Lund University
Department of Electrical and Information Technology

Simulation and Verification Methodology of Mixed Signal Automotive ICs

Author: Banafsheh Rezaeian
Examinor professor: Dr.Joachim Rodrigues
Supervisor: Alexander W. Rath

November 22, 2012

This Page is Intentionally Left Blank

Abstract

The Universal Verification Methodology standard provides immense advanced automatic techniques to the digital verification world. In fact without mechanisms like constraint random stimulus generation, functional coverage and self-checking testbenches, functional verification of today's complex integrated circuits is not conceivable. While digital verification benefits from all these modern methods, analog verification is still a manual process.

This report presents a new verification approach which simulates mixed-signal designs through utilizing sophisticated methods used in digital verification. The developed verification technique in this work makes it possible to simulate a DUT with both digital and analog interfaces using event driven simulators. This method is compatible with existing digital verification techniques. The aimed device under test (DUT) contains register transfer level (RTL) and real number model (RNM) blocks.

As transaction level modeling affects digital verification to a great extent, analog transaction level modeling is exploited in this work to achieve more facile strategies in challenging mixed-signal verification.

Acknowledgements

Alexander Wolfgang Rath deserves the greatest thanks for his supportive supervision of my thesis. It was working with him that made writing this thesis a fascinating experience. I appreciate the time he spend mentoring this thesis during every steps of this work and correcting its final report.

I owe special thanks to my professor Dr.Joachim Rodrigues, who encouraged me during this research. He generously shared his knowledge of chip design and verification with me during my studies at Lund University.

I would like to express my deep gratitude to Dr.Ulrich Fiedler and Dr.Paul Wallner for the unique opportunity that they gave to me while doing this thesis and providing me with all requisite at Infineon Technologies AG.

My special thank to my family and all my friends in Lund and Munich who helped me get through the graduate school.

Banafsheh Rezaeian

Contents

List of Figures

Chapter 1

Introduction

The ultimate goal of ASIC designs is to construct a chip based on highly precise specifications. These predefined specifications have to be inspected during and after design flow. In order to acquire the highest possible level of certainty in the functionality of the design it is of vital importance to use accurate, liable verification techniques.
In design of today's SoCs functional verification complexity rises exponentially with hardware compelxity doubling exponentially with time [1]. Many novel, advanced and liable methods have been introduced for digital verification (see chapter 2.4.1). However, up to 70 % of design developement time and resources are still dedicated to functional verification [1].
As SoCs became multifunctional more analog blocks are integretaed on a single chip. Therefore, mixed signal verification difficulties became more visible to verification teams. Moreover, at chip level verification, when block level verification is accomplished successfully, it is required to develop a technique through which analog and digital sub blocks of a design are considered holistically.
Universal verification methodology (UVM) by offering base class libraries, brings much automation to the digital verification world. In addition, this methodology is simulator vendor independent. It is possible to create UVM components and reuse them in different projects. Taking advantage of constraint random stimuli generation offered by UVM, the engineering effort has been turn into building automatic checkers instead of writing directed test. The typical approach in UVM based verification flow is to define a test case and simulate the design under test (DUT) with constraint random stimuli to put the design into corner cases. Automatic checkers ensure the correctness of the design functionality according to the generated stimuli. Furthermore, coverage mechanisms are used in order to measure the inspected specification of the DUT and to point out to the verification closure.
The main goal of this work is to utilize the provided facilities by UVM in mixed-signal verification environments. The proposed structure by this work highly improves the mixed-signal verification quality in terms of verification performance with minimum human effort devoted. It is possible to generate real value input stream and drive analog

interfaces of the design under test using automated digital verification techniques like constraint random stimuli generation. Moreover, the utilized analog transaction level modeling approach (first introduced in [2]) enhances full-chip verification procedure significantly.
In this work, a UVM based mixed-signal verification environment is proposed. Taking advantage of developed technique by this work and under-developed methods for monitoring, checking and coverage collection a DUT with both digital and real valued interfaces can be verified at nearly digital simulation speed.

This report is structured as follows. In the second chapter an introduction to verification is given along with a review of different verification approaches. Furthermore, functional verification and its utilized methods in digital world are studied. Eventually a brief study of analog circuit simulation is given. In the third chapter, after a comparison between analog behavioral modeling approaches, analog transaction level modeling is discussed. It is in chapter four where the main advantage of this work over previous work is presented. Moreover, the implementation detail is given in this chapter. Finally, in chapter five the application of developed technique in a real world project is described.

Chapter 2

SoC Verification Methodologies

During last decades semiconductor industry has experienced immense advancements in terms of integrated circuit's capacity, complexity and fabrication technology. New technological capabilities besides increasing demand for high quality, multi functional ICs bring about integrating more and more diverse functions on a single chip. Moore's prediction has been realized and as complexity and capacity of electronic designs grow, production costs rise concurrently.
Costly semiconductor realization draws attention to obtain assurance of debugging all the bugs prior to the design fabrication. Expensive design development signifies the necessity of verification in all design steps. In the absence of verification specifically from the early stages of the design, presence of functional defects is not a far fetched.

2.1 Introduction to Verification

Design flow of a system-on-chip involves diverse steps. Data "Transformation" when crossing the steps is an inevitable outcome.
The design flow tasks are performable taking advantage of numerous complicated electronic design automation (EDA) tools. Large amount of data is required to be able to apply these advanced tools properly and this is not an unlikely occurrence to face several errors during the flow.
In spite of the fact that design *automation* has driven the IC implementation in a way where human impact on it has diminished significantly, human impact is still an important source of error.
Moreover, today's productive SoC design flow is based on intellectual property (IP) reuse. In order to integrate IPs more efficiently into the target system, the proposed verification IP (VIP) by IP vendor plays an important role.

A general verification definition can be:

"Act of checking, testing, inspecting whether a product, service or system (or set thereof) meets its initial requirements and specifications." [3]

The seriousness of verification has been frequently emphasized in many papers. A financial disaster might occur for a company if a bug is discovered in an already sold IC. Intel's Pentium floating point division bug in 1994, imposed 480 million U.S dollar detriment.
Although verification has been a main investigation subject between designers in recent years, still there is wide gap between design capabilities and verification capabilities.
Today, verification is the major bottleneck of designs and between 50-70% of project effort is dedicated to design verification.

2.2 Verification in design flow

From the early cycles of project development, it is essential to concentrate on verification. To prevent time-to-market delay it is of vital importance to verify the design in every steps of design flow. The sooner a bug or defect is discovered, the more unnecessary time to devote is redeemed.
Verification indispensability has been demonstrated through different methods and with different purposes.

2.2.1 Formal Equivalence Checking

This technique is a prevalently used technique in integrated circuit development. This technique is considered as a subset of *Formal verification* which is discussed briefly in section 2.3.1.
In general, this method ensures that two different types of design representation show identical behavior. Equivalence checking can be employed in different levels of design abstraction. For instance, in order to compare the RTL code with generated Netlist after synthesis step and before using the netlist in layout design, this technique is commonly used.
Fig. 2.1 depicts the output comparison between design representation A and B for the similar input.

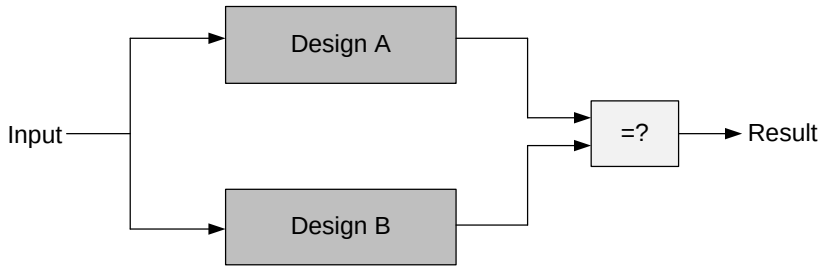

Figure 2.1: Formal Equivalence Checking

2.2.2 Physical Verification

This technique is also known as Layout Verification and is carried out after physical layout design. By means of layout verification, a set of recommended design rules are inspected carefully. In addition, circuit geometry and connectivity has to be verified regardless of behavioral expectations. These are known as *Design rule checking (DRC)* and *Electrical rule checking (ERC)*, respectively. Furthermore, checking *Layout versus Schematic (LVS)* is a process whereby the layout of specific design is checked against the original schematic.

2.2.3 Functional Verification

Increased complexity of designs in addition to growth of design size in recent years, are making their functional verification vastly complicated. Presence of numerous test cases to verify the functionality of a design brings ambiguity about verification closure.
A definition for functional verification can be:
"Functional verification is demonstrating the intent of a design is preserved in its implementation." [4]
The primary goal when performing functional verification is to ensure that design intent was captured correctly and completely by the implementation. Several challenges might occur when verifying the functionality of a design including using random stimulus, defining a set of desired functionalities, writing related tests that are aimed for defined functionalities, indicating verification closure through specifying coverage metrics, writing a reference model and checking design against it and finally using simulators with acceptable performance.

Testing versus Verification Many different views were presented to indicate differences between validation and verification. In general we test a device to ensure that it works but we verify a design to check if it meets its predefined specification.
While testing a series of tests are applied to a DUT and for each test the output is inspected. It is obvious that testing is not sufficient since not all design aspects are

examined.
To be able to indicate the verification closure coverage must be collected. The process begins with defining a verification plan, generating stimulus according to the plan and measuring the verification progress by collecting coverage.
However testing is one way to verify a design it is not a trustworthy approach for todays system-on-chips.

2.3 Functional Verification Techniques

The attempt, when verifying the functionality of a design, is always to answer this question : Does this proposed design do what is intended?
Response to this question, as it has been discussed in section 2.2.3, is a complicated task. In order to overcome the existing challenges various methods are used to verify the functionality of a design.

2.3.1 Simulation based Verification

Simulation is still the most widely used technique between verification teams. The verification environment is constructed by composing a testbench in one of HDL languages. To examine as much distinct functional behaviour of the design as possible, extensive testbench simulation has to be carried out.
Without physically implementing a system, simulation can be defined as "using a mathematical model to recreate a situation in order to estimate the likelihood of various outcomes, often repeatedly." [5]
Simulation, as a traditional verification method, promotes the verification flexibility and is applicable in different design levels. However, building an efficient simulation environment often brings about some difficulties. Moreover, according to some inherent restrictions in HDL's nature, demand for *Hardware Verification Languages (HVLs)* has been arisen.

Formal Verification Due to in-comprehensiveness nature of simulation based verification, a complementary technique is always deployed to detect unrevealed, corner case bugs in the design. *Formal Verification* complements simulation by discovering bugs that are failed to be hit even in efficient simulations.
Furthermore, it is possible to examine the design for some illegal stimuli (input vectors to the design under test which are deemed improbable) by formal verification. Consequently, a solution for future possible changes in interface behaviour can be predicted.
Formal Verification can be described as "mathematical methods used to prove or disprove properties of a design" [6]. Since system specification is mathematically described in early stages of formal verification, a *System model* which illustrates what actually system does is required.

As mentioned, formal Verification is used as a complementary technique to the simulation. Existing limitations in formal verification tools bring about the necessity of also deploying other verification tools. These limitations can be referred concisely in [6].

2.3.2 Coverage Driven Verification

Verification of integrated circuits with significant growth in terms of complexity and design size and multi functionality, brings this question up that how to achieve and ensure the verification closure?
Reliable Verification can be accomplished by defining the verification goals meticulously. *Coverage Driven Verification (CDV)* integrates the automatic test generation procedure, self-checking capacity of the testbench and coverage metrics to highly decrease the time-to-market of a chip. In Fig. 2.2 a testbench structure with the selfchecking ability is depicted. This structure by automatically validating the DUT's output, is a significant facilitation in terms of verification task.
Taking advantage of CDV the tiresome task to generate hundreds of tests and often repetition in running some of them can be avoided.

In CDV focus of attention is on features of the design rather than implementation details and these features are defined precisely in a comprehensible, reusable *verification plan*. By quantitatively measuring the verification progress with the predefined coverage metrics, achieving verification closure is not impossible to reach.

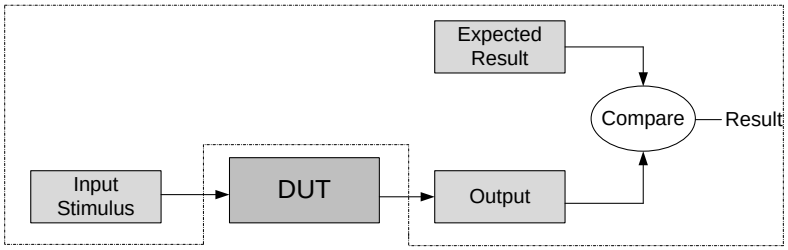

Figure 2.2: Self Checking Testbench Structure

Coverage is obtained through different approaches. *Code coverage* and *Functional coverage* are used as complementary methods together. Moreover, *assertions based coverage* is a practical method to measure verified portions of design behaviour.

Code Coverage Code Coverage is a strong technique between verification teams. By this technique simply, "measuring the executed parts of code" is obtainable. In the absence of code coverage information, portions of design might not be exercised at all. The point to consider is inspected portions of code does not lead to this assumption that

the code functions as intended [7]. Various types of code coverage is used in verification procedure, bringing them up is out of the scope of this Thesis.

Functional Coverage Functional coverage is another coverage oriented approach to collect information on the DUT's behaviour. The primary goal of functional coverage is to inspect if the design meets all it's predefined specifications. Consequently, it is of vital importance to define all the design specifications accurately.
Functional coverage collects data from specification and verification plan to indicate whether and to what extent design functions have been exercised according to the verification plan requirements. [7]
Functional coverage data availability makes it possible to find the holes and reactively adjust the generated stimulus to the DUT during run time of the simulation to cover those holes. This is a significant advantage since more test scenarios can be created and efficiency of generated stimulus can also be increased.
There are always some corner cases which are difficult to be reached by randomly generated tests. In this case, direct testing could help undoubtedly to achieve more coverage. Fig. 2.3 shows the paths to achieve complete coverage.

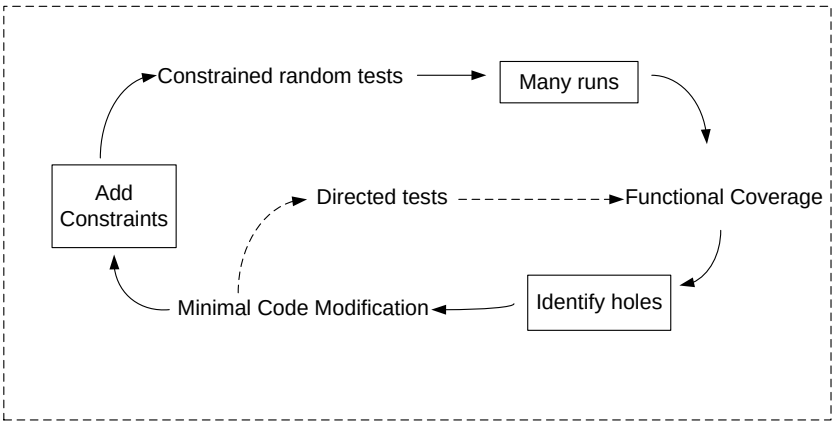

Figure 2.3: Coverage Convergence [8]

Functional coverage and Code coverage each aim for particular goals in the process of verification. Both methods are adequately considered by verification engineers since full coverage in one, does not mean that the DUT has been verified properly.

Assertion Coverage Similar to coverage driven verification flow where code coverage is a powerful technology to estimate the verification closure, in *Assertion based verification* flow, assertion coverage performance indicates the completeness of assertions. [9]
Assertion based verification brings about the possibility to detect and debug design bugs more effectively. Taking advantage of assertion based methodology a design intent is captured and the correctness of it is verified through various verification technologies. Both

dynamic methods like simulation or static methods thorough formal verification are applicable to ensure the correctness of defined condition in an assertion. A user-specific action can be defined to occur in case of either correctness or falseness of assertion's condition.
Some of hardware verification languages, like SystemVerilog, support assertions. Validation of specific design behaviour can be proved by writing an assertion.

2.4 Digital Verification Methodologies

To have a robust, dependable and comprehensive verification environment kernel, existence of a methodology is of fundamental importance. Methodology helps us to perform tasks in a reliable system, taking advantage of rich, predefined and detailed set of rules. Standard libraries offered by a methodology are highly valuable in testbench construction.
Towards successful market competition, adaptation to standard methodologies is an essential requirement for companies. In this manner they can utilize third parties facilities in order to improve their products.

Various Methodologies have been introduced to the semiconductor industry to verify digital integrated circuits. Verification engineers have been encountered numerous challenges towards verification of vastly complicated digital circuits. As a consequence digital verification has been developed also along by the digital design.

2.4.1 Digital Verification Maturity

Digital verification has experienced vast progression in terms of reliability, reusability and automation. Highly developed design technologies require mature verification techniques. Otherwise, with traditional verification technologies it is not possible any longer to inspect all different activities of designs.
Before the occurrence of exponential growth in terms of design size and complexity, it was designers responsibility to verify the *Register Transfer Level (RTL)* code by writing tests and testbenches in one of hardware description languages (HDLs). The natural limitations of HDLs, had raised the demand for hardware verification languages (HVLs). Various HVLs have been introduced to the industry in recent years including *OpenVera*, *e*, *SystemC* and *SystemVerilog*. These languages are comparable to high level programming languages like C++ and contain required features to fill the existing verification holes. Taking advantage of HVLs capabilities, the demand for high quality, bug free semiconductors in shorter project schedules has been satisfied. [10]
The necessity of using HVLs and the required proficiency in building a test environment brought about the demand of a verification team on each project.
Using standard, industry-wide verification methodologies between verification teams also

became a vital necessity to accelerate testbench construction. Distinct methodologies from different EDA companies have been released to enhance verification process. Table 2.1 depicts theses methodologies.

Table 2.1: Verification Methodologies

	Vendor	**Year**	**HVL**
e Reuse methodology (eRM)	Cadence	2002	e
Reference Verification Methodology (RVM)	Synopsys	2003	Vera
Advanced Verification Methodology (AVM)	Mentor	2004	SV/SC
Universal Reuse Methodology (URM)	Cadence	2006	SV/e
Verification Methodology Manual (VMM)	Synopsys	2004	SV
Open Verification Methodology (OVM)	Cadence/Mentor	2008	SV
Universal Verification Methodology (UVM)	Accellera	2011	SV/e

Due to the high demand for interoperability and reusability of verification IPs between projects and companies, a unified verification methodology, UVM, was introduced by Accellera. One of the key features of UVM is that it is simulator vendor independent. It was tested on multiple simulators to ensure its interoperability.
Built-in-automation, coverage driven verification, constrained random stimulus generation and transaction level modelling are some of the added accelerations to digital verification today.

2.5 Metric Driven Verification (MDV)

A great level of automation including automated stimulus generation, independent checking and coverage collection obtained through utilizing UVM. One of the important limitations of coverage driven verification (CDV) was covered by introducing the concept of metric driven verification (MDV). Although the collected coverage via CDV is of significant importance for the verification engineer to find the uncovered holes in the design, still without checking mechanism it is not easy to predict the verification closure. The term "metric" in MDV is used to determine the end of verification. Verification metrics are defined through coverage, checks and assertions.

Furthermore, another MDV preference over CDV is the ability to build an executable verification plan(vPlan). In vPlan a list of prioritized design specification helps the engineer in efficiently utilizing the captured data. A successful verification project is highly dependent on building a comprehensive and accurate vPlan. It is possible to precisely measure the verification progress based on design specifications which are defined in vPlan.
To be able to cover all design specifications it is necessary to run several simulations in

parallel. It is possible to augment verification throughput if automatic stimulus generation and self-checking mechanisms are utilized practically. Intelligent testbenches with these highly automatic mechanisms are able to generate meaningful stimulus to the DUT based on predefined specifications in vPlan and also perform DUT output sampling in order to prove or disprove the correctness of the DUT behaviour. This can be accomplished by comparing the DUT's output samples with expected output. [11]
Moreover, by utilizing the MDV approach in the universal verification methodology (UVM), an open loop verification process converts to a closed loop verification cycle including plan, construct, execute, measure. This closed loop cycle is depicted in Fig. 2.4. As it is shown in Fig. 2.4, MDV is not specific to particular verification engine and different tools support this approach.

2.6 UVM Verification Environment

A typical UVM based verification environment contains three main building blocks : UVM component (UVC), UVM env (environment) and UVM test. Fig. 2.5 shows this UVM verification architecture. Universal verification methodology was released based on OVM and VMM methodologies by Verification Intellectual Property Technical Sub-Committe (VIP TSC).

UVM_test UVM test block is derived from *UVM_component* base class and it is used as the base class for user-defined test classes. User can define multiple test classes and

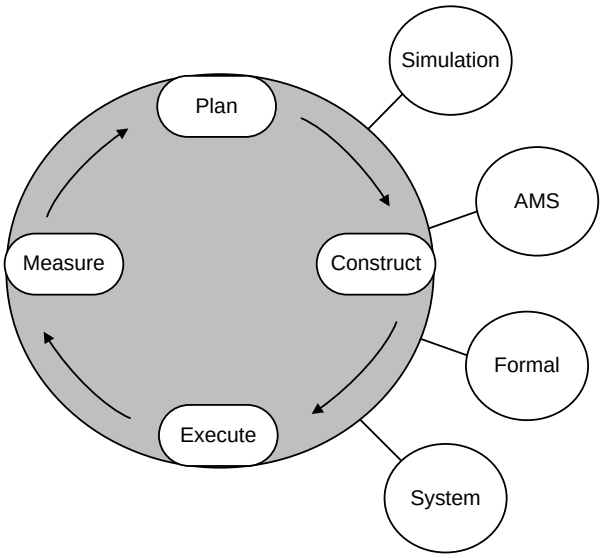

Figure 2.4: MDV Closed-Loop Verification Cycle [11]

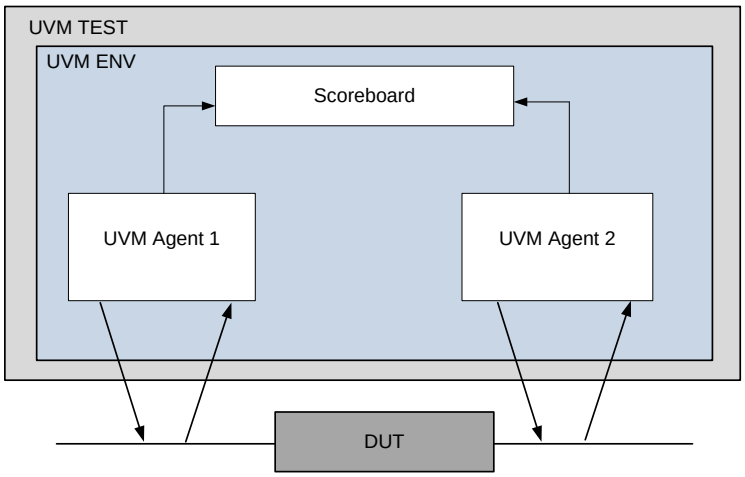

Figure 2.5: Typical UVM environment

randomly select them via command line. In top-level module, the global *run_test()* task is invoked together with DUT instantiation and interface definition (DUT and testbench interface) and the test which is defined in *UVM_TESTNAME* via simulator command line starts the simulation.

UVM_env The *UVM_env* class is instantiated in *UVM_test* and is derived from *UVM _component*. As it is shown in Fig. 2.5 this block is used in order to encapsulate and configure agents (also known as UVM verification components (UVCs)). It includes one or more *agents* and it has capability to define them as an active agent or passive agent depending on configuration. It is environment class main task to generate meaningful random stimulus, sampling and monitoring DUT's result, validating the result and collecting coverage [12].

UVM_agent A UVM agent typically includes three main blocks: Driver, Sequencer and Monitor. All these sub blocks communicate using *transaction level modelling (TLM)* connections. It is through SystemVerilog virtual interface that agent communicates with DUT. Another agent's property is configuration which indicates either it is an active or passive agent. User can define any other control parameters using agent configuration. Active and passive mode architectures of UVM agent are shown in Fig. 2.6.

Scoreboard Scoreboard is build on top of the monitor and performs analysis on the data stream received from monitor. Scoreboard has a significant role in self-checking verification environments. Functional behaviour of the design under test is analysed by this component. Scoreboard at least contains two analysis import including an import to receive input data stream to the DUT, and another import to convey monitored

activities on the DUT's output pins (see Fig. 2.6).
Scoreboard component contains a golden model through which performs a comparison between actual results of DUT (sampled by monitor) and expected results of DUT (Golden model's output in response to the same input vector to the DUT).

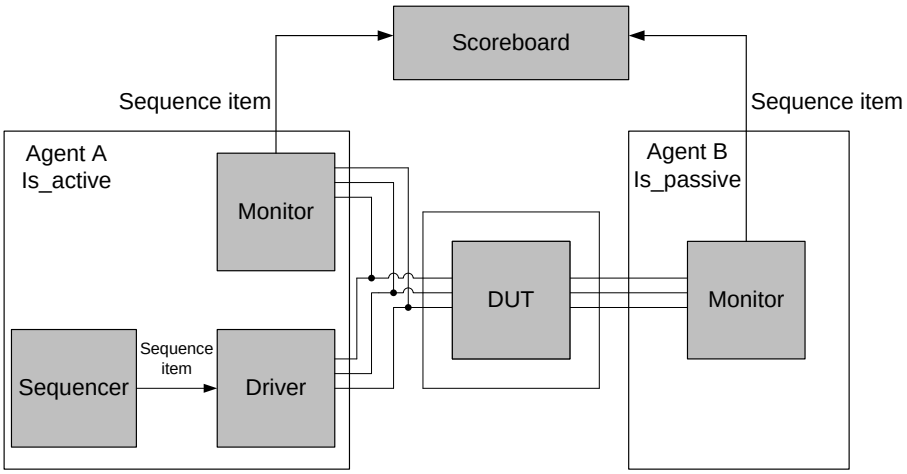

Figure 2.6: UVM agent: in active and passive mode

A typical active agent contains all three subcomponents and in one hand generates random stimulus and drives the DUT pins through driver, and on the other hand samples DUT's input pins (drived by the agent's driver) via monitor. This monitor converts data stream back to transaction level and send it out to other analysis components.

A typical passive agent, without driving the pins of a DUT only performs monitoring of the generated result of a sample DUT. This means that only the monitor component is instantiated in a passive mode agent. Agent in this mode is used for coverage collection and checking operations.
In Fig. 2.6, agent A is depicted with the *is_active* as a configuration property, while the same flag in agent B is set to *is_passive*. As it is depicted the same passed sequence item by the sequencer to the driver, is generated again by the monitor later. In this way, the DUT input pins are driven by the driver component and are sampled by the monitor component of an active agent. This is a reliable practice to ensure that the incoming sequence item to the driver is properly converted to signal level activities and DUT is driven with desired stimuli.

2.6.1 Transaction Level Modelling

In order to increase verification productivity and manage complex system-on-chips verification, higher abstraction level of the design is strongly requested. In an abstraction level higher than RTL, interfaces are defined in terms of transactions.

"In transaction-level-model (TLM), the details of communication among computation components are separated from the details of computation components." [13]

While the DUT communicates with verification environment at signal-level, it has been proved that it is necessary to handle most of verification tasks such as stimulus generation or coverage collection at transaction level [12]. In this abstraction level, complicated data transferring between units is managed at a higher level.
Many UVM components communicate to each other using TLM. In this way, components are isolated from other component's modifications. In order to handle the communication between DUT and verification environment which is at signal-level, it is required to provide a layer in the testbench in which UVM components are able to convert transaction-level activity to signal-level activity and vice versa.
Moreover, it is through TLM that encapsulated, reusable *verification components* can be obtained and whereby verification environment construction is facilitated.
In UVM base class libraries, a base class called *UVM_sequence_item* for all user-defined transactions is declared. This class is extended from *UVM_transaction* and ultimately is inherited from *UVM_object*.

TLM communication in basic level is depicted in Fig. 2.7. Transaction is generated by the producer which holds a *port* (square). Transaction is consumed by the consumer which holds an *export* (circle). A basic definition of a transaction states that the producer puts a transaction and consumer gets it.

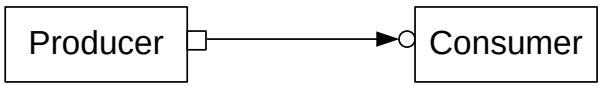

Figure 2.7: Simple producer consumer [12]

Transaction level communication between two components might happen in two different situations as it is shown in Fig. 2.8. If data and control flow are in the same direction, it is producer which *puts* transaction into consumer (a). While in other situation, when data and control flow directions disagree, it is the consumer which requests transaction from the producer via *get* port (b). However, it is *export* (circle), which implements the transaction in both situations. [14]

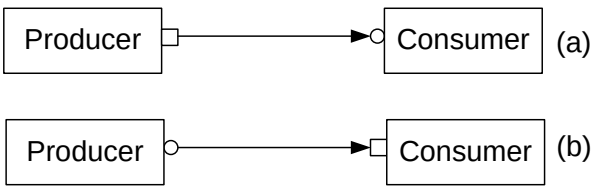

Figure 2.8: Put versus Get

Transaction-level models are utilized in UVM environment efficiently. A UVM based verification environment holds four basic transaction-level components.

Sequencer Sequencer creates transaction-level traffic and pass it to the driver. Extending from *UVM_sequencer* base class, the sequencer is able to communicate with driver in a parametrized way. These parameters are of same type and are called *request* and *response*. User can specify a user-defined transaction type to these parameters. Otherwise, they have the default *UVM_sequence_type* type.
Upon a request from driver, sequencer selects a sequence from listed sequences and passes it to the driver. A sequence is a group of transactions. Multiple sequences together perform a planned scenario. Basically, sequencer's arbiter role controls the flow of these sequences from multiple generators.
Through sequencer's *seq_item_export* data items (transactions) are sent to driver's *seq_item_port*. In Fig. 2.9 basic interactions between a driver and a sequencer to shape a transaction model is shown. In this model the following tasks are executed in depicted order.

First, a transaction is created in sequence using *UVM_create (req)* method. By calling *wait_for_grant* method, sequencer blocks any activity until the driver asks for the next item through calling *get_next_item* method. Only after driver's request, the transaction can be randomized optionally. Next step is to send the transaction via *UVM_send (req)*. The retrieved transaction by driver is consumed and converted to signal level according to the application specification. Driver raises the *item_done (rsp)* flag to send a response back to the sequencer (this is an optional step). Finally, *wait_for_item_done ()* method in sequence, which is a blocking method, gets unblocked.

Driver Driver component is responsible for driving DUT's pins. As it was mentioned in this section, driver is a component which is located in the layer in which transaction-level activities are converted to signal-level activities. In other words, driver retrieves transactions from sequencer and translate them to a signal value. This signal drives DUT's pins via SystemVerilog virtual interface.
In UVM base class libraries, the *UVM_driver* is defined as a base class for user defined drivers. When driver's connectivity to sequencer via *seq_item_port* and to the DUT via virtual interface are implemented, it can obtain the next available data item from sequencer. Driver can also use another port called *rsp_port* to send a response back to the sequencer.

Monitor DUT's output pins activity is monitored and sampled by monitor component in order to determine if it behaves correctly. Monitor's performance is in opposition to the driver. In Fig. 2.10 the location of monitor and driver in a mediator layer between DUT and verification environment is depicted. In order to verify DUT's behaviour, it is monitor's duty to extract pin-level activities of the DUT and translate them into

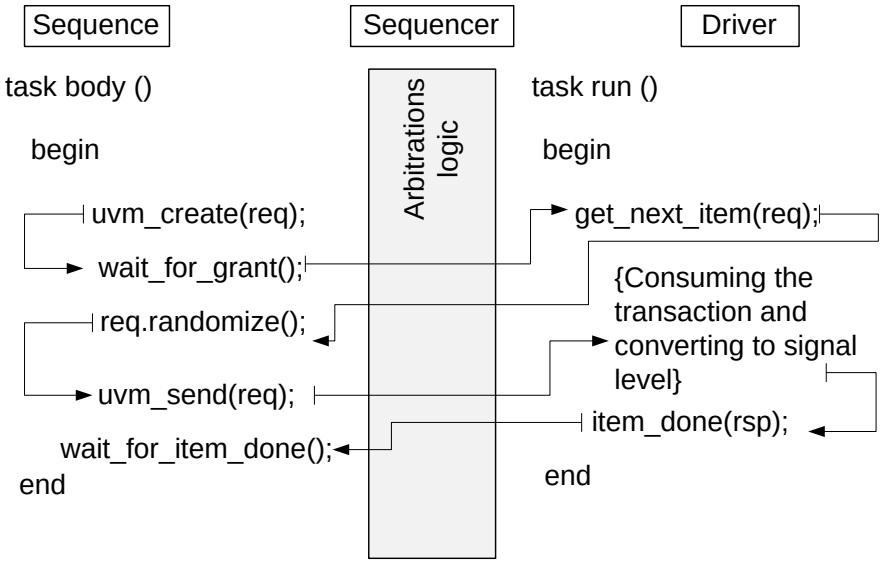

Figure 2.9: Sequence and Driver communication

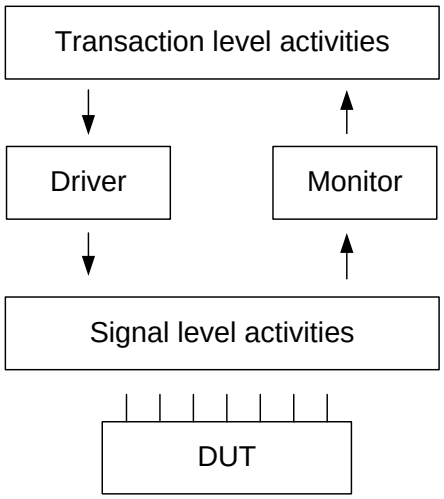

Figure 2.10: Driver and Monitor in mediate layer

transaction level and send it to other components for analysis. Monitor's main tasks can be count as basic monitoring and coverage collection. DUT's functional behaviour is verified by other high level components like Scoreboard.
Virtual interface, is the way through which monitor collects data from information bus. Collected data might be used in coverage collection or exported to analyser components via *UVM_analysis_port*.

TLM analysis communication is a different approach from the mentioned put/get TLM communication approach in section 2.6.1. The major difference is that in put/get approach, no matter if it is put or get, presence of a component with export connection is necessary to implement the transaction. While, in TLM analysis communication regardless of target's presence it is important to generate transactions and send it through analysis port. In other words, this port is not dependent on export connection. The transaction simply returns in the absence of any export connection.

A base class named *UVM_monitor*, preserves all basic features of user-defined monitors. Monitor checks the observed data format and generates UVM reports, accordingly. A print function is defined in UVM transactions which can be called by monitors in order to print out the transaction's content.

2.6.2 UVM phasing

To have an ordered execution flow in an UVM based verification environment, eight standard UVM phases are introduced in order to arrange the major steps of the simulation process. It is from the top level module where the run_test() task is called and pre_run phases build(), connect(), end_of_elaboration(), start_of_simulation() all execute at time 0. After completing the pre_run phases its time to start the run() phase. All simulation related tasks are fulfilled at run() phase. Post_run() phases extract(), check(), report() are execute when run () phases stops.
In this section a brief description of each UVM different phases is declared. It is important to notice that all UVM standard phases except run() phase are execute in zero time.

Build phase This phase is executed at the start of the simulation and builds the testbench components hierarchy from the top to the down. It other words, it is only after the execution of parent component build() method that the child component build method is executed. It is also possible to configure the low level component construction from the higher level components.

Connect phase This phase is executed after the build phase and is in reverse direction i.e from bottom upwards. All UVM component connections are established in this phase importantly TLM ports and exports of the TLM components.

End of elaboration Before start of the simulation the final configuration and connectivity tuning of the existing components is done bottom up during this UVM phase. Besides, connections are validated during the end_of_elaboration phase.

Start of simulation phase This phase is the last phase before the beginning of the only time consuming UVM phase(run_phase). It is possible to print the entire testbench topology at the end of this phase.

Run phase This phase is implemented as a task and therefore consumes time. During this phase all run_tasks of UVM components are executed in parallel. This phase is used by transactors. User_defined stimulus is generated during this phase and DUT is simulated. Reset, configure, main and shutdown phases are all executed during the run_phase.

Extract phase Clean up phase starts with extract phase and gathers data on the final DUT status. The data is gathered from scoreboard and monitors.

Check phase During this phase the correctness of the extracted data from DUT is examined against the expected result. This phase is used by analysis components.

Report phase The simulation results from analysis components are reported and printed out.

2.6.3 Base class libraries

UVM common base class libraries (CBCL) enable users to build a modular, scalable, reusable and interoperable test environment. Fig. 2.11 shows the UVM base classes hierarchy.
All UVM data classes derive from *UVM_object* base class which contains common data operations methods like *create*, *compare*, *copy*, *print* and *record*.

UVM_transaction base class is the root class for transaction classes like *UVM_sequence _item* class. User defined transaction classes drive from *UVM_sequence_item* and inherit *UVM _transaction* methods, indirectly.
UVM_component base class is the root class for UVM components. UVM components are defined as "quasi-static objects". It is possible to obtain a structural hierarchy through UVM components like modules hierarchy. [16]

2.7 Analog Circuit Simulation

Analog integrated circuits are simulated using analog simulators, like SPICE, over the last decades. SPICE (simulation program with integrated circuit emphasis) is the most

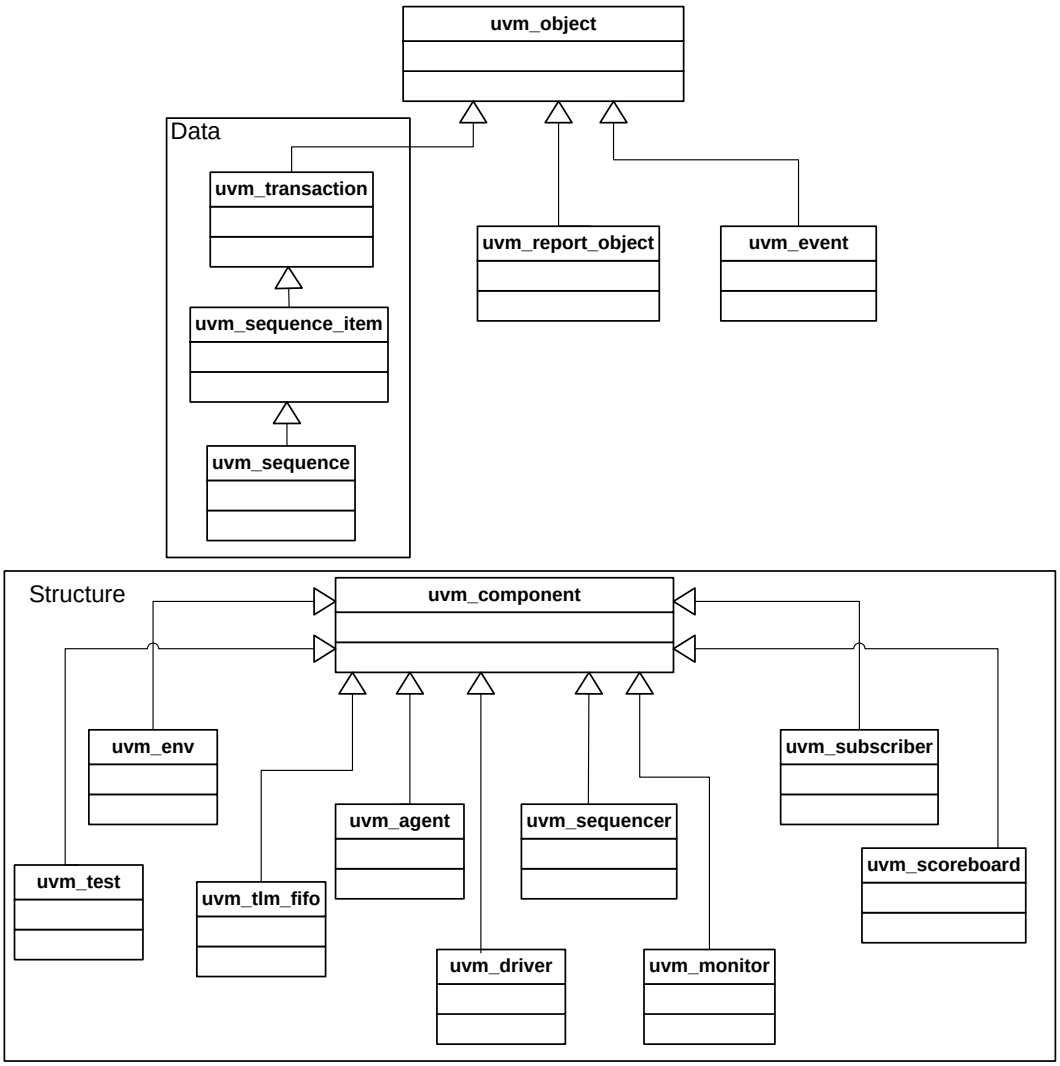

Figure 2.11: UVM base classes hierarchy [15]

well known tool for analog simulation. SPICE simulation before manufacturing the integrated circuits is a common task for designers to ensure the correctness of the design. The tool was first developed at the University of California at Berkeley as a class project about 40 years ago and since then it became a worldwide standard circuit simulator.
SPICE simulation covers wide range of components from simple passive to complicated active devices like MOSFETs. By writing a text-description of a component, the SPICE simulation model of it is achievable and it's behaviour is predictable. SPICE like simulators are suitable for detailed design analysis, when the design is constructed in transistor-level [17]. Even for a small portion of a design SPICE simulation might take a week or more to be accomplished. SPICE simulation performance was improved to speed up the simulation at about 2-4 orders of magnitude over the last years. Although this improved simulation speed was achieved at the expense of loosing simulation accuracy [18].

In order to provide higher levels of productivity, it is of significant importance to utilize various simulation tools during all steps of design development. For an analog circuit designer to gain better productivity it is crucial to model the circuit in a higher level of abstraction and verify its functionality. This is how designers can simply detect some functional defects prior to devoting time to design to the transistor level.
Moreover in chip level verification when the design is going to be checked in context of the whole system and verification aim is more to check system level behaviour or interconnectivity between blocks, detailed results of SPICE models are not requested.
As it is discussed in details in [18], a top-down approach in analog and mixed-signal

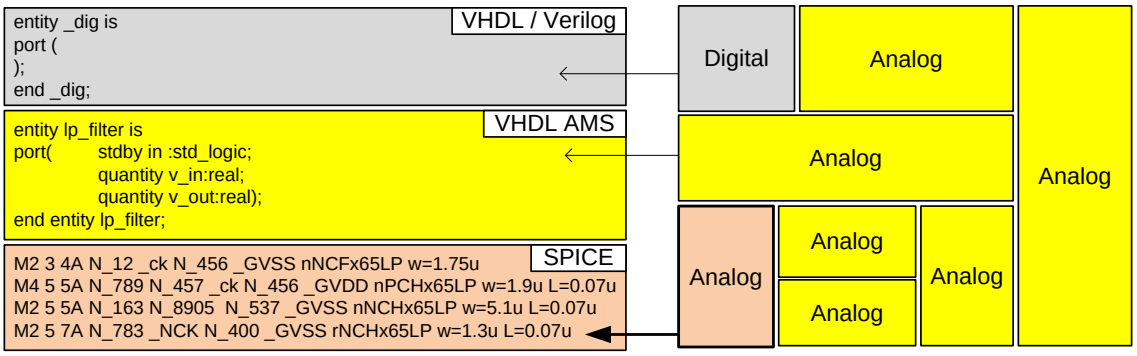

Figure 2.12: Top down approach in analog centric designs [19]

designs inhibits from facing wrong assumptions after dedicating considerable amount of time and effort to design to transistor level. To overcome the increasing complexity of analog centric integrated circuits, an abstract model in the beginning of the design process is very much helpful. This abstract model as it is shown in Fig. 2.12 might contain digital blocks (modeled using HDLs), more repetitive analog blocks (modeled using analog mixed signal (AMS) languages) and other analog blocks(modeled with SPICE netlists).

While the mentioned approach suits well for analog centric designs with vastly domi-

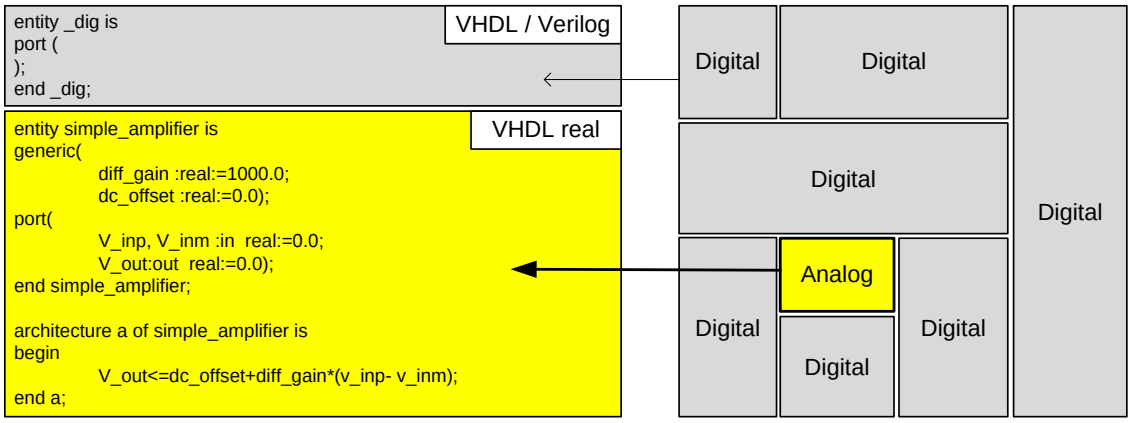

Figure 2.13: bottom up approach in digital centric designs [19]

nant analog content, for digital centric designs a traditional bottom-up approach can be a better solution. In this approach analog blocks are designed according to the defined specifications and are verified against these specifications using SPICE like simulators. Chip level verification of digital centric designs can be performed using behavioural models of analog portions. For instance, Fig.2.13 depicts a dominant digital content design where digital blocks are defined using HDLs and analog part of the design is modelled using Real Number Modelling (RNM). The most significant achievement of this approach is that it provides the possibility to use event driven simulators in chip level verification.

To conclude, since analog and mixed-signal integrated circuit simulation and verification became a challenging task, using only SPICE like simulators is not anymore a rational choice. While digital verification is heavily automated, still analog verification progress cannot satisfy the demanded requirements. Moving forward from chips with thousands of transistors to the chips with millions of transistors, several useful approaches have been deployed in digital verification (see chapter 2.4.1). In order to enrich analog and mixed-signal verification quality as well, using abstract models of analog circuits is inevitable. These abstract models are utilized in different levels and design steps. The time-to-market for leading semiconductor companies has been highly decreased taking advantage of abstract models in either bottom-up or top-down manner.
It is significantly important to slightly abandon the detailed results of SPICE level simulations in some levels of verification process and use a concise but comprehensive simulation model instead. In this manner, beneficial digital like strategies are employable in analog circuit verification.
One of the prominent advancements of digital verification is transaction level modelling. As it is discussed in section 2.6.1, TLM provides an abstract view from protocol implementation in the verification process. The aim of this work is to investigate on a way to utilize the concept of transaction in analog circuits verification. The *Analog Transaction* concept is introduced in section 3.

Chapter 3

Analog Transaction Level Modelling

Todays design methodologies are highly affected in two different directions. In one hand, it is required to inspect electrical behavior of the design to avoid physical defects and later product respins. Detailed design representation is necessary in order to verify it in this level. On the other hand, time-to-market pressure forces designers to use abstract representation to be able to overcome the challenging task of complex mixed signal circuit verification.
Today between 70% and 80% of designs are mixed signal designs and therefore mixed signal verification takes enormous amount of time to be accomplished [19]. To be able to use digital like simulators and therefore speed up the verification process, it is crucial to consider analog and digital portions of the design holistically. This highlights the demand for novel strategies in order to verify an analog IP in the SoC context.
The aim of this project is to improve the existing mixed signal verification methodologies to enhance this holistic design consideration.

3.1 Mixed-Signal Verification Challenges

As it has been recorded, more than 50% of design respins at 65 nanometer and below are the overcome of mixed-signal designing [20]. The consequent wasted time and high costs of product respin, bring about the demand of new strategies for SoC verification. It is not possible any more to simply assume the practicality of old strategies in SoC level mixed-signal verification. To have an overview of old mixed-signal SoC level verification strategies, brief description of two of them are stated in this section.

Black Box Approach One of the common methods to verify a mixed-signal design with dominant digital content is called *black box verification*. In this approach a pre-

verified analog block is integrated into a bigger design using its highly abstracted model. The abstract model contains the interface description only in many cases. This is how simulation speed degradation can be avoided but this problem arises that verification quality degrades instead. Today's mixed-signal SoCs are not only more complex in terms of added analog blocks, but also in terms of complicated interconnections and feedback loops between analog and digital portions of the design. It is not reliable any more to replace analog portions of the design with a black box model and perform SoC level verification. [20]

Capture and Replay Approach In this approach after clearly specifying analog portions of the design, the boundary of the analog portion from the SoC simulation is extracted during some particular simulation time. The extracted waveform is then replayed back as an input to analog/mixed-signal (AMS) simulation of the analog portion. Although it is possible to detect some functional errors in design using capture and replay approach, it is still not reliable since reactive feedback between analog and digital parts of the design is missed. Moreover the process is carried out manually which brings about limitations in generating test cases. [21]

3.2 Mixed signal Verification Facilitation

SoC level verification of mixed-signal ICs has been getting more attention in last years. Several approaches has been proposed by researchers in order to enhance the verification process. Since digital verification teams were able to build highly automated testbenches, it was time to introduce more digital like and consequently more automated techniques for analog and mixed-signal circuits verification. Although analog verification with SPICE is still a golden standard and cannot be ignored, to achieve better simulation speed, different levels of design abstraction are used to model an analog subsystem. Taking into account SPICE and fast SPICE simulation next level of abstraction, *analog behavioural modelling* (with Verilog-AMS and VHDL-AMS), improved simulation speed significantly. Furthermore, *Real Number Modelling (RNM)* and pure digital models are the other approaches to describe the analog subcircuits in higher levels of abstraction and therefore to achieve higher simulation performance. Moreover, another crucial factor to choose an appropriate abstract model is the required effort to build a simulation environment. To clarify more a comparative chart is shown in Fig. 3.1.
The chart shows the distinction between mentioned approaches in terms of required effort for simulation setup in (a) and performance trade off in (b). As depicted in this figure, although RNMs or pure digital models are less accurate models but less effort is required to build a simulation environment using these models than AMS models. Therefore this is the most apparent advantage of these models in full-chip verification.

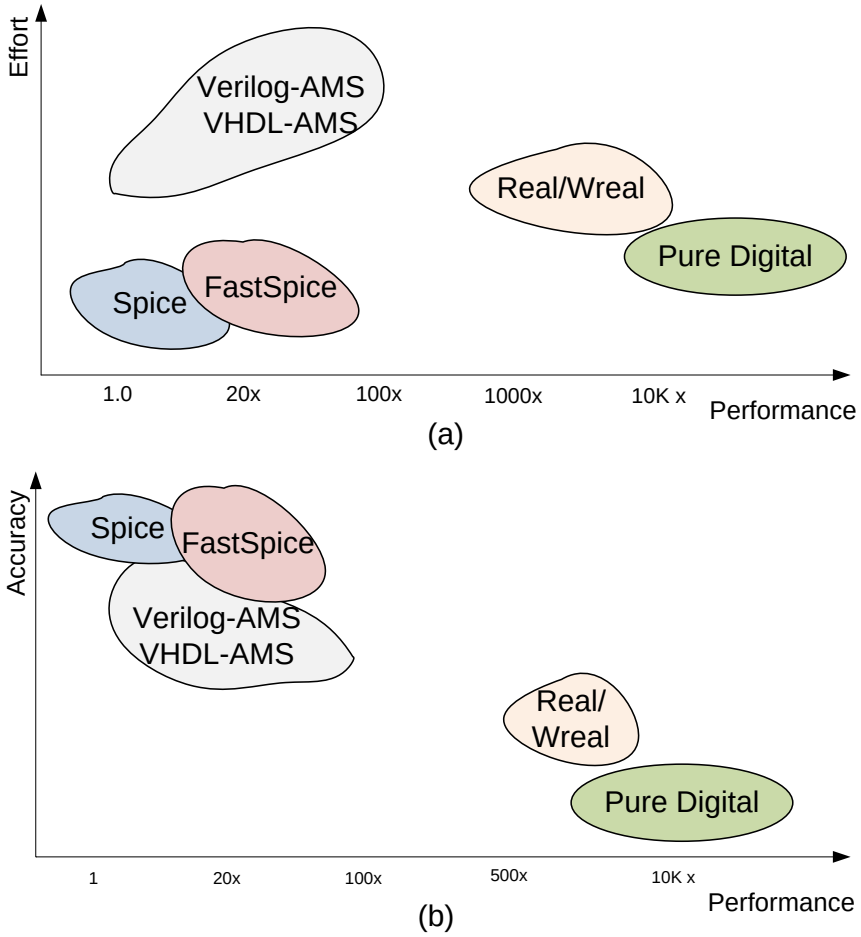

Figure 3.1: A comparison between different modelling approches in mixed-signal verification in terms of required effort and performance [22]

Analog behavioural modelling In general analog behavioural modelling approach is used to create a module which encapsulates high-level behavioural description of an analog or mixed-signal subsystem. In order to enable designers to write this behavioural models some specific languages were created like *Verilog-AMS*, *Verilog-A* and *VHDL-AMS*. These analog mixed-signal (AMS) languages offer both continuous-time and event-driven semantics for verification task. Apparently analog designers are the best choice to write the analog behavioural models. Although they are familiar with their own analog design, they usually lack Verilog or VHDL knowledge to write these models. Similarly, digital designers with enough knowledge of these languages are not able to write an accurate model since they know less about the analog circuits [22]. As it is clearly depicted in Fig. 3.1, AMS model simulation needs most set up effort. In one hand, it is possible to write these models to gain more simulation speed, at the sacrifice of accuracy. On the other hand as it is shown in Fig.3.1 they have the capability to be modelled with the accuracy close to SPICE like simulators depending on the application. However it is important to notice that over-idealized models can bring about convergence issues for unskilled modellers. [23]

Real Number Modelling "In real number modelling, also known as real value modelling (RVM), values are continuous -floating-point(real)numbers- as in the analog world. However, time is discrete, meaning the real signals change values based on discrete events" [23].
It is possible to achieve simulation performance near digital simulation speed taking advantage of RNM. In chip level verification, where it is usually in an abstract level and therefore not deep in implementation details, at the expense of loosing some accuracy, time and costs can be highly reduced using RNM. This approach is restricted to signal flow and therefore benefits from the absence of troublesome convergence issues. From Fig. 3.1 it is possible to notice that in compare to AMS models, this approach requires less **effort** and provides higher **performance** when verification goal is apart from implementation details (**accuracy**).
Three HDL languages support RNM such as VHDL (with real type), SystemVerilog (with real type) and Verilog-AMS (with wreal type)
Simulation performance comparable to what is already exist in digital verification techniques, makes it possible to apply highly automated digital techniques like verification planning, random test generation, coverage and assertions in analog verification area by using real number models. This is the most well-liked verification capability which can be achieved in analog mixed-signal verification.

3.3 Analog Transaction

A sample DUT with RNM blocks and RTL blocks is presented in Fig. 3.2. In this case the DUT has both digital and real-valued pins. The existing approach to drive

real-valued pins of DUT, is to hard code stimuli in the test (direct way). However in [2] the proposed technique leads to a way in which analog pins of the DUT are driven using constrained random stimulus generation.

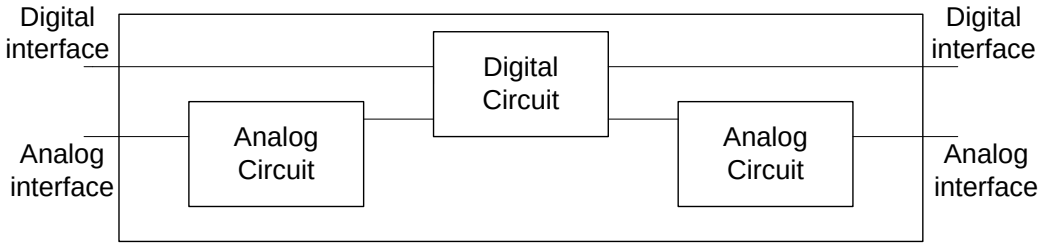

Figure 3.2: A mixed-signal design with analog and digital sub-circuits interfacing with each other and outside world

The proposed method in [2] to drive analog pins of depicted sample DUT in Fig. 3.2 resulted in a new concept which is *Analog Transaction*. Transaction concept is defined in section 2.6.1 and to clarify more it is noticeable that transaction is a data structure containing parameters. In a UVM based testbench, data fields within transaction are randomized and are used in driver which implements the protocol separately and wiggles the DUT pins.
To be able to extend the concept of transaction in analog domain, a term replacement from "protocol" in digital world to "shape" in analog world was suggested in [2]. Analog signals can have different shapes for example harmonic, linear or cubic spline. To be able to describe or generate these analog shapes besides the name of the shape we have to define parameters. This brings about a request to use analog transactions to generate analog waves with defined parameters and consequently in TLM way. In other words in a UVM based testbench data structure including parameters is pass to the driver and according to the specific numerical algorithm driver reflects those parameters to generate the desired analog wave.

3.4 MDV adoption to Analog environment

UVM, as a popular verification methodology, enhanced the verification process of complex SoCs. The most efficient aspect of this methodology is the application of metric driven verification. In order to enhance analog verification one should think about MDV techniques and the possibility of adopting them to an analog environment. There are some limitations in this process which are explained in [24]. In this section some of the most important limitations are discussed briefly.

Analog vPlan and Coverage Planning for verification (vPlan) is a beneficial technique which is more complicated when talking about analog features. As vPlan must

capture the defined features of the design, it has to be richer to adopt analog features as well. Moreover, measuring those analog features could be a sophisticated task. This is because analog properties including amplitude, gain, frequency or some other similar values are different than logic values naturally.

Simulation performance In order to adopt MDV and therefore, take advantage of its ability to a run large number of simulations automatically, analog designs have to be modelled using AMS languages or RNM. This is because the SPICE level simulation is slow in comparison to digital simulation.

Constrained random stimulus To explore DUT different behaviours it is required to generate both digital and analog random stimulus. Efficient randomization of analog inputs to the design is a major request when adopting MDV to analog environment.

Self-checking A self checking testbench can determine that the design behaves as planned or not. In digital designs this is carried out by monitoring the output pins of DUT and comparing with expected output in Scoreboard. This process is more complicated when checking an analog design. For further studies on limitations of adopting MDV to analog design it is suggested to study [24].

Chapter 4

Testbench Implementation

The goal of this work is the improvement of existing verification methodology and testbench structure of mixed-signal designs. In order to do so, a testbench structure is proposed in which analog interfaces of the depicted DUT in Fig. 3.2 are driven taking advantage of transaction level modelling. The proposed testbench structure is responsible for generating random stimulus in order to shape various analog waves and sending them to the DUT's analog interface automatically.
The aim is to generate more different analog waves in transaction based manner in which the driver component receives transactions from sequencer and drives DUT pins. The diver is implemented to behave independent of the defined algorithm in transaction. In other words, regardless of what is intended by the user for the "shape" of analog wave driver reflects the received transaction and eventually generates the intended shape at the inputs of the DUT.
Moreover, generated components in the proposed verification environment can support and deliver transactions with various data structures. This enables the user to implement the application specific algorithm and reuse the generated SystemVerolog library in testbench environment.
In future studies, implementation of a testbench with self-checking capability is of vital importance.

4.1 Related Work

In this thesis work the goal is to design a UVM based testbench through which analog input pins of a sample DUT are driven using the same strategy as digital input pins. Since different verification methodologies are used to verify digital and analog blocks, it is crucial to have a holistic view of both digital and analog sub-blocks of the design at chip level verification.
As it was presented in [2], to be able to use advanced verification techniques like randomization and therefore to avoid writing analog real-valued input stimulus in a directed way, transaction level modeling is extended into analog domain.
In [2], analog transaction was introduced for the first time. The proposed technique in [2] generates analog stimulus through passing data structures to the driver. As it is discussed in section 3.3 transactions are data structures containing particular parameters. Via sequencer and driver communication transactions are passed to the driver. Driver wiggles DUT's pins according to the specified protocol and by decoding parameters within a received transaction. Hence "transactions offer abstraction of the protocol." [2]

UVM_seq_item class holds parameters which are required to define an analog wave (like slope and value of a certain point in time for a linear signal). During run task of the driver, when it sends a request to the sequencer for a new sequence item, those parameters within sequence item will get randomized. As an example, generation of a harmonic analog signal was presented in [2]. The randomized spectrum of signal was delivered by the *UVM_seq_item* class and the harmonic signal was obtained through performing Inverse Fourier Transformation in driver class.

It is SystemVerilog's Direct Programming Interface (DPI) which provides the possibility to use various algorithms written in C language. SystemVerilog like many other modern languages provides the feasibility of interconnection with libraries written in C. SystemVerilog layer and foreign language layer are completely isolated. Communication between these two layers is through function calls between languages.
In proposed structure in [2] from SystemVerilog layer driver calls a C wrapper for FFTW [25]. It was through DPI that driver passed the randomized spectrum to the foreign language library (FFTW) and received transformed data subsequently. Fourier transform implementation in C is more easy and efficient than writing it in SystemVerilog. The transformed data which was received from C layer was the signal representation in time domain and was sent to the pins of DUT by driver component. The connection between driver in SystemVerilog layer and FFTW in the foreign language layer is shown in Fig. 4.1.
The most important advantage of proposed method in [2] is that by randomly generating complex analog stimulus, it is possible to detect design bugs much faster than writing direct tests.

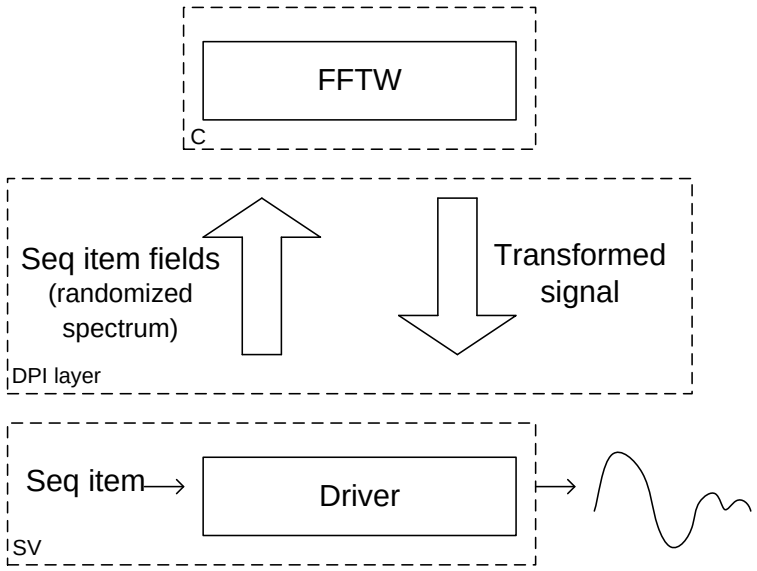

Figure 4.1: Driver communication in SV layer with FFTW in foreign language layer

4.2 Key features of this work : Extendibility and Separability

In this section the main purpose of this work is illustrated. Based on the information from previous work in this area, described in section 4.1, the aim of this work is to build a testbench which is fully reusable in different projects when other shapes of analog wave are required to drive a DUT.
UVM classes in proposed structure in [2] were implemented to perform an algorithm (like inverse fourier transformation) with its specification. To clarify more, *UVM_seq_item* class was implemented to hold parameters which are required to generate a specific analog wave. In described example in section 4.1 to generate a harmonic signal, signal spectrum is needed. In one hand parameters within *UVM_seq_item* class were defined to deliver signal's spectrum. In other words, spectrum was used as sequence item field and was sent to driver from sequencer. This means that sequence item field was designated to a complex valued dynamic array describing the spectrum of a signal.
On the other hand, driver class is responsible to decode the received transaction via *seq_item_port*. Driver and sequencer are parametrized in order to use a specific transaction type for communication. They communicate through sending *REQ* and *RSP* of type *seq_item* class. In described example in section 4.1, the transaction type was defined to be signal spectrum. It contained a complex valued array field and duration period field. These fields were decoded in driver class (using SystemVerilog DPI it was possible to perform the inverse fourier transformation using fftw) and eventually the transformed signal was sent to DUT pins through virtual interface. The implemented driver was able to decode those defined fields within the transaction exclusively. This brought limitation

in terms of flexibility and usability.

In this work, the main focus is to implement a UVM based testbench in which transaction type is independent from algorithm specification. Moreover, with this algorithm independent transaction type, driver class is also implemented in a generic way. In this manner it is possible to generate various analog signal shapes like linear, cubic spline or any other shapes using the same structure. This is how mentioned limitations (flexibility and usability) in proposed structure in [2] can be resolved. In many verification projects it is required to generate different shapes of analog waves. To have a fully flexible testbench with the ability of generating different shapes of analog waves it is necessary to define an unconstrained transaction level communication between UVM components. For more illumination, in Fig. 4.2 three different approaches are illustrated. In this figure (a) is a digital driver class. It receives a sequence item from sequencer and decodes it using a state machine. Sequence item class contains application specific data fields. According to the defined protocol, driver uses sequence item fields and generates the digital wave. The proposed structure in [2] is shown in (b). The driver class is shown with basic arithmetic operations. This means that driver performs a particular numeric algorithm to generate the output analog signal. In this work, basic arithmetic operations are used to represent an algorithm. Driver receives application specific sequence item

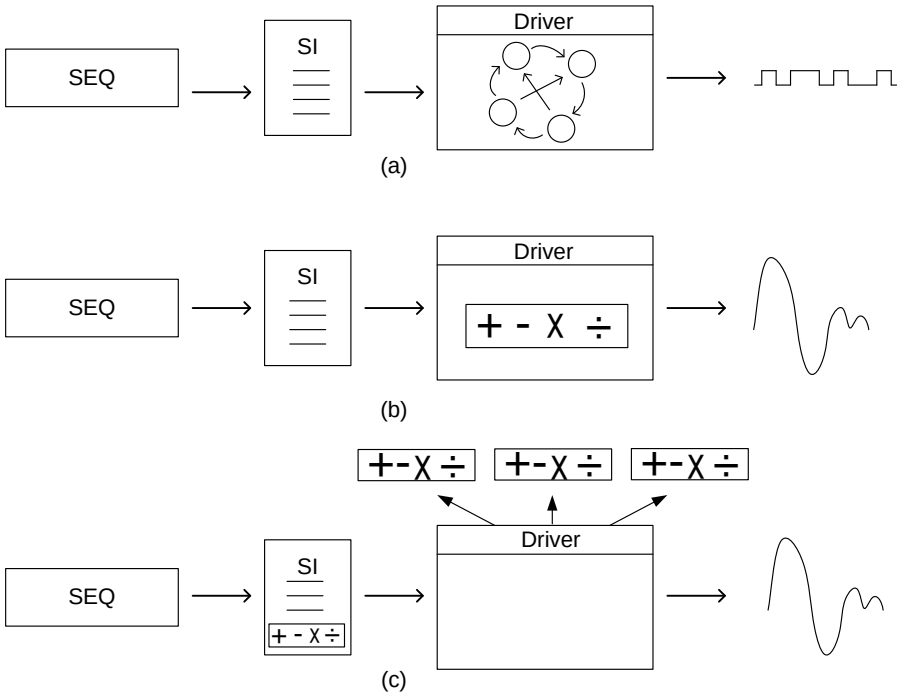

Figure 4.2: UVM based verification environment (a) digital, (b) proposed structure in [2], (c) improved structure

fields and performs the decoding process based on the user defined algorithm. Desired analog output wave is generated eventually.

The proposed structure in this work resolves the limitations in (b) by using the (c) structure. Here, the sequence item class is implemeted to hold different data structures. It allows all kinds of algorithms with different required data structure to be plugged in to the algorithm layer. A new filed in sequence item class plays a key role when the driver receives the transaction: Algorithm name. The algorithm name indicates which algorithm is selected in the test. Driver communicates with the intended algorithm subsequently.
It is important to mention that the driver is independent of user defined algorithm. For decoding process SystemVerilog DPI provides the accessibility to foreign language algorithms. Driver in communication with algorithm specific classes remains independet and gets the generated stimulus and drives the DUT.
Key features of this structure can be enumerated as extendibility and separability.

Extendibility In order to save time and therefore meet the project deadlines it is always beneficial to reuse the code between projects. In mixed-signal verification projects it is not far fetched to need the analog wave to simulate the design from one project to another. By using improved structure in (c) it is easily possible to define and add a new demanded algorithm to the algorithm layer. In this manner, code reuse between mixed-

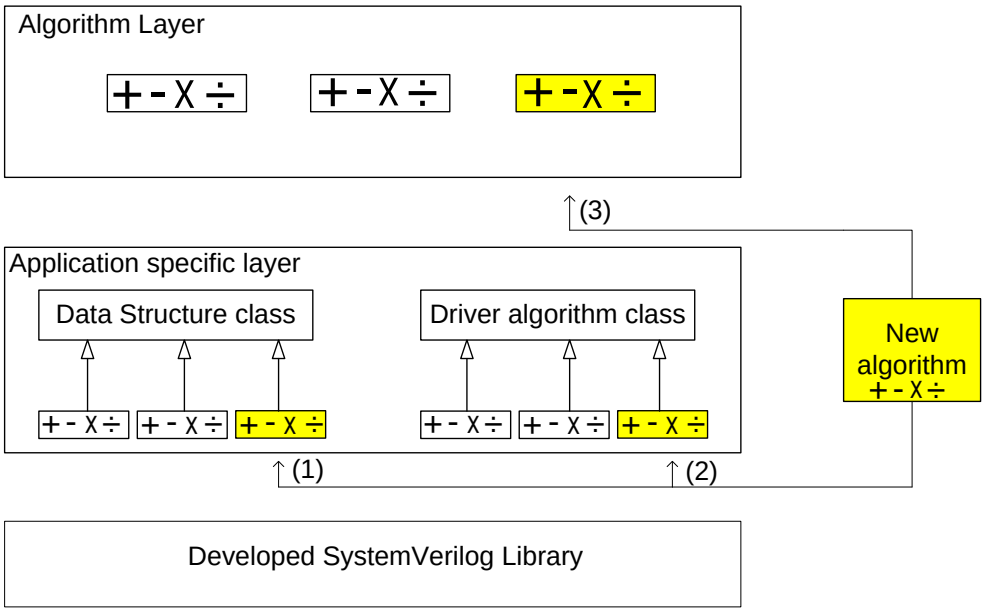

Figure 4.3: Key features of this work : Extendibility, Separability

signal verification projects is possible since this structure is fully extendible. This feature is shown in Fig. 4.3. When it is required to add a new algorithm it is possible by adding algorithm-specific files (user_defined components) to the environment. The developed SystemVerilog library shown in this figure is compatible with all SystemVerilog supporter event driven simulators. The interface through which SV library communicates with algorithm layer is predefined and enables the verification engineer to access every new defined algorithm in algorithm layer. This is the aimed extendibility feature of this work.

Separability Code reuse between projects is an essential requirement and is achievable through accurately separating project-specific components and fixed components. When it is time to reuse a code the less time to devote for updating facilitates the process. In Fig. 4.3 fixed components will remain unchanged when adding user-defined components. This is the planned separability feature in this structure.
Three layers are shown in Fig. 4.3 : algorithm layer, application specific layer and developed SystemVerilog Library. User who intends to add a new algorithm has to do the following tasks:

1. Add algorithm_specific *data structure* class to application specific layer.
2. Add algorithm_specific *Driver algorithm* class to application specific layer.
3. Add the algorithm to the algorithm layer (or a C wrapper for those algorithms implemented in other languages).

4.3 Class diagram

To have a comprehensive inspection of the whole verification environment, a class diagram of the generated environment is shown here. The verification environment classes and the interrelation between them is shown in Fig. 4.4. Moreover, Fig. 4.5 shows the data item classes that are processed by the verification environment.
According to described patterns in the book *Design Patterns: Elements of Reusable Object-Oriented Software* [26], Fig. 4.4 includes the *strategy* pattern. This pattern is applicable when different algorithms are available to use by the user but only one is used at different times. Different classes can be used to encapsulate different algorithms and the encapsulated algorithm in a class is called strategy.
Using the strategy pattern it is possible to define many classes with different behaviours. It is necessary to introduce a common interface to call/configure classes through it. In Fig. 4.4 this common interface is the *UVM_driver_algorithm* virtual class. *UVM_driver* uses this interface to call user_defined algorithms supported by the interface. It has a reference to *UVM_driver_algorithm* class.
Strategy pattern provides various conveniences for the designer. Although there is a

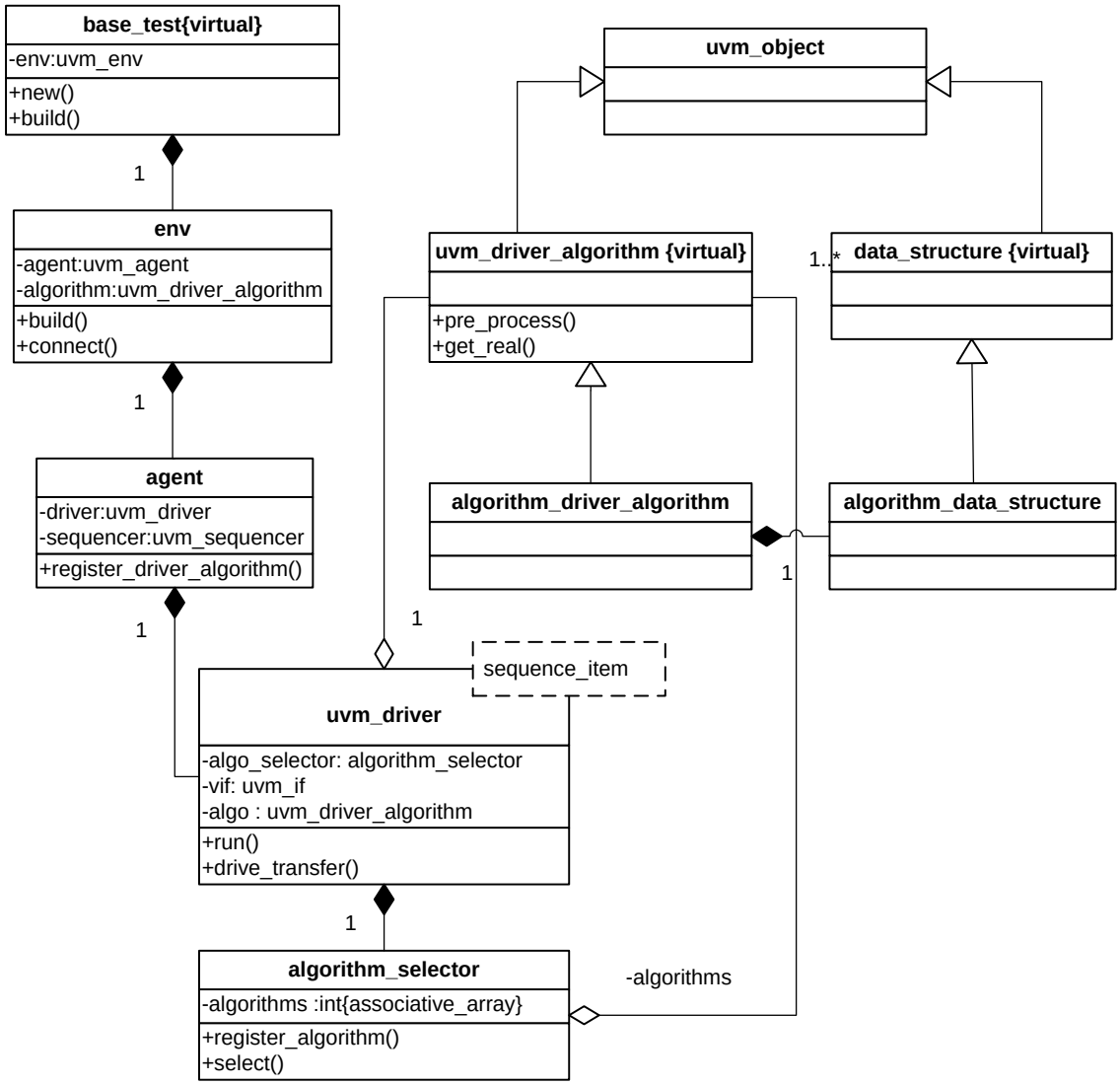

Figure 4.4: Verification environment class diagram, Strategy pattern [26] is used in class design

possibility to create several *UVM_driver* classes using *inheritance* and therefore support several algorithms, the result will not be flexible and extendible. In addition, the code will be a mix of driver code and algorithm implementation which makes it complicated and hard to understand. Using strategy pattern and algorithm encapsulation brings about the possibility to extend the algorithm classes independent of driver class. Moreover, this way dynamically changing the algorithms is achievable.

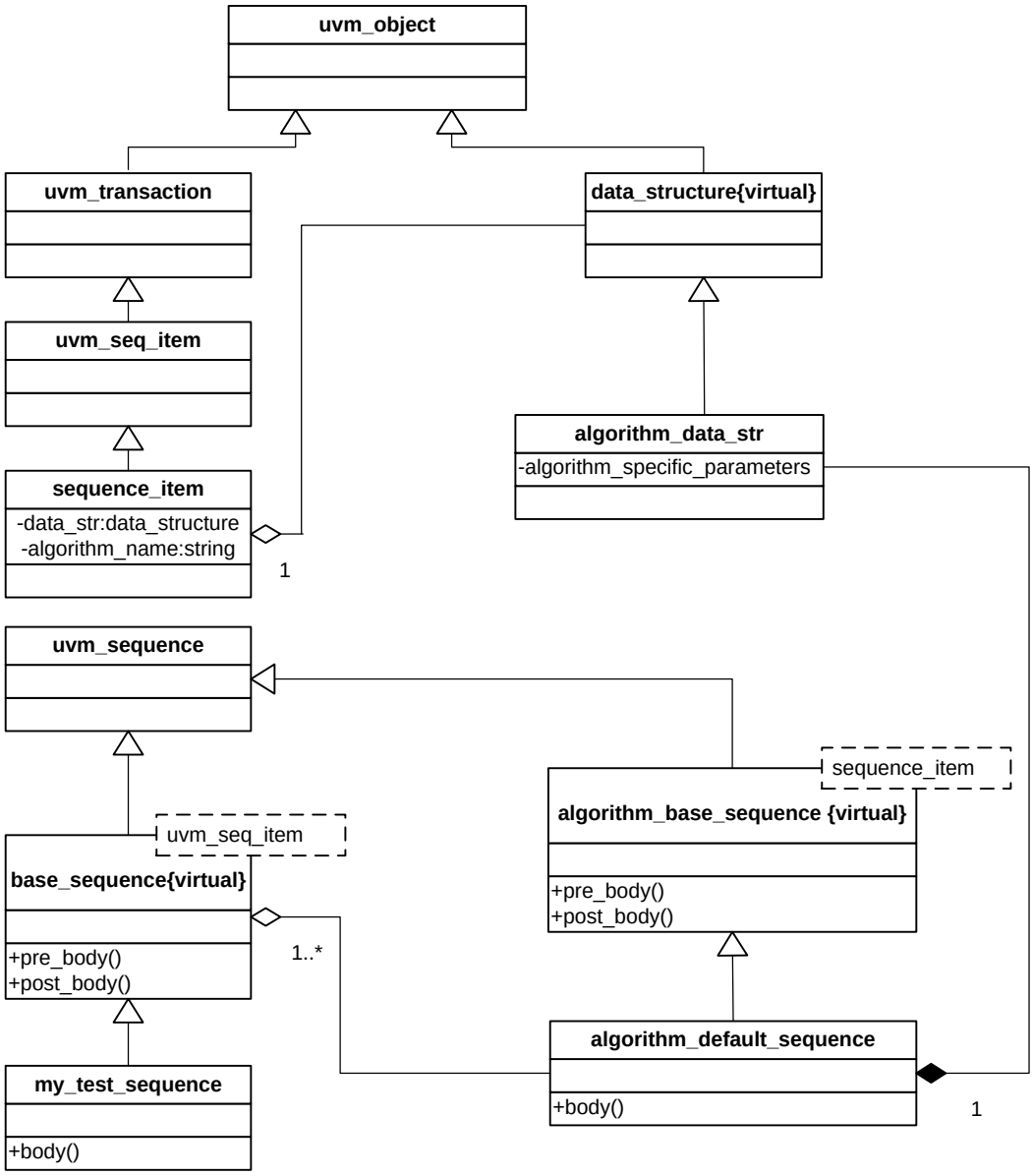

Figure 4.5: Data item class diagram

The class diagram shown in Fig.4.5 depicts transaction classes. On one hand *my _test_sequence* class is the sequence which is invoked from the test and on the other hand *algorithm_default _sequence* class is defined by the user. Upon a request an *algorithm _default_sequence* item is created. An instance of *algorithm_data_str* class lives in *algorithm_default_sequence* class which holds algorithm specific parameters and supports implementation of algorithms with different data structures.

4.4 Driver

The driver class as a subclass of UVM_driver base class is a fixed component. This means that by adding more and more algorithms, the alike driver code is still reusable. Driver class together with driver_algorithm class are fundamental components in order to accomplish thesis scopes. Improvement and generalization of existing techniques are accomplished by implementing an algorithm independent driver component which contains a variable of type driver_algorithm class. The algorithm_selector class is also declared in driver and delivers the registered algorithm to the driver upon request.
During the run_task of the driver, a request for the next transaction is sent to the sequencer. The virtual protected drive_transfer task is responsible for converting transaction level activities to signal level activities. The received transaction during the run_task is passed to the drive_transfer task. The input argument to drive_transfer task is of type seq_item_class.

Drive_transfer task This task of the driver class plays an important role and wiggles the DUT pins via a virtual interface. First and foremost it is necessary to retrieve one of the fields within the transaction. This field indicates the registered name of the algorithm and is called algorithm_name. When a user intends to generate a new analog wave the desired algorithm name has to be registered in algorithm_selector class. By invoking the algorithm_selector class in drive_transfer task it is possible to retrieve the created object of the algorithm class to which algorithm_name is pointing. The retrieved algorithm from algorithm_selector class is assigned to a variable of type driver_alforithm class. Through this step all the instances of registered algorithms in algorithm_selector class are accessible.
It is highly important to notice that the instance of all algorithms in the algorithm library have to be created and entered in an associative array. This array is defined in algorithm_selector class. A UVM_fatal is generated in case the algorithm is not registered.
A variable of type driver_algorithm class provides communication between the driver and the instance of the algorithm defined within the transaction. This variable provides the access to the methods of the algorithm class. Algorithm classes all have similar methods since they are all extended from a same virtual parent class [4.6].
Furthermore, the pre_process method is invoked from the algorithm class and the data

_structure is passed to it without manipulation. Data_structure holds required algorithm specific parameters to generate a new analog wave.
Eventually, through invoking the get_real method the real valued data is acquired. These real valued data is used to drive DUT pins through a virtual interface subsequently.

4.5 Data Structure Class

The data structure class is a virtual class extended from *UVM_object* base class. This virtual class is overridden by user_defined data structure classes. In other words, a new algorithm needs a data_structure specific class which is extended from virtual data structure class. Data structure class plays a key role in achieving algorithm independent structure of verification environment.
To clarify more about the importance of this class, it is necessary to have a short review of a related work. In [24] a similar verification environment was implemented and a sine wave was generated. It was controlled by defined signal properties in sequence item class. Signal frequency, amplitude, phase and DC bias of the signal were defined as sequence item fields. Eventually a sine wave was generated by the driver in *wire UVC* and was connected to real value ports of the DUT.
The main advantage of data structure class implemented in this work over proposed structure in [24] is that by using data structure class it is possible to generate much more complicated algorithms and generate various analog waves that might be required in different verification projects. Sine wave and other similar simple algorithm implementations are accomplished in this work besides more complex algorithms like Inverse Fourier Transformation. In addition, implementation of even more complicated algorithms than IFT can be achieved using data structure class.
To achieve this, object oriented programming concepts like *inheritance* and *polymorphism* are used to implement this class and its subclasses.
User who intends to update this verification environment with a new algorithm has to write the algorithm specific data structure class which is a child class of data structure virtual class. This user_defined class inherits all defined attributes within the parent class. Therefore data structure class is the *parent class* or *super class* of all user_defined algorithm dependent data structure classes. However, the virtual data_structure class is only an empty prototype definition. Each subclass of this class defines algorithm specific attributes. Hence, a variable of type data_structure class can hold different objects of user_defined data_structure classes.

Taking advantage of polymorphism it is possible to have different objects belonging to different types and responding to method, field or property calls of the same name. Algorithm specific data structure is defined by the user in the inherited child class and its corresponding parameters can be assigned to fixed or randomized values when an object of the class is generated.

4.6 Driver algorithm class

Driver_algorithm class is a virtual class which is inherited from *UVM_object* base class. A variable of type driver_algorithm class exists in driver class and holds an instance of user_defined algorithm. Driver_algorithm virtual class, as a parent class, includes two pure methods *pre_process* and *get_real*. All user_defined algorithms have to override these two methods in their definition. Extension from this class brings about the possibility to have different instances of different types and invoke their methods using the same variable.
In an algorithm specific driver_algorithm class, pre_process function receives an input argument from driver class. This input argument is the algorithm related data structure class.
For instance, in order to generate a sine wave signal period and amplitude are defined as properties of sine data structure class. These properties are totally different in another algorithm specific subclass of data structure class. The pre_process function within sine driver_algorithm class receives all those properties. This function is responsible for data preprocessing. A variable of type sine data_structure class is defined in this class and in pre_process function received properties are put into this local variable. A fault report is generated if received data structure is empty.

Moreover, in some algorithms like cubic spline preprocessing is done in a broader way. To be able to generate a cubic spline analog wave cubic spline specific data_structure is an input argument to the pre_process function. The related C algorithm processes an array of time_value elements during two steps. In other words, by calling DPI_C interface in pre-process function, C code receives data from SV code and transfers the processed data back.

Only after preprocessing it is possible to call DPI_C interface within get_real function. Local variables holding preprocessed data are output arguments and are passed to the algorithm layer in get_real function. Return argument is real valued data structure to be consumed by driver class.

As it is shown in Fig.4.3 this class is in application specific layer which is a mediate layer between SystemVerilog layer and algorithm layer. It is through this layer, and specifically through subclasses of driver_algorithm class, that SystemVerilog code is connected to the foreign language environment. User who intends to add a new algorithm to the algorithm layer, has to add an algorithm specific driver_algorithm class to the application specific layer. DPI interface is defined specifically in each subclass and allows inter_language function calls. By calling a C function through DPI, actual data object which needs to be processed by the algorithm is passed by its reference (pointer). Input arguments with only small values can be passed by value. Obviously DPI declaration in this class is optional as long as algorithm is not implemented in a foreign language.

4.7 Algorithm selector class

The Algorithm_selector class is an extension of *UVM_object* base class. To add a new algorithm to the algorithm library it is necessary to register it via algorithm_selector methods. Otherwise, driver cannot retrieve the created instance of the algorithm afterwards.
This class contains two methods : register_algorithm function and select function and an associative array in order to store all user_defined algorithms. An associative array is the best choice to store a number of dynamically changing elements. The associative array within algorithm_selector class gets index of type string which is the algorithm name. This algorithm name is passed down to this class from upper classes in the class hierarchy.

It is first in the env class that a new algorithm has to be declared and constructed. The constructed instance of the algorithm is passed down to a particular agent during the connect phase. In fact the created instance of the algorithm and its name are passed to the function register_algorithm. It is through algorithm_selector type variable in driver class that the agent invokes the register_algorithm function and transfers the received algorithm to this function.

In register_algorithm function a new algorithm is registered in an associative array using the algorithm name as an index. An instance of all existing algorithms can be created and put into the associative type array.
Furthermore, in driver class the select function is invoked in order to retrieve the corresponding instance of the algorithm within the associative array. The input argument to the select function is the string type algorithm name. The *exists()* method of the associative array checks if the received string exists within the stored indices of the array. A UVM_fatal message is generated if the input algorithm name is not registered via algorithm_selector class yet.
It is important to mention that since all user_defined algorithm classes are subclasses of driver_algorithm virtual class, it is possible to store all into a single associative array. Taking advantage of polymorphism, through superclass variable (driver_algorithm), subclass properties and methods are accessible. This is a powerful method in object oriented programming techniques.

4.8 Algorithm Layer

In this work several algorithms are examined in order to inspect the impact of differences between them. There are many existing algorithms mostly written in C and through SystemVerilog direct programming interface it is simply possible to reuse them and generate various analog waves. DPI by following the so called black box approach brings about an absolute separation between specification and implementation. Through DPI declaration in driver_algorithm class, it is possible to invoke a C function and pass Sys-

temVerilog algorithm specific data structure to the C code. The processed data can be retrieved from the C code afterwards.
To inspect the proposed verification structure different interpolations have been implemented and added to the algorithm layer. Linear, Cubic Spline and Rational interpolations are all accessible through DPI. Interpolation algorithms are written in C language and randomized data structure required for these interpolations are passed to them from SV code. To perform an interpolation required data structure is a set of points. A SystemVerilog point array (containing time, value real type pairs) is generated in a randomized manner and the algorithm in C receives it subsequently.
The proposed verification environment has been inspected for more complicated algorithms as well. The fastest Fourier transformation in west (FFTW) is a C subroutine library for Discrete Fourier Transformation computation. An Inverse Fourier Transformation is implemented using this library. To review FFTW advantages concisely it is important to enumerate its speed, particularly for purely real value data. To perform this transformation a C wrapper is used and algorithm specific data structure class is defined. In the data structure class a complex value array is defined to generate signal spectrum. The typical way in UVM based testbenches is to randomize the stimulus which is used to drive the DUT pins. In this example, the complex value data gets randomized and the randomized spectrum is transferred to the FFTW C code via a C wrapper. The C Wrapper is called within Inverse Fourier Transformation algorithm class. The transformed data is returned to get_real function afterwards.
Moreover, to investigate more algorithms with different data structures the algorithm layer is extended to also include MATLAB environment. This time a completely new algorithm is chosen to be implemented in MATLAB. In this algorithm the generated signal is the transfer function of a finite impulse response (FIR) filter. To generate this signal the FIR data structure class is defined. This class includes filter order, filter cut_frequency and input stream.
As it is also mentioned in section 4.5 the strength point of this work, in compare to previous works in this area, is the development feasibility of algorithm layer with entirely different data structures. It is possible to notice that each of these added algorithms to the algorithm layer requires different data structure. A set of time_value pairs is needed for interpolations, an array of complex values is required for IFT algorithm and to see the transfer function of an FIR filter it is necessary to define filter's sepecification.

4.9 SystemVerilog, MATLAB Integration

MATLAB as a powerful numerical computing system provides wide range of toolboxes and built-in functions and therefore a proper tool for algorithm development. Due to the fast algorithm developement in MATLAB, integration of SystemVerilog and MATLAB can accelerate the verification process significantly. Combining the power of MATLAB in signal generation, spectral analysis and image processing with SystemVerilog random stimulus generation, highly strengthens the verification procedure. The co-simulation is

achieved through building a C wrapper around MATLAB Engine and DPI coupling of SystemVerilog and the C wrapper. The co-simulation structure is illustrated in Fig. 4.6.

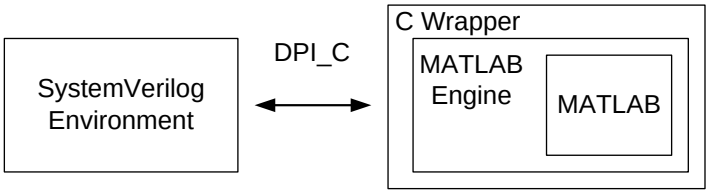

Figure 4.6: Co-simulation between SystemVerilog and MATLAB workspace

4.9.1 MATLAB Engine Library

It is possible to call MATLAB from C environment using MATLAB Engine library and employ MATLAB's computational capacity. MATLAB processing is operational as a separate process from C programme while data and commands can be send to and from MATLAB. MATLAB Engine library offers several routines all starting with the prefix *eng* and provides MATLAB environment controllability from a C /C++ programme.

Since MATLAB is generated to work only with single type data, MATLAB arrays, the C wrapper is responsible to manipulate the received SV array and put it into MATLAB workspace. The generated MATLAB specific array by C code has a *mx* prefix and is called *mxArray*. Data transferring from C to MATLAB is accomplished using the command *engPutVariable(Engine *ep, const char *name, const mxArray *pm)* with ep as an Engine pointer, name of mxArray in MATLAB workspace and pm for mxArray pointer respectively. *engGetVariable* is a similar command to read from MATLAB workspace. Other useful Engine library routines are attached to the appendix 1 of this document.

4.9.2 C Wrapper

SystemVerilog direct programming interface (DPI) is an interface between SystemVerilog and a foreign programming language. It consists of two separate layers: SV layer and foreign language layer [27].
Although the only foreign language which is currently supported by the SV is C language, it is possible to extend the foreign language layer to also include the MATLAB workspace. Data transfer between SystemVerilog and MATLAB workspace is realized in two layers: from SV code to a generated C wrapper and from C wrapper to MATLAB workspace. The opposite direction is also supported.
In this work a Sine wave is generated in MATLAB and the MATLAB array containing

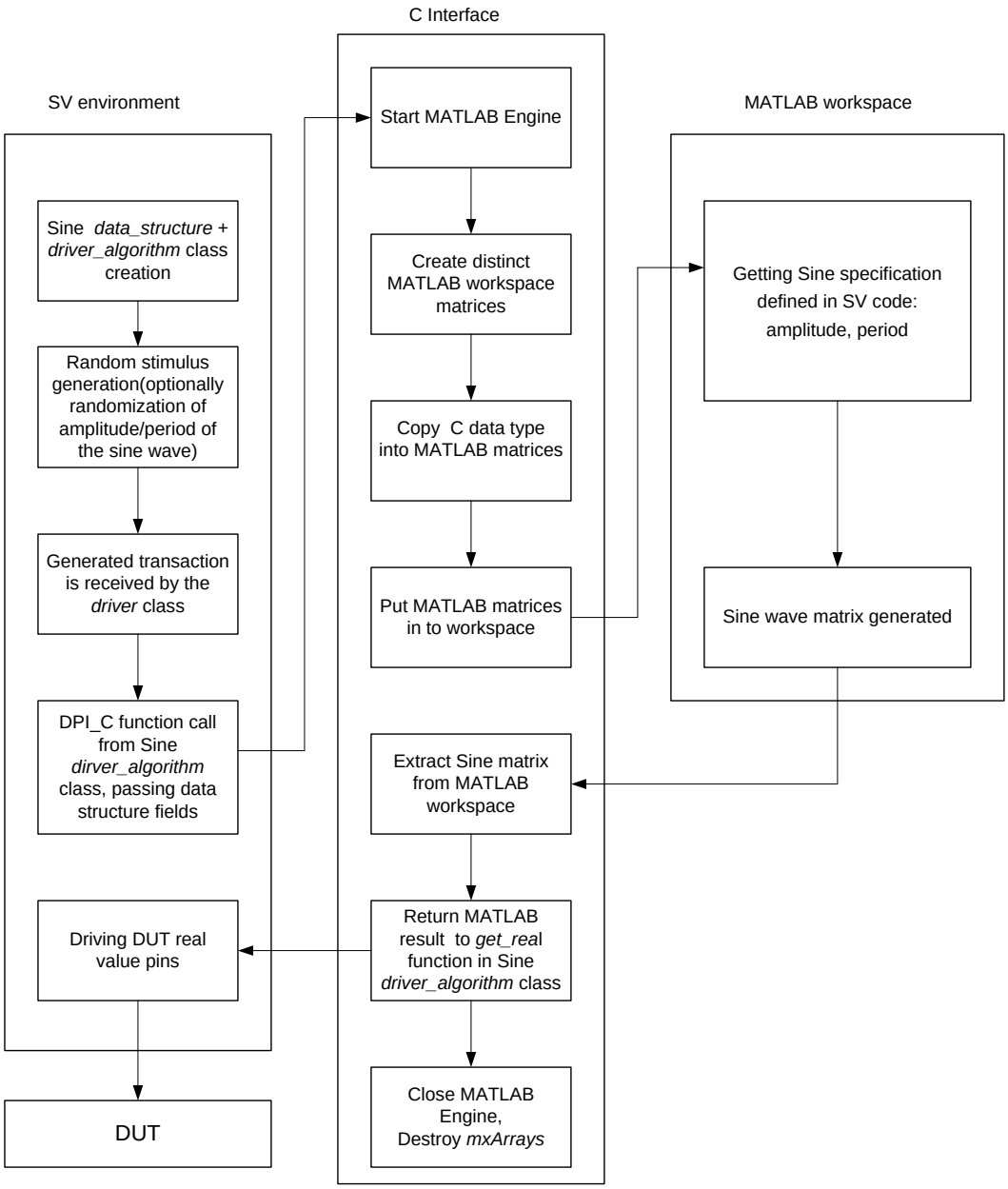

Figure 4.7: SystemVerilog and MATLAB communication

real values of the Sine wave is retrieved by the driver class in SystemVerilog layer. Sine specification like magnitude and period are part of the transaction which driver receives. User_defined data structure class contains these two fields and during the run time of the simulation, a fixed or random value is assigned to them. Through DPI connection in Sine diver_algorithm class generated data is sent to the C wrapper.
Furthermore a pointer to the MATLAB workspace is created. Data has to be manipulated in C environment in order to be traversed to MATLAB workspace. The wrapper generates *mxArrays* and copies the magnitude and period values to distinct mxArrays, consecutively. Once the Sine wave is generated in MATLAB workspace, the result is obtained and copied to a dedicated mxArray. The MATLAB engine can be closed and mxArrrays are destroyed in order to release the memory subsequently. The communication between SV environment and MATLAB workspace is depicted in Fig. 4.7 in detail. In Fig.4.8 the IFT output wave is shown and the resulted Sine wave from MATLAB is the adjacent wave.
SystemVerilog, MATLAB integration allows the utilization of effective algorithm implementation in MATLAB environment. It is also eligible to implement golden models in MATLAB and link them to SystemVerilog environment.

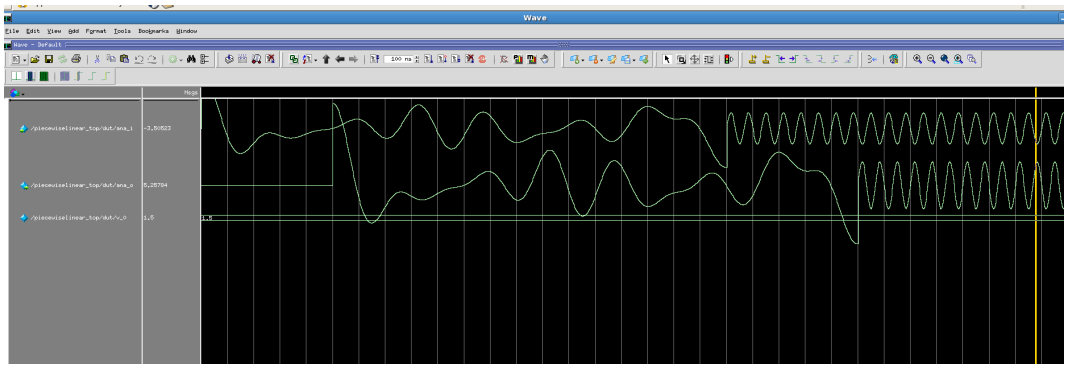

Figure 4.8: IFT and Sine wave in wave form window of Questasim simulator

Chapter 5

Application of the developed technique in a real world project

The technique in this project is applied in a real_world project to ensure the applicability of the developed verification structure.
The project is a three phase motor drive IC for automotive applications. The device is used together with an external micro controller and drives the gate source terminals of 6 external N_channel MOSFETs. This is how it controls both speed and direction of the motor. In Fig. 5.1 a high-level view of the device is depicted.
This device works in two basic operation modes:

1. Motor mode: in this mode motor driver amplifies the digital input signal Hx (x refers to the number of corresponding phase) and generates HGx and HSx signals to drive the gate and source terminals of high side transistors, respectively. In addition, LGx and LSx output signal are also generated through amplification of digital Lx input signal in order to control the low side transistors. This way current flow from battery to the motor is controlled by the micro controller.

2. Generator mode: in this mode the current flow direction is reversed. The device in this mode switches the transistors to reload the battery. Therefore, this mode is also called recuperation mode. The resulted technique of this work is used to verify the device in generator mode.

In generator mode the generated voltages by the motor are sensed by three ADCs (Analog to Digital cenverters). The input voltage to the ADC block is generated using proposed approach by this work. To verify the motor driver chip, the shown voltage sequence in Fig. 5.2 is applied to the real value interface of the device and the ADC behaviour is inspected.

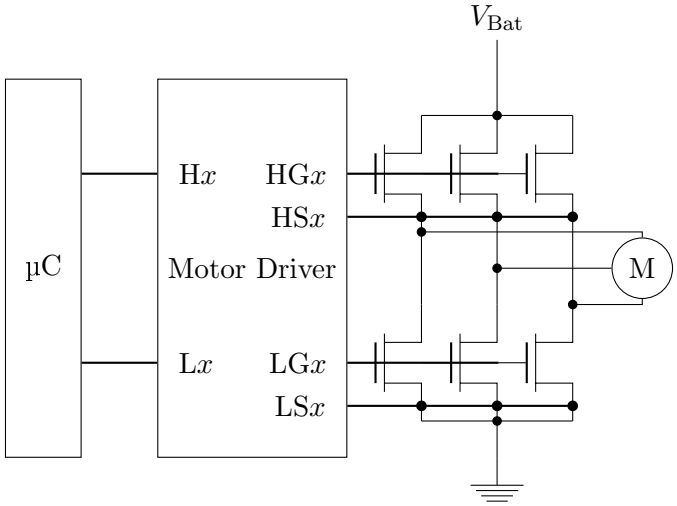

Figure 5.1: An abstract view of the application of the DUT used to qualify the achieved verification technique in this work. Current flow to the motor is controlled by the external micro controller.

The depicted voltage sequence is applied to the real number models of ADCs at chip level verification. To do so a new algorithm is added to the algorithm library which contains a C wrapper and the actual implementation of the algorithm is in MATLAB. Driver algorithm and data structure classes (two application specific components, see Fig. 4.3) are defined and the created UVC is able to drive the ADC interface using TLM.
The shown voltage sequence in Fig. 5.2 is generated using three parameters. The shown parameters are defined in data structure class. Upon a request from driver, these parameters can be randomized or directly filled. These parameters are what we need to apply the transaction concept and generate constraint random stimulus. This means

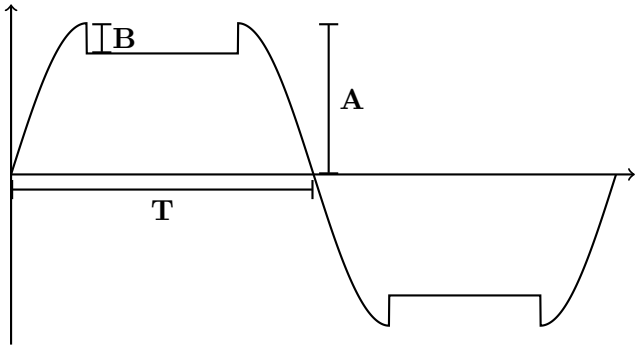

Figure 5.2: Voltage sequence of one phase generated by the motor. T is the half period of the phase, A is the amplitude and a is the overshoot.

Figure 5.3: Generated input stimulus to the ADC model in wave form window of Questasim simulator

that by changing the shown parameters different behaviour of the motor can be modeled and ADC reaction to that can be inspected. This method offers a great acceleration in verifying RNMs of analog blocks in this verification project. Algorithm developement in MATLAB environment is rapidly obtainable and can be also used in selfchecking components. The generated sequence in Questasim simulator is shown in Fig.5.3.

Chapter 6

Conclusion and Outlook

In this project a novel verification technique is introduced in order to improve the verification procedure of mixed-signal SoCs. The proposed method undertakes the real value stimulus generation using transaction level modeling for the designs with analog interface to the outside world. The verification environment is built taking advantage of UVM standard which highly automates digital verification. The resulting testbench provides the possibilty of real value stimulus randomization in transaction level.
Through defining parameters in order to describe an anlog wave a transaction can be generated. The generated transaction in an UVM based environment facilitates the verification procedure. The developed library in this work supports transactions with potentially different types of parameters. The library is application independent and easily extendible. This means that new application specific components can be inserted to the generated environment to form a new analog wave.
This introduced technique together with under-developed techniques for monitoring, checking and coverage collection enables the user to stay within the digital simulation environment and verify a mixed-signal design.

In order to develope the existing verification techniques study of many new proposed featurs in SystemVerilog 2012 standard is the focus of this thesis in future work. New language capabalities like *interface classes* can impressively affect our verification structure. Taking advantage of multiple inheritance offered by *interface classes* the goal is to simplify the implementation of proposed technique and implement more robust verification environment [28].

C Engine Routines

Function	Purpose
engOpen	Start up MATLAB engine
engClose	Shut down MATLAB engine
engGetVariable	Get a MATLAB array from the engine
engPutVariable	Send a MATLAB array to the engine
engEvalString	Execute a MATLAB command
engOutputBuffer	Create a buffer to store MATLAB text output
engOpenSingleUse	Start a MATLAB engine session for single, nonshared use
engGetVisible	Determine visibility of MATLAB engine session
engSetVisible	Show or hide MATLAB engine session

Figure 1: MATLAB engine controlling routines [29]

Bibliography

[1] A. Molina and O. Cadenas, "Functional verification: approaches and challenges," *Latin American applied research*, vol. 37, no. 1, pp. 65–69, 2007.

[2] A. Rath, V. Esen, and W. Ecker, "Analog transaction level modeling," in *High Level Design Validation and Test Workshop (HLDVT), 2011 IEEE International*. IEEE, 2011, pp. 82–82.

[3] ANSI/ASQC A3, *Quality systems terminology*. American Society for Quality Control, 1978.

[4] A. Piziali, *Functional verification coverage measurement and analysis*. Springer, 2004.

[5] T. Ören, "The many facets of simulation through a collection of about 100 definitions," *SCS M&S Magazine*, vol. 2, no. 2, pp. 82–92, 2011.

[6] Lund university, Digital ASIC Group , "Digital asic design, a tutorial on the design flow," 2005.

[7] Cadence, "Using coverage, guiding verification to efficient completion," 2003.

[8] C. Spear, *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Verlag, 2008.

[9] J. Sordoillet and S. Davey, "Integrated, comprehensive assertion-based coverage," in *EDA Tech Forum*, vol. 3, no. 1, 2006, pp. 22–25.

[10] I. NIŢĂ and A. RAPAN, "Improving verification methodologies in digital circuits modeling."

[11] B. Bhattacharya, J. Decker, G. Hall, N. Heaton, Y. Kashai, N. Khan, Z. Kirshenbaum, and E. Shneydor, *Advanced Verification Topics*. lulu.com, 2012.

[12] "Universal Verification Methodology (UVM) 1.1 Users Guide," May 2011. [Online]. Available: www.uvmworld.org

[13] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2003, pp. 19–24.

[14] "Uvm transaction-level visual debugging." Aldec, 2011. [Online]. Available: www.aldec.com

[15] "Uvm adapter class." Doulos, 2012. [Online]. Available: www.doulos.com

[16] "Uvm 1.1 class reference final 06062011." Accellera, 2011. [Online]. Available: www.accellera.org

[17] L. Nagel, O. Enterprises, and N. Randolph, "Is it time for spice4?" in *2004 Numerical Aspects of Device and Circuit Modeling Workshop, Santa Fe, New Mexico*, 2004, pp. 23–25.

[18] K. Kundert, "Simulation of analog and mixed-signal circuits." Cadence.

[19] A. Eisawy, "Ams design configuration schemes, design topologies." Mentor Graphics. [Online]. Available: www.verificationacademy.com

[20] K. Karnane, G. Curtis, and R. Goering, "Solutions for mixed-signal soc verification," 2012.

[21] N. Khan and Y. Kashai, "From spec to verification closure: a case study of applying uvm-ms for first pass success to a complex mixed-signal soc design."

[22] K. Karnane, G. Curtis, and R. Goering, "Solutions for mixed-signal soc verification, as old methods fall short, new techniques make advanced soc verification possible," 2012.

[23] W. Hartong and S. Cranston, "Real valued modeling for mixed signal simulation, ies 8.2," 2009.

[24] N. Khan, Y. Kashai, and H. Fang, "Metric driven verification of mixed-signal designs," *Proceedings of DVCon*, 2011.

[25] M. Frigo and S. Johnson, "The fastest fourier transform in the west," www.fftw.org, 1997.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co, 1995.

[27] "Systemverilog 3.1a, language reference manual," 2004.

[28] S. Sutherland and T. Fitzpatrick, "Keeping up with chip the proposed systemverilog 2012 standard makes verifying ever-increasing design complexity more efficient," 2012.

[29] "Using matlab engine." [Online]. Available: www.mathworks.com