(cross(V(in)-vil_tmp,+1)) begin
    if(triggering_event == 1) begin
      output_value = $abstime;
    end
  end
```

```
  @(cross(V(in)-vil_tmp,-1)) begin
    if(triggering_event == 2) begin
      output_value = $abstime;
    end
  end

  V(out) <+ output_value;
end
endmodule
```

List of Figures

1.1 Time-to-market improvement by ATE 2
1.2 Analog Mixed-Signal Design Flow [BSED07] 2
1.3 Random Manufacturing Defects 3
1.4 Schematic of the test environment 4
1.5 Flow diagram of DOT 5
1.6 Overview of an abstract language for portable stimuli [Gro16] 8

2.1 Analog Mixed-Signal Design Flow focussed on final verification [BSED07] . . 11
2.2 Hardware setup of an ATE test 14
2.3 Software setup of an ATE test 15
2.4 ATE test flow 15
2.5 Continuity test 17
2.6 Setup of the Teradyne Flex Tester 18

3.1 Modelling part in the design flow [BSED07] 21
3.2 Top-Level setup of the simulation environment 22
3.3 Connection between DUT supply and load board 24
3.4 Connection between DUT and load board 25
3.5 Schematic of a test instrument 26
3.6 Pinning of a test instrument 26
3.7 Meter outputs 28
3.8 ATE test instrument channel 28
3.9 ATE test instrument 30
3.10 Example definition of a waveform 31
3.11 Digital checker block 32
3.12 Example for the functionality of the digital checker 32
3.13 Event stamper block of the instrument 33
3.14 Connections of the four event stampers 34
3.15 Example of an event stamper 35

4.1 Translaton process in the design flow [BSED07] 37
4.2 Abstract schematic of ATE 38
4.3 Relay functionalities 39
4.4 Overview about the basic general functions of an ATE 40

4.5 Overview of the basic properties concerning the levels 41
4.6 Overview about the basic timing functions of an ATE 42
4.7 Overview about the basic analog functions of an ATE 43
4.8 Overview about the basic timing functions of an ATE 44
4.9 Overview about the basic digital functions of an ATE 45
4.10 Translation process for an ATE test program 48
4.11 Flow diagram for the code injections 49
4.12 Flow diagram with limit checker included 53
4.13 Schematic of the functionality of the limit checker 54
4.14 Schematic of the state machine for the limit checker 54
4.15 Connections of VS_DC30 for the continuity test 58
4.16 Forced signals by VS_DC30 for the continuity test 59
4.17 Limit checker of the continuity test 59

5.1 ATE test setup 62
5.2 Simulation times 64
5.3 Distribution of the test time in percent 64
5.4 Expected behaviour for the IN4 pin 65
5.5 Fault in the pinning of the chip 66
5.6 Faulty behaviour for the IN4 pin 67

List of Tables

4.1 Relay functionalities and commands 39
4.2 Commands available on the force line 40
4.3 Commands for configuring the pin levels 41
4.4 Commands available for the timing parameter 42
4.5 Commands available for the source 43
4.6 Commands available on the sense line 44
4.7 Commands for configuring the event stampers 45
4.8 Commands available on the digital sense line 45

5.1 Overview about the independent parts 63

Glossary

load board sets up the connections between ATE instruments and DUT. 3

random manufacturing defects defects generated by particles during chip manufacturing (e.g. opens or shorts). 2, 12

test coverage given a set of possible faults, this number indicates how many of them are detected by the test. 1, 3, 4, 69, 71

Acronyms

AMS analog mixed-signal. 1, 2, 4–7, 12, 14, 16, 31, 38

ATE automatic test equipment. xv, 1–5, 7, 11–18, 20–23, 25–31, 33–35, 37, 38, 40, 42–48, 50–52, 54, 56, 58–60, 77, 78

CCVS current controlled voltage source. 27

DOT defect oriented test. 4–6, 16, 77

DUT device under test. 3, 4, 12, 13, 16, 18, 19, 22–25, 27, 34, 38, 40, 49, 51, 58, 77

IPWL current piecewise linear source. 29

IPWLF current piecewise linear source - file controlled. 29

PSet parameter set. 18, 20

SoC system on chips. 1

VCVS voltage controlled voltage source. 27

VIH voltage in high. 28, 32, 33, 40, 44

VIL voltage in low. 28, 32, 33, 40, 44

VOH voltage out high. 40, 42

VOL voltage out low. 40

VPWL voltage piecewise linear source. 27–29, 52

VPWLF voltage piecewise linear source - file controlled. 29, 31, 33, 56

Bibliography

[BHA95] A. Balivada, Y. Hoskote, and J. A. Abraham. Verification of transient response of linear analog circuits. In *VLSI Test Symposium, 1995. Proceedings., 13th IEEE*, pages 42–47, Apr 1995.

[BK92] S. C. Bateman and W. H. Kao. Simulation of an integrated design and test environment for mixed signal integrated circuits. In *Proceedings International Test Conference 1992*, pages 405–, Sep 1992.

[BSED07] H. G. Bakeer, O. Shaheen, H. M. Eissa, and M. Dessouky. Analog, digital and mixed-signal design flows. In *2007 2nd International Design and Test Workshop*, pages 247–252, Dec 2007.

[BXvK+99] R. H. Beurze, Y. Xing, R. van Kleef, R. J. W. T. Tangelder, and N. Engin. Practical implementation of defect-oriented testing for a mixed-signal class-d amplifier. In *European Test Workshop 1999. Proceedings*, pages 28–33, May 1999.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[DG04] B. Deng and W. Glauert. Formal description of test specification and ate architecture for mixed-signal test. In *2004 International Conferce on Test*, pages 1081–1090, Oct 2004.

[FGK01] Liquan Fang, G. Gronthoud, and H. G. Kerkhoff. Reducing analogue fault-simulation time by using ifigh-level modelling in dotss for an industrial design. In *IEEE European Test Workshop, 2001.*, pages 61–67, May 2001.

[Fil11] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397, 2011.

[Gro16] Portable Stimulus Working Group. Update from dvcon u.s. 2016. Accellera Standards Technical Update, 2016.

[GV99] A. Ghosh and R. Vemuri. Formal verification of synthesized analog designs. In *Computer Design, 1999. (ICCD '99) International Conference on*, pages 40–45, 1999.

[HDT+11] H. Hashempour, J. Dohmen, B. Tasić, B. Kruseman, C. Hora, M. van Beurden, and Y. Xing. Test time reduction in analogue/mixed-signal devices by defect oriented testing: An industrial example. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[JBG+15] S. Jakšić, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničkovié. From signal temporal logic to fpga monitors. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 218–227, Sept 2015.

[KRT02] G. Krampl, M. Rona, and H. Tauber. Test setup simulation - a high-performance vhdl-based virtual test solution meeting industrial requirements. In *Test Conference, 2002. Proceedings. International*, pages 870–878, 2002.

[KTH+11] B. Kruseman, B. Tasić, C. Hora, J. Dohmen, H. Hashempour, M. van Beurden, and Y. Xing. Defect oriented testing for analog/mixed-signal devices. In *2011 IEEE International Test Conference*, pages 1–10, Sept 2011.

[Lan90] R. Lanoo. A mixed digital-analog simulation and test environment. In *Euro ASIC '90*, pages 7–10, May 1990.

[RGA02] Rob A. Rutenbar, Georges G. E. Gielen, and Brian A. Antao. *A Tutorial Introduction to Research on Analog and MixedSignal Circuit Testing*, pages 735–753. Wiley-IEEE Press, 2002.

[RKR03] M. Rona, G. Krampl, and F. Raczkowski. Automating the device interface board modeling for virtual test. In *The Eighth IEEE European Test Workshop, 2003. Proceedings.*, pages 71–76, May 2003.

[SGOT98] M. B. Santos, F. M. Goncalves, M. Ohletz, and J. P. Teixeira. Defect-oriented testing of analogue and mixed signal ics. In *Electronics, Circuits and Systems, 1998 IEEE International Conference on*, volume 2, pages 419–424 vol.2, 1998.

[TSR+09] F. Torres, R. Srivastava, J. Ruiz, H. P. Wen, M. Bose, and J. Bhadra. Portable simulation/emulation stimulus on an industrial-strength soc. In *2009 International Test Conference*, pages 1–1, Nov 2009.

[Wan06] Wen Wang, Wu. *VLSI Test Principles and Architectures.* Morgan Kaufmann, 1 edition, 8 2006.

[Web89] B. A. Webster. An integrated analog test simulation environment. In *Test Conference, 1989. Proceedings. Meeting the Tests of Time., International*, pages 567–571, Aug 1989.

[Xin98] Y. Xing. Defect-oriented testing of mixed-signal ics: some industrial experience. In *Test Conference, 1998. Proceedings., International*, pages 678–687, Oct 1998.

[ZT09] M. Zaki and S. Tahar. Formal verification of analog and mixed signal designs. In *2009 4th International Design and Test Workshop (IDT)*, pages 1–1, Nov 2009.