NTNU
Norwegian University of
Science and Technology

Efficient Verification with Portable Stimulus

Karianne Krokan Kragseth

Master of Science in Electronics
Submission date: June 2018
Supervisor: Kjetil Svarstad, IES
Co-supervisor: Isael Dias, Nordic Semiconductor

Norwegian University of Science and Technology
Department of Electronic Systems

Abstract

Portable Stimulus is an upcoming technique for increasing productivity and quality of verification of digital designs. A single test description shall be used to generate tests between multiple abstraction levels and platforms. Questa inFact is a tool that supports a subset of an upcoming standard from Accellera.

The thesis consisted of analyzing the current status and future development of Portable Stimulus. This included conducting a proof of concept through inFact. Generated code from a single description was proven to be used in simulation at IP-, sub-system- and SoC-level in UVM testbenches, as well as with C code running on a CPU. Only the stimulus was portable, meaning frameworks for driving the stimulus had to be created.

The current implementation of inFact is limited when compared to the Portable Stimulus specification, as a lot is not yet supported. It was possible to reuse the same test description, and through coverage driven simulation becomes verification more productive and the quality is increased. This is because of the shortened simulation time and the focus on what to test, rather than how. Thus, Portable Stimulus is heading in the right direction, but it still has a long way to go.

Sammendrag

Portable Stimulus er en kommende teknikk for å øke produktiviteten og kvaliteten på verifikasjon av digitale design. En eneste testbeskrivelse skal brukes til å generere tester mellom flere abstraksjonsnivåer og plattformer. Questa inFact er et verktøy som støtter deler av en kommende standard fra Accellera.

Prosjektet besto av å analysere nåværende status og fremtidig utvikling av Portable Stimulus. Dette inkluderte et gjennomførbarhetsbevis (proof of concept) ved å benytte inFact. Kode generert fra en eneste beskrivelse har blitt bevist å kunne brukes ved simulering av IP-, subsystem- og SoC-nivå i UVM testbenker, i tillegg til ved kjøring av C kode i en CPU. Kun stimulansen er transporterbar, som betyr at rammeverk for å operere stimulansen også måtte skapes.

Den nåværende implementasjonen av inFact er begrenset sammenlignet med Portable Stimulus spesifikasjonen, da mye enda ikke er støttet. Det var mulig å gjenbruke den samme testbeskrivelsen, og gjennom ”coverage” drevet simulering blir verifikasjonen mer produktivt og kvaliteten økes. Dette er på grunn av senket simuleringstiden og fokuset på hva som skal testes, fremfor hvordan. Dermed er Portable Stimulus på vei i riktig retning, men har fremdeles en lang vei å gå.

Acknowledgements

I would like to thank my supervisors Isael Diaz and Vinodh Ravinath at Nordic Semiconductor and Kjetil Svarstad at NTNU for their guidance through weekly meetings. Their advice and feedback have been valuable. I would also like to thank Nordic Semiconductor by providing me with a place to work as well as tools and simulation frameworks required for conducting this thesis.

I also extend my gratitude to Staffan Berg at Mentor for demonstrating and teaching their tool Questa inFact, in addition to fast response when answering questions about inFact.

Table of Contents

Abbreviations

API	=	Application Programming Interface
DMA	=	Direct Memory Access
DSL	=	Domain-Specific language
EDA	=	Electronic Design Automation
HSI	=	Hardware/Software Interface
IDE	=	Interactive Development Environment
LSB	=	Least Significant Bit
PI	=	Procedural Interface
PPI	=	Programmable Peripheral Interface
PSS	=	Portable Stimulus language Specification
QoS	=	Quality of Service
RAM	=	Random Access Memory
RTL	=	Register Transfer Level
RX	=	Receive
SDV	=	Software Driven Verification
SLN	=	System Level Notation
SoC	=	System on Chip
SUT	=	System Under Test
TLM	=	Transaction-Level Modeling
TX	=	Transmit
UART	=	Universal Asynchronous Reciver/Transmitter
UVM	=	Universal Verification Methodology
VIP	=	Verification IP
WG	=	Working Group

CHAPTER 1

Introduction

Development of techniques, such as constrained randomization, functional coverage and the UVM, for block level, have provided improvements in the productivity and quality of design verification. However, at the sub-system and SoC level's verification, challenges continue to grow. The complexity of SoC projects are increasing, making this even more challenging. Also, a lot of effort is put into writing the same kind of tests at different abstraction levels, IP-, sub-system and SoC-level, and at multiple platforms, simulation, emulation and silicon.

As a result, a new approach "Portable Stimulus" is being addressed. The goal is to have a single description of test intent that can generate stimulus for IP-, sub-system- and SoC-level. Portable Stimulus wants to increase verification productivity, and make reuse between abstraction levels as well as between different projects easier. Also, Portable Stimulus wants to increase the quality of design by making bug detection easier and more efficient.

During this thesis, a Portable Stimulus standard was being developed by Accellera, but an early adopter release was present. The first version is supposed to be released during the summer of 2018. A tool by Mentor, called "Questa inFact", is a contributor to this standard, and a subset of Portable Stimulus was supported. This thesis is about analyzing Portable stimulus, and conduct a proof of concept through inFact. It contains an explanation on how Portable Stimulus is achieved at the different abstraction levels, and between different platforms. This includes RTL simulation with UVM testbenches and C code running on an CPU. UVM is used because Portable Stimulus is specified to generated UVM sequences at RTL level. Nordic Semiconductor, an integrated circuits provider, has requested this thesis.

1.1 Problem Description

The problem description states to do a investigation of existing proposal for universal stimulus, and compare them. However, only one proposal has been examined in this thesis, Portable Stimulus, as this is in the process of becoming a standard. Thus, most effort will be put towards this proposal, making it unnecessary to investigate other proposals. In addition, this standard is a collection on multiple proposals. A proof of concept, with a limited implementation of

Portable Stimulus, has been conducted as the problem description states. The methodology followed in this thesis is evaluation of a suggested standard. This through analyzing what it is good for and use it in a realistic example which, hopefully, exposes both strong and weak sides of language/notation/tool.

The work in this thesis include:

- Investigating Portable Stimulus, and supporting tools
- Proof of concept:
  - Choosing a scenario: receive and transmit with UART
  - Creating UVM frameworks, from scratch, for three different abstraction layers (IP, sub-system and SoC)
  - Creating a C code framework
  - Creating Portable Stimulus code by importing UVM transactions and sequences in inFact
  - Creating Portable Stimulus code from scratch in inFact
  - Integrating generated code from inFact into UVM frameworks and running it
  - Integrating generated code from inFact into C code framework and running it

1.2 Layout of the report

Chapter 2 contains an explanation of UVM and Portable Stimulus, and what the idea behind Portable stimulus is and how it should work. Chapter 3 presents the tool, Questa inFact. The process on writing the UVM and the C code frameworks are explained in Chapter 4. The UVM frameworks consist of a separate testbench at IP-, sub-system and SoC-level. How the PSS code is created, both through an import function and from scratch, are also presented. Last, the frameworks and the PSS generated code are connected and simulated. Chapter 5 further discusses and analyses this.

CHAPTER 2

Theory and Background

2.1 Universal Verification Methodology (UVM)

Universal Verification Methodology (UVM) is a result from the UVM working group formed by Accellera, which involves standardization of verification[1]. UVM is a methodology for functional verification improving verification complexity and interoperability between verification components and environments[2]. UVM is also designed for reusability.

2.1.1 UVM Testbench

Figure 2.1 shows a typical UVM based testbench architecture. This include components with defined intent, such as generating stimulus, driving stimulus to the DUT, monitoring the outputs of the DUT, and comparing the inputs and outputs. Testbench top in this example is *top*, instantiating UVM framework[3].

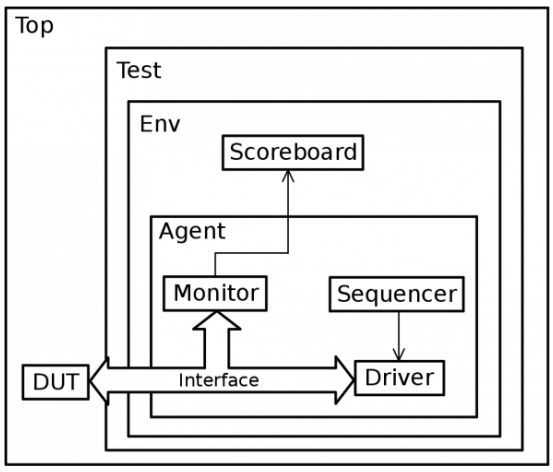

Figure 2.1: Typical UVM testbench [4].

Transactions

UVM components are transaction based through transaction-level communication interfaces and channels[3]. A transaction in UVM is a class object that includes whatever information needed to model a unit of communication between two components. These objects usually include variables and constraints. They contain a model of inputs and outputs that are used for generating stimulus to the DUT and monitoring the DUT. A call to a transaction will randomize the input values, which can be limited through constraints. A transaction is defined by declaring a class that extends *UVM_sequence_item*.

UVM Test

The top-level UVM component is the UVM *Test*[3]. Here is the environment instantiated, configured (through factory overrides or the configuration database), and stimulus is applied through the environment to the DUT by invoking UVM *Sequences*. A UVM *Test* is defined by declaring a class that extends *UVM_test*.

UVM Environment

The UVM *Environment* is a hierarchical component that groups together the UVM *Scoreboard* and UVM *Agent*, and sometimes other UVM *Environments*[3]. The top-level UVM *Environment* encapsulates all the verification components targeting the DUT. A UVM *Environment* is defined by declaring a class that extends *UVM_environment*.

UVM Scoreboard

The UVM *Scoreboard* is used for checking the behavior of the DUT[3]. It receives transactions from the UVM *Monitor* analysis ports. The inputs of the transactions are run through a reference model, called *predictor*, which produces expected outputs, and compares this with the outputs in the received transactions. A UVM *Scoreboard* is defined by declaring a class that extends *UVM_scoreboard*.

UVM Agent

The UVM *Agent* is also a hierarchical component that groups together the verification components that interact with the DUT interface[3]. These components typically include a UVM *Sequencer* managing stimulus flow, a UVM *Driver* applying stimulus to DUT, and a UVM *Monitor* monitoring the DUT interface. Examples of other components that can be included are coverage collectors, protocol checkers and a Transaction-Level Modeling (TLM) model. There are two different operation modes for a UVM *Agent*, active mode where it can generate stimulus, and passive mode where it can only monitor the interface and not control it. A UVM *Agent* is defined by declaring a class that extends *UVM_agent*.

UVM Sequencer

The UVM *Sequencer* works as an arbiter for controlling transaction flow of UVM *Sequence Items* transactions generated by one or more UVM *Sequence*. A UVM *Sequencer* is defined by declaring a class that extends *UVM_sequencer*.

UVM Sequence

A UVM *Sequence* is used for generating stimulus, and is not a part of the component hierarchy[3]. They can be transient or persistent, and come into existence for a single transaction, drive stimulus for the whole stimulation, or anywhere in-between. It is possible to create a hierarchy of UVM *Sequences*, starting with one sequence called a *parent sequence*, invoking one or more sequences called *child sequences*. All UVM *Sequences* needs to be bound to a UVM *Sequencer* to operate. Multiple UVM *Sequences* can be bound to a single UVM *Sequencer*. A UVM *Sequence* is defined by declaring a class that extends *UVM_sequence*.

UVM Driver

The UVM *Driver* applies stimulus to the DUT interface based on received UVM *Sequence Item* transactions from the UVM *Sequencer*[3]. I.e. it converts transaction-level stimulus into pin-level stimulus. Transactions are received through a TLM port. A UVM *Driver* is defined by declaring a class that extends *UVM_driver*.

UVM Monitor

The UVM *Monitor* monitors the DUT interface, and collects the inputs and outputs into transactions that is set to rest of the UVM *Testbench* for further analysis, for example a scoreboard[3]. A typical monitor would covert a certain interface pin wiggles to transactions. The transactions are sent through a TLM analysis port. A UVM *Monitor* is defined by declaring a class that extends *UVM_monitor*.

2.2 What is Portable Stimulus, and Why?

The complexity of SoC projects are increasing and verification productivity is not scaling with it[5]. A significant challenge is test intent reuse across vertical platforms - different phases of RTL design such as IPs, sub-systems, integration, chip and so on. To overcome reuse potential, an initiative is Portable Stimulus (PS), which try to address mainstream and methodical automation of test content reuse. The concept of PS is to only describe test intent and behavior once, and then target it for multiple abstraction layers, such as IP, sub-system and SoC[6]. This includes the verification platforms simulation, emulation and silicon. These platforms have separate requirements and use different languages, as well as different approaches when testing aspects of the system[7].

There are multiple types of engineers across the platforms, such as architects, verification and software engineers[8]. Therefore, it is not easy to reuse tests across platforms nor different engineers with today's approaches and technology. In addition, it is difficult to work on a project and easily reuse this on the next project. It is challenging for one person to completely understand a complex system and test intent infrastructure, and with PS it will be easier to specify complex tests based on design intent, and let the tools take care of the implementation that is difficult to do manually. The purpose of PS is increasing productivity and quality of designers. However, it's not meant to force everyone to use the same level of abstraction, and it will not replace all testing activities.

2.2.1 Verification on Different Platforms

The roles in verification have differences in what they care about and points of view[5]. The architects typically care about throughput, latency and QoS. They want to write tests from the system point of view. The IP designers care about micro-architecture, performance and functionality. They want to write tests that are IP centric. System designers care about correct connectivity, system robustness and use cases. They also want to write tests from system point of view. Evaluation engineers cares about silicon performance and these engineers (ideally) do not want to write tests at all. Today's languages are not expressive enough, and runtime frameworks are not portable.

With RTL, the abstraction level is raised from gate-level through synthesis tools[5]. The abstraction level can be raised further to transactions by using UVM. On top of this are scenarios, and this is where PS comes in. PS wants to solve the reuse problem between platforms by providing a mechanism for specifying intent, in an abstract way, that can be mapped into the different platforms[7]. The abstract approach to creating tests will move the problem solving from think about how it will be done to what is the intent. This will make it into a single specification for what should be tested, making the differences in platforms irrelevant because of the raised abstraction. The specification should be specific enough for a PS tool to be able to analyze it and create the actual implementation for whichever desired platform. Figure 2.2 is a visual representation of the goal of PS as described.

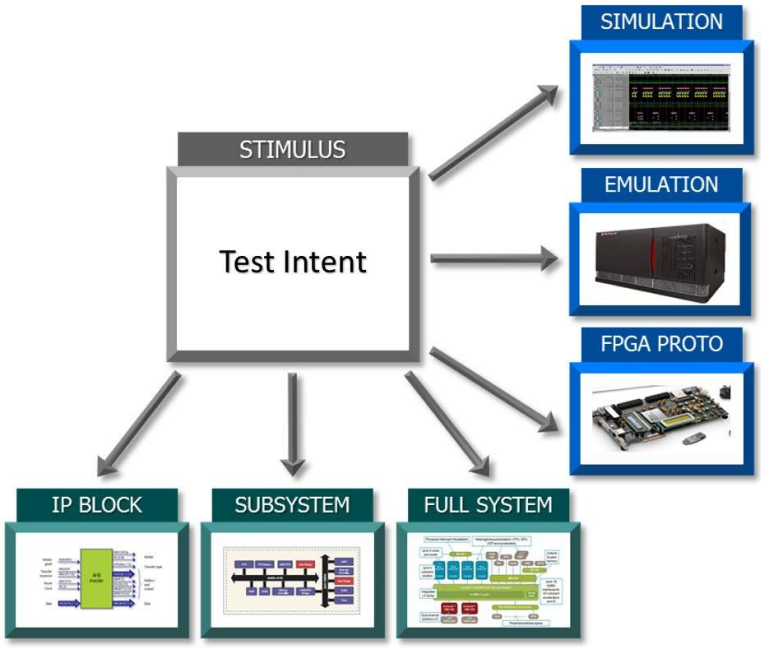

Figure 2.2: The goal of PS, specify test intent once and retarget it to multiple platforms [9].

2.2.2 What One Wants to Test

PS is more about what it means to test and what is intended to be tested rather than how to do it[7]. The PS model will be developed from system specification, containing a description of supported behavior of a device and its requirements. This means it is necessary to understand the operations an element can support and what the requirements are for these operations. When this is clear, individual scenarios can be specified to what one wants to test. For example, verifying that one can receive data and store it into memory. This represents the meaning of test intent, a high-level specification of what one wants, where one can add more details gradually.

2.2.3 UVM vs. Portable Stimulus

Reusable tests are a large part of PS, and it is also a main idea behind UVM[10]. UVM is advantageous as it provides a common language/framework for verification engineers[5]. However, SystemVerilog and UVM are a challenge outside conventional hardware verification environments because non-verification engineers are not familiar with them. In addition, it is overly complicated and hard to debug, and one needs to be an expert to create a simple directed test. Even though UVM is excellent for block/IP level verification, it does not scale to system level verification. This is because verification at block level is typical in in SystemVerilog, while system level would typically require C code. As a result, it is difficult to reuse the stimulus at system level.

2.3 Portable Stimulus Specification Working Group

Accellera Portable Stimulus Specification Working Group (PSWG) is creating a standard for PS[11]. Members are leading semiconductor companies such as Intel and also EDA vendors such as Mentor, Cadence and Breker. The goal of this group is to ”create a standard for verification stimulus captured in such a manner that enables stimulus generation automation, and the same specification to be reused in multiple verification languages and contexts”.

In block and sub-system verification are SystemVerilog mostly used, while SoC and system level uses embedded software[11]. Mechanisms for reuse of code and described behavior is important, and will be provided for these and other languages[6]. However, it should be noted that PS is not meant to be a replacement for existing languages, such as SystemVerilog and C/C++.

The working group was created by initiative of Mentor, and the work started in 2014[7]. They started by identifying requirements and completing a feasibility study. From gathering of requirements and looking at existing technology, they found that Breker, Cadence and Mentor had solutions in this area. Therefore, it was reasoned, given the requirements and existing tools, PS would be feasible solution with additional input. In addition, by having Intel (a SoC provider) as a user, it would be possible to generate a standard. After this, the actual working group was formed.

2.4 Portable Stimulus Specification

On the 15th of June 2017 the PSWG released an early adopter specification for its Portable Stimulus Specification (PSS), followed by version 2 on 28th of February 2018 [12]. This includes a comprehensive explanation for a standard of PS consisting of two languages, a Domain-Specific language (DSL) and a C++ class library. These two languages support PS by making it possible to express a single intent, and then assemble them into test suites. The test suites can be used by many engineers, from hardware designers through to embedded software engineers, and deployed in many different environments across different levels of integration[10].

The specification came after preliminary work of the submitted proposals[7]. The result was a combination of the proposal from Cadence and Mentor. Both had their own declarative language solutions that were merged together to create the DSL. Breker had a similar solution to Mentor, but with C++. The DSL was used to define the semantics, and the same semantics were created for C++. This means Accellera is responsible for defining the standard, but the technology comes from vendors.

2.4.1 Fundamentals

Raising the level of abstraction is one of the fundamentals of Portable Stimulus, in addition to enabling automation of tests for complex scenarios emerging in sub-system and SoC level[6]. The fundamental of the PSWG development is built upon constrained-based and transaction-level verification, which is already widely adopted within industry. As a result, their PSS supports random and non-random data fields and structures that are familiar to SystemVerilog. From object-oriented programming they support familiar inheritance patterns. Figure 2.3 shows a representation of the anatomy of PSS, how one goes from a PS description to executable code.

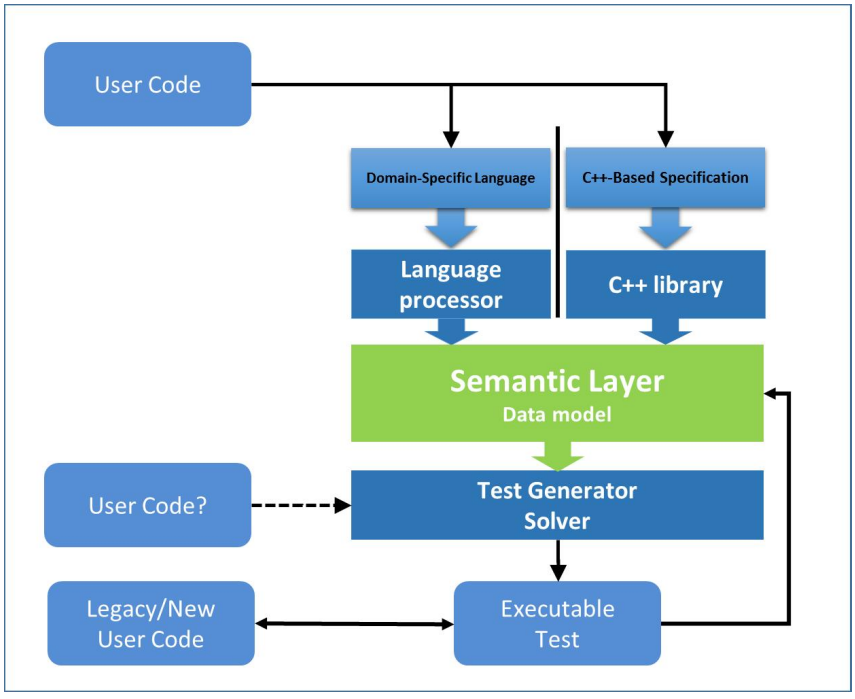

Figure 2.3: The anatomy of PSS[13].

How the engineers will use PSS will vary[7]. The architects figure out what the elements of the design should do and how they relate, in addition to elements like performance characteristics. These can be layered into PSS, especially the performance, once one knows the operations the design should be able to perform.

Verification engineers verify the block level and sub-system level on RTL implementations[7]. They specify test intent that will interact with the RTL. PSS will generate UVM sequences, which shall be used as the test stimulus. A UVM environment is needed for how the traffic interacts with the design, through agents, drivers and monitors. It will not be possible to synthesize a complete testbench from DSL. One will still need the infrastructure that currently is required for whatever platform, but one gets to reuse the test intent. Thus, the RTL verification engineers can reuse the operations the architect has specified, and map it into the UVM environment. This also means that verification engineers need to think on a higher level of abstraction. An analogy to using PSS for creating tests is when one synthesizes the RTL. Only a high-level code needs to be written, while the tools actually does the gate-level implementation. Thus, one thinks in a more abstract way, and the tools automates the implementation. Similar things are happening with PSS, instead of writing pieces for each type of target platform, the tool automates this.

A software engineer, at system level, will generate C code that is going to run on a processor on an emulator or FPGA prototype[7]. They still have to write the driver, but they can use PSS to create scenarios to test the functions of the driver. Today's manual implementation of system level test are of high complexity, which makes them expensive, incomplete and error-prone[14].

Infrastructure pieces still needs to be made at each level, but the intent is reusable[7]. Whenever one writes a specification of intent, multiple scenarios can be created. The output can be generated in multiple ways in the system, which the tool will pick from when generating data. This allows for reuse by generating multiple tests in the infrastructure one has, in addition to reuse of that intent for scenarios on other platforms. A lot of pieces of existing testbenches can be leveraged in PSS, such as UVM coverage components, scoreboards, and so on. With PSS one can say that the thinking is revolutionary, it is something new, but the implementation is evolutionary because what one already knows is not thrown away.

2.4.2 Language

PSS is a declarative language designed for abstract behavioral description[15]. It is intended for modeling scenario spaces of systems, generating test stimulus, and analyzing test runs. This is by creating use cases, including data and control flows. These use cases are used to produce a wide range of possible and legal scenarios for multiple execution platforms by capturing the intent of the test.

A PSS model, a representation of a view of a system's behavior, is made up of elements of behavior called **actions**, and passive entities called objects[16]. The objects are used by **actions**. **Activities** specify the behavior of the **actions**. **Actions** and objects can form reusable model pieces when encapsulated in components, which also can be encapsulated in a package for additional reuse and customization. An instantiation of a PSS model, called scenarios, consists

of a set **actions** and data object instances, scheduling constraints and rules defining relationships between them. By describing the inputs and outputs of an **action**, what resources in the system they require and how they relate to each other, one get a thorough understanding of the capabilities of the system[7].

The language is specifiable for all engineers, in the range from verification of IP block with transactional UVM testbenches to preoperative tests on the silicon, because of the two different input formats[7]. These two languages are equivalent in semantics and are completely interchangeable. DSL is more like SystemVerilog, as engineers in the earlier phases are familiar with, while the C++ syntax are for those in the later phases. An example of how DSL and SystemVerilog is alike is that the transaction-level of PSS overlaps with the SystemVerilog constraint subset so that SystemVerilog constraints can be converted and reused to some extent in PSS[6]. DSL is its own declarative language as it allows users to specify all test intent and how they relate, and then have the tool analyze it to figure out what is necessary for a given platform.

The C++ part consists of a limited library used to create the same model as the DSL specification does. Then, that model is used to generate the target implementation. The reason for having the C++ syntax is to allow engineers already familiar with it to learn PSS faster without having to learn a new language. The C++ library is directly mapped to the DSL constructs. It is more or less a one-to-one mapping, i.e. a line in DSL has a line in the C++ library that is equivalent. Because they are equivalent, only different in input format, one can combine them into a single model. For example, if two engineers have written two pieces representing the system, where one is in C++ and the other in DSL, a third engineer can put them together to create a single coherent output for a target platform.

2.5 Portable Test and Stimulus Standard

This sub-section explains the content of the PSS specification, in addition to an example.

2.5.1 Execution Semantic Concepts

A PSS test scenario is identified by a PSS model and an **action** type designated as the root **action**[16]. Execution of the scenario consists of execution of a set of **actions** defined in the model. For **atomic actions**, the execution is decided by the mapped behavior of **exec body**. While execution of **compound actions** are decided by the behavior specified by their activity statements. Execution of **actions** observed in a run correspond to either being explicitly called by **activities** or being implicitly introduced to achieve correct flow according to model rules. Execution order of **actions** are dictated by the **activities**, starting from root **action**.

2.5.2 Components

A mechanism for encapsulating and reusing elements of functionality is **components**[16]. A PSS model is broken down during execution into parts based on roles of the different actors. The mechanism of **components** are aligning with structural elements of the system and execution environments. Examples are hardware engines, software packages, or testbench agents.

Components are structural entities instantiated under other components, and constitute a hierarchy in a tree structure[16]. The tree structure begins with a top/root **component**, called *pss_top*. They have unique identities, corresponding to their hierarchical path, and can contain imported functions and imported class instances. **Components** have no data attributes or constraints of their own. **Components** are type **namespace** for **actions** and struct types defined under them. They are responsible for reusable grouping of **actions**, **pools** (object resources), and configuration parameters[5].

2.5.3 Actions

Actions are modular, which means they are reusable, and they can interact with other **actions**, have inputs and outputs that define dataflow requirements, and can claim system resources[5]. **Actions** are **simple** if they map directly to target implementation, **atomic** if specified via an **exec block**, and **compound** if they contain an **activity** that instantiates and schedules other **actions**. Assembling **simple actions** into multiple **compound actions** are a way of reusing **actions** many times[10].

Actions are a response to limitations in SystemVerilog, in terms of reuse and customization without changing the original code when constrained random generation are mixed with procedural code[6]. Behavior is specified within an **action**, where complex **actions** can contain specification of sequential and parallel execution of sub-actions and repetitions of sub-actions. This allows for a high degree of automation and static analysis. The behavioral description in **actions** can be reused, and contains a flexible mechanism to map to different targets[5]. **Actions** represent functionality, where the first step is to identify target design behavior to be exercised. Then one needs to figure out what data these behaviors require or produce. The next step is to answer where these behaviors are executed, in the DUT or VIP. Lastly, one needs to figure out what system resources are required to accomplish these behaviors.

Actions interact with each other by flow objects, through inputs and outputs, and resource objects, by claiming resources[16]. Scheduling of **actions** relative to each other is done by the use of activity graphs[10]. In order for the tools to create complex legal scenarios automatically from **actions**, rules or constraints can be written[6]. These describe legal scenarios involving dedicated constructs for modeling the resource requirements and data exchanges. Through constraints, test generation tools can create realistic scenarios and use cases. Constraining these **actions** is done through flow objects and resources.

2.5.4 Activities

Behavior of **actions**, if they contain multiple operations, is described using **activities**[16]. One can think of **action** instances as nodes in an activity graph[5]. There can only be one **activity** in an **action**. Because of this and that they are explicitly a part of an **action**, they do not have a separate name. An **activity** specifies the temporal control and/or data flow between nodes. The relationship between them are described via rules. A **symbol**, declared to represent a subset of an **activity** functionality, is used for reuse and simplifying the specification of repetitive behaviors in an **activity**. The **symbol** can be used as a node in the **activity**.

Activities define top-level scenarios, while **compound actions** define high-level intent, and graphs define scheduling of **actions**[5]. Resource and flow objects define additional scheduling

constrains, by for example locking them prevents other **actions** from using them. Stream objects require another **action** to execute in parallel, while buffer objects execute them sequentially. By default, **action** statements in an **activity** arescheduled sequentially[16]. A **parallel** statement is used to schedule them in parallel. A **select** statement is used to schedule them to be executed in whichever order. When using the **repeat** statement in an **activity**, will execution be repeated, either for a specific number of timer or in a while when waiting for an expression to become true. **foreach** is used to iterate across elements in an array when scheduling them. It is also possible to use an **if-else** block when scheduling.

2.5.5 Flow Objects

A flow object is used as input and output of **actions**, which represent data/control flow, or a pre-condition or a post-condition[16]. Flow objects include buffer objects, stream objects and state objects. Buffer objects define sequential data/control flow[5]. They are pre- or post-conditions for **action** execution. They have persistent storage, and can be read after being written. An **action** that inputs a buffer object must be bound to an action that outputs a buffer object of the same type. A buffer object output can be connected to 0:N input **actions**, if they are of the same type. The producing **action** must complete before execution of consuming **action** may begin.

Stream objects are used between **action** during concurrent activity, for example over a bus or network or across interfaces, by representing transient data or control exchanged[16]. They represent data item flow or message/notification exchange. An **action** that inputs a stream object must be bound to an **action** that outputs a stream object of the same type. State objects represent the state of some entity in the execution environment at a given time. State object writes must be sequential, but reads can be concurrent.

2.5.6 Resource Objects

Resource objects represent computational resources available in the execution environment, which are user defined and created in a type struct[16][10]. They can be assigned to **actions** during their execution, allowing them to lock resources from other **actions** or share them. While sharing, no **action** can lock that resource until all **actions** are finished. Both resources and constraints can be inherited from a base **action**.

2.5.7 Pools

The mechanism of pools are representing collections of resources, state variables, and connectivity for data-flow purposes[16]. **Pools** are structural entities instantiated under components. Their task is to determine the accessibility to flow and resource objects for **actions**. Thus, they shape the legal test scenarios when determining possible assignments of objects to **actions**. To achieve this, the object-reference fields of **action** types need to be bound to pools of the respective object type. The associated bind directives in the **component** scope are: Resource references with a specific resource **pool**, state references with a specific state **pool**, and buffer-/stream object references with a specific data-object **pool**. Because they determine possible assignment of objects to **actions**, they shape the legal test scenarios. Flow exchange in a **pool** are transported, one **action** outputs an object while another inputs it. When an **action** locks or shares a resource, it is within a **pool**.

2.5.8 Randomization Specification Constructs

Scenario properties are expressed as algebraic constraints over attributes of scenario entities[16]. These exist for several categories of struct and **action** fields. The constraints shape every aspect of the scenario space, such as determining the legal value space for attribute fields of **actions**. Another examples is that they can affect the legal assignment of resources to **actions** and scheduling of them, and they can restrict binding of input and outputs of **actions**. Constrained randomization is the process of selecting values for scenario variables. The order of execution, as specified in **activities**, decides when the randomized values of variables becomes available.

Two types of constraint blocks for **action**/struct members are supported in PSS: Static constraints that always hold, and dynamic constraints that only hold when they are traversed in the **activity**[16]. Randomization of plain data models associated with scenario elements, and randomization of different relations between scenario elements, such as scheduling, resource allocation, and data flow, are supported in PSS.

2.5.9 Coverage Specification Constructs

PSS have a `coverspec` construct, that is used to specify coverage target[16]. They identify key value ranges and value combinations that must occur to exercise key functionality. Examples of coverage targets are **action**, scenario, datapath, value and resource coverage.

2.5.10 Type Extension

Type extensions are added in PSS for enabling the decomposition of model code, such that reuse, and portability is maximized[16]. Model entities, such as **actions**, objects, **components**, and data types, can have a number of properties or aspects that are logically independent. However, distinct concerns for a specific entity will often need to be developed independently. An example of concern is implementing **actions** and objects for a specific target platform/language. Another concern is with model configuration of generic definitions for a specific DUT that affects **components**/data types declared elsewhere. Defining a functional element of a system that introduce new properties to common objects is also a concern. What is extensible in PSS is composite and enumerated types. Compound type extensions include **struct**, **action** and **component**.

2.5.11 Packages

To group, encapsulate, and identify sets of related definitions, type declarations and type extensions, one can use **packages**[16]. **Packages** are useful for allowing extensions to the same types that are inconsistent with one another to coexist and be managed more easily. Examples are when introducing contradicting constraints or specifying different mappings to the target platform. **Packages** work as a namespace for the types declared in their scope.

2.5.12 Test Realization

An **exec block** is used for mapping of a PSS entry to its foreign-language implementation[16]. A PSS model interacts with external foreign-language code for help to compute stimulus values

or expected results during stimulus generation. Another reason is that code corresponds to the behavior represented by leaf-level actions, for example APIs of the SUT or utility libraries. For this can the procedural interface (PI) be used. An **exec block** is a mechanism for specifying functionality associated with a **component** or **action**.

2.5.13 Hardware/Software Interface

Hardware/Software Interface (HSI) is an abstraction responsible for device management [5][16]. The purpose is to generate hardware dependent software (drivers) from a single specification in different environments and languages. HSI is used to ensure portability of scenarios across environments and devices/SoCs. The HSI specifies device initialization, interrupt management and other operations such as configure, transmit/receive, and registration of device capabilities.

HSI is a set of constructs for capturing the hardware aspects required to implement the abstraction [5][16]. I.e. it captures the programmer's view of a peripheral device in a manner that is unknown to the underlying verification environment and platform. A provided C++ API is used to capture HSI specification, such as software programmable registers, virtual registers, DMA descriptors and interrupt properties. The API captures test intent as the users can use it to specify the programming sequence for different operations that can be performed on a peripheral device.

This enables an implementation, on a given language and verification platform, to be derived for the abstract representation of the HSI[16]. HSI enhances PS by abstracting away the stimulus model from the verification platform specific implementation of HSI, and can be ported to a different verification platform. An example is simulation to emulation. It is also enhanced by basing the HSI specification on a standard interface/API contract for a given device, so that the stimulus model can easily be ported to a different device.

2.5.14 A Simple PSS Example

This PSS example involves a UART receiving/transmitting data packets via a data port, as seen in Figure 2.4[5]. Packets are moved to/from memory (MEM) through a DMA (direct memory access). The command port accesses registers to configure UART and DMA, and reading and writing MEM. The example is simplified and does not include the whole code. The purpose of this example is to give an overview of how the described elements in PSS looks in the code, and not give an exact representation.

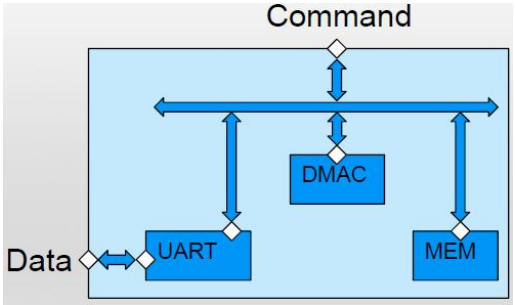

Figure 2.4: Block diagram of a UART together with a DMA controller and memory[5].

In Listing 2.1, two objects are created, a buffer object operating sequentially, and a stream object operating in parallel. These will be used between the UART and DMA. Listing 2.2 and 2.3 demonstrates the components for the UART and DMA when using the data port. The UART component specifies two actions for receiving and transmitting, using the stream object. They also lock the UART resource. The DMA component specifies two actions transferring data between the memory and UART. The data from/to the UART are of the stream type, while the data from/to memory are of the buffer type. They lock the DMA channel when operating.

```
1  stream data_stream_s { rand int size; }
2  buffer data_buff_b   { rand int size; }
```

Listing 2.1: Flow objects: Buffer and stream[5].

```
1  component uart_c {
2    resource uart_r {   };
3    pool [1] uart_r uart_p;
4    bind uart_p {*};
5
6    action read_in_a {
7      output data_stream_s data;
8      lock uart_r myuart;
9      constraint c1 {
10       data.size % 4 == 0;}
11   };
12
13   action write_out_a {
14     input data_stream_s data;
15     lock uart_r myuart;
16   };
17 }
```

Listing 2.2: PSS component for UART[5].

```
1  component dmac_c {
2    pool dma_channel_r chan_p;
3    bind chan_p {*};
4
5    action q2m_xfer_a {
6      input data_stream_s in;
7      output data_buff_b out;
8      lock dma_channel_r chan;
9    }
10
11   action m2q_xfer_a {
12     input data_buff_b in;
13     output data_stream_s out;
14     lock dma_channel_r chan;
15   }
16
17   action m2m_xfer_a {...}
18 }
```

Listing 2.3: PSS component for DMA[5].

The top component is created as seen in Listing 2.4.

```
1  component pss_top {
2    uart_c uart0;
3    dmac_c dma0;
4
5    pool data_stream_s stream_p;
6    bind stream_p {*};
7
8    pool data_buff_b buff_p;
9    bind buff_p {*};
10 }
```

Listing 2.4: Top component instantiating the UART and DMA component[5].

A test is created in Listing 2.5. Here the actions of the UART and DMA components are bound together. The activity states that receiving and writing to memory should happen in parallel, and then that transmitting and reading from memory should happen in parallel. The names are not correct according to previous definitions in this example, and not all objects have been defined.

```
1  action loopback_test {
2    bind rd_i.data q2m.src;
3    bind wr_o.data m2q.dst;
4    bind q2m.dst m2q.src;
5
6    activity {
7      parallel {
8        rd_i;
9        q2m;
10     }
11     parallel {
12       wr_o;
13       m2q;
14     }
15   }
16 }
```

Listing 2.5: PSS test case[5].

2.6 Reuse Opportunities

There are opportunities for reuse between PSS and the user code shown in Figure 2.3 from subsection 2.4.1[13]. This includes, during the solve process, and during the execution. In the solve process there are reuse opportunities when creating expected results and managing procedural heavy processes, for example memory management. In execution can existing procedural code be used to carry out test intent by PSS.

2.6.1 Translating Existing Declarative Descriptions

In SystemVerilog there are declarative descriptions, such as random variables and constraints in its classes[13]. This can be reused in PSS, as PSS constraint constructs are created in a way for easy reuse. The flow of SystemVerilog class reuse for PSS is demonstrated in Figure 2.5. The PSS language is generated by a processing tool reading the SystemVerilog code and identifies classes and/or covergroups, and translates it to the corresponding PSS constructs.

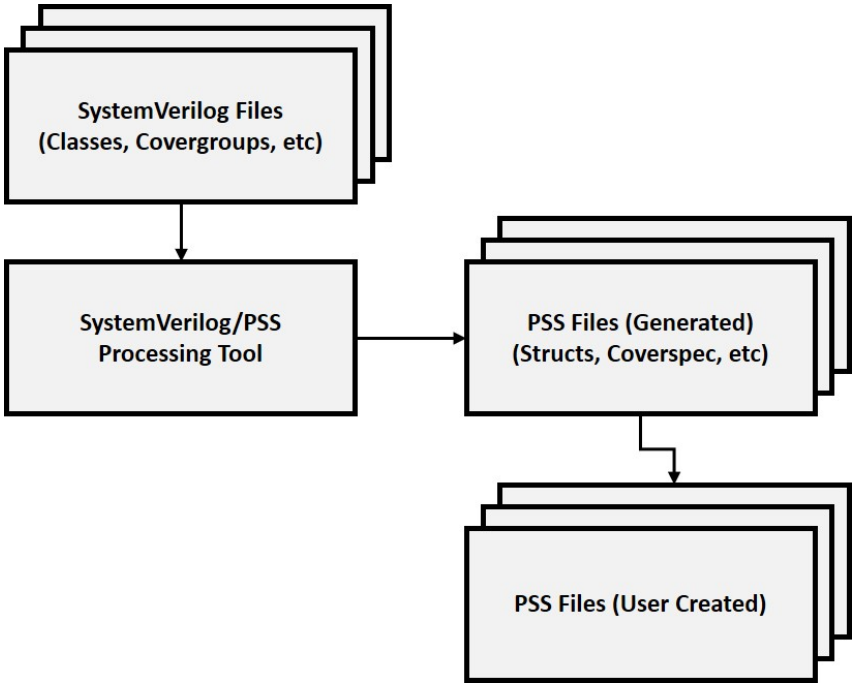

Figure 2.5: Reuse flow for SystemVerilog Declarative Description[13].

Figure 2.6 demonstrates how similar SystemVerilog classes is to PSS, where the only difference is that PSS uses a struct instead[13]. PSS uses struct because the data structures is pure data structures, and does not include methods as in SystemVerilog.

```
typedef enum {
  MODE_1,
  MODE_2,
  MODE_3,
  MODE_4
} mode_t;

class device_cfg;
  rand mode_t     mode;
  rand bit[15:0]  coeff1;
  rand bit[15:0]  coeff2;

  constraint coeff_c {
    coeff1 <= coeff2;
  }

  constraint mode_c {
    if (mode == MODE_1 ||
        mode == MODE_2) {
      coeff1 == 0;
    }
  }
endclass
```

SystemVerilog

```
enum mode_t {
  MODE_1,
  MODE_2,
  MODE_3,
  MODE_4
}

struct device_cfg {
  rand mode_t     mode;
  rand bit[15:0]    coeff1;
  rand bit[15:0]    coeff2;

  constraint coeff_c {
    coeff1 <= coeff2;
  }

  constraint mode_c {
    if (mode == MODE_1 ||
        mode == MODE_2) {
      coeff1 == 0;
    }
  }
}
```

Portable Stimulus Specification

Figure 2.6: SystemVerilog Translation to Portable Stimulus Specification[13].

2.6.2 Existing Test Realization Code

Reuse of test realization code might be the biggest opportunity of reuse for PSS[13]. This is because it is code that carries out test intent. Examples of test realization code are UVM sequences carrying out lower level operations, SystemVerilog utility tasks, and C/C++ driver/stub code for programming an IP. PSS provides a procedural interface enabling external functions to be referred from it for implementation of test behavior.

An example of a configuration of a device using a C utility API is shown in Figure 2.7[13]. How this API can be used in PSS is shown in figure 2.8. Here is the behavior and data for configuring the device encapsulated in the **action**. Function prototypes are specified for the external methods, and for extraction of these methods can PSS tools get them from C/C++ header files. The realization of behavior of the **action** is specified in the **exec block** of type *body*. First, a random mode is selected, and then two calls to *set_coeff*, before calling *init*.

```c
#include <stdint.h>

typedef enum {
  MODE_1,
  MODE_2,
  MODE_3,
  MODE_4
} mode_t;

void set_mode(mode_t m);

void set_coeff(uint8_t coeff_num, uint16_t coeff);

void init(void);
```

Figure 2.7: Device programming API in C code[13].

```
import void set_mode(mode_t m);
import void set_coeff(bit[3:0] coeff_num, bit[15:0] coeff);
import void init();

component device_comp {

  action do_device_cfg {
    rand device_cfg cfg;

    exec body {
      set_mode(cfg.mode);
      set_coeff(1, cfg.coeff1);
      set_coeff(2, cfg.coeff2);
      init();
    }
  }
}
```

Figure 2.8: Calling Test Realization Code in PSS[13].

The same PSS description can be used for the UVM sequence for programming the device, shown in Figure 2.9[13]. Here are the external functions implemented as SystemVerilog tasks, which uses a UVM sequence to program registers within the device. These two examples show that by the use of external test realization code will allow the PSS description be more abstract. Benefits are that the description is more easily portable, reuse of existing code is encouraged, and it helps to avoid bugs.

```
class device_reg_seq extends uvm_sequence;
  // ...
endclass

class device_api_uvm;
  uvm_sequencer #(reg_seq_item)seqr;

  task set_mode(mode_t mode);
    device_reg_seq seq = device_reg_seq::type_id::create("seq");
    seq.item.addr = MODE_REG;
    seq.item.data = mode;
    seq.start(seqr);
  endtask

  task set_coeff(byte unsigned id, shortint unsigned value);
    // ...
  endtask

  task init();
    // ...
  endtask

endclass
```

Figure 2.9: Reusing UVM Sequences[13].

2.6.3 Existing Modeling Code

In addition to test realization code, related to triggering of behavior, modeling of the data when triggering test behavior is also important[13]. Modeling code can also be reused, and a common example is memory allocation. Figure 2.10 shows an example where one wants to control and contain the size of a memory buffer being written. It will be easier to reuse an existing memory allocation implementation in PSS than to create one.

```
import bit[31:0] alloc(bit[31:0] sz);

struct buf_addr {
  rand bit[15:0]    sz;
  bit[31:0]         addr;

  exec post_solve {
    addr = alloc(sz);
  }
}

component top_comp {

  action do_write {
    rand buf_addr   buffer;
    // ...
  }

  action entry {
    do_write    wr1, wr2;

    constraint c {
      wr1.buffer.sz != wr2.buffer.sz;
    }

    graph {
      wr1;
      wr2;
    }
  }

}
```

Figure 2.10: Reusing UVM Sequences[13].

PSS provides a mechanism that enables interaction between existing behavioral code and the solver within the processing code, where the PSS description is evaluated[13]. **Exec blocks** specify the behavior between PSS and external code. Used in a pre-solver will they be evaluated before the solver selects values for random variables. The external code can initialize non-random values, later being used by the solver. Post-solver is also possible, where they are evaluated after the solver has selected specific values for random variables. Then, the external code can compute values for non-random variables that are based on the random ones generated form the solver.

The *alloc* function is external, and can be used by calling it from a pre-solver **exec block**[13]. This way will a random size be selected before calling the *alloc* function, allowing the external code to use the value.

2.7 PSS Tools

At the early adopter release about 80 % of the requirements are met in the PSS specification[14]. Here, the vendors cooperate on standards, but compete on tools. How one goes from having specified a PSS model to getting an executable code is demonstrated in Figure 2.11. The PSS tool, also called the ”secret sauce”, is provided by the vendors[17]. The purpose of the tool is to map the specified intent and behavior of the PSS model into different platforms, and create stimulus, the implementation, for them. APIs are for specifying content of the target platform, for example it has four processors and two DMAs.

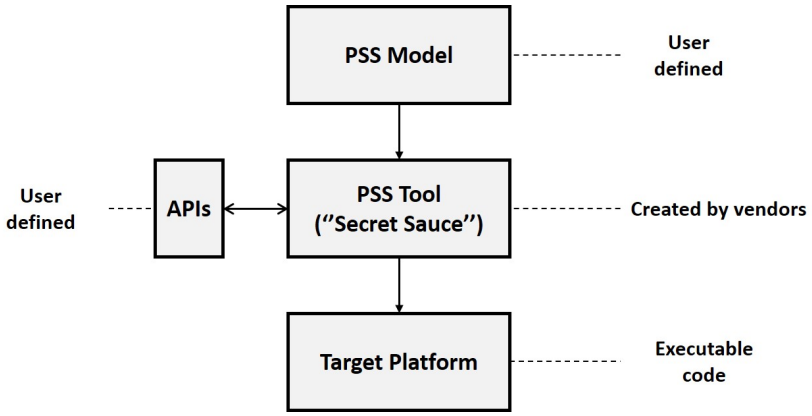

Figure 2.11: From PSS model to executable code.

Mentor has expanded its portable test and stimulus technology across its full Enterprise Verification Platform[18]. This is offered through Questa inFact[19]. This tool supports UVM and standard languages such as SystemVerilog, SystemC, C/C++, and C tests for embedded processors. Mentor does also offer PSS courses and seminars through their verification academy.

Breker have a C++ language representation of PS, and are contributing this to the PSS C++ syntax[20]. This is available in the TrekSoC. As the Portable Stimulus standard moves toward ratification, Breker will continue to implement the domain-specific language (DSL) into TrekSoC portfolio to be fully compliant with the standard.

Cadence has a tool called Perspec System Verifier[21]. This tool was originally created for a self-defined system level notation (SLN) language used to model SoC and SW. After joining the Accellera PSWG, a lot of SLN are donated to the new PSS languages. They are planing to support PSS shortly after the standard is officially ratified, but in the meantime, they support SLN. Perspec is able to generate test in SystemVerilog and C code.

CHAPTER 3

Portable Stimulus Tool Questa inFact

This chapter will cover the chosen PSS tool Questa inFact. Understanding how inFact works is a crucial step before writing PS code. This will be further explained in this chapter.

3.1 Choosing a PSS Tool

Among the three available tools for PS code, as presented in sub-section 2.7, Questa inFact was selected. inFact already supports a lot of the PSS specification, in addition to being continuously developed alongside the specification. The new DSL language is supported by inFact, which is preferred because of the similarities to the already known SystemVerilog. Also, Nordic Semiconductor uses Questa SIM as a simulation tool for their digital designs, making it easier to utilize the result from the created PS code in inFact.

There are some expectations and desired requirements for inFact. The generated codes from the PS code would hopefully cover most of the total verification, allowing PSS to be the main focus instead of a supplement. The UVM framework and a driver for the software are expected to be in place before PSS results can be utilized, but integrating the results shall be easy and require as little effort as possible. In addition, the effort of creating PS code shall be less than the effort of writing all the codes at the different abstraction levels. This is not tool specific, but PSS specific. Thus, this is indifferent of Questa inFact.

3.2 Introduction to Questa inFact

The tool Questa inFact is a graph-based verification tool intended for generating and directing stimulus for a testbench[22]. It is non-intrusive to the testbench, and implemented by instantiating pieces called test components within the testbench. Simulation of inFact is run in conjunction with the simulator, and inFact can replace or be combined with existing stimulus generators. It can control stimulus generation as shown in Figure 3.1. It should be noted that inFact can also control an existing testbench configuration, or a combination of both, but controlling stimulus generation is the focus of this thesis.

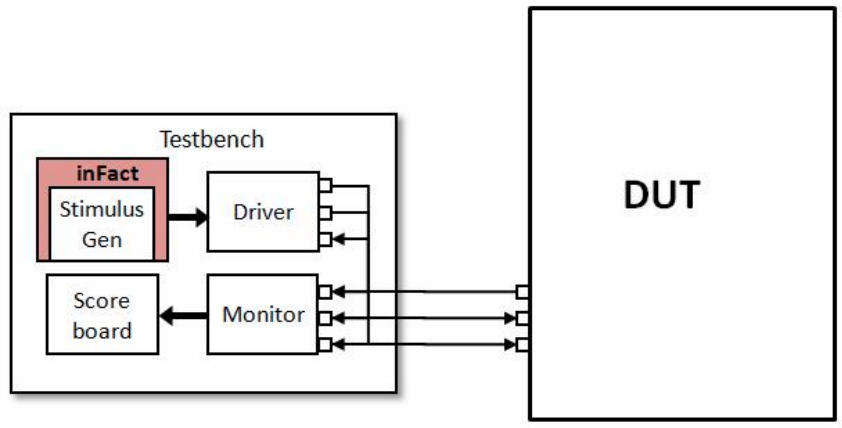

Figure 3.1: Stimulus Generation Control through Questa inFact[22].

An Interactive Development Environment (IDE) is accompanied with inFact[22]. This consists of wizards to set up project files and directory structures, view them and their associated attributes, as well as an editor to create and modify inFact-specific files. Projects can be created through this IDE, but also through the command line. This is possible as a makefile is generated together with the inFact projects, allowing it to be regenerated. The IDE is based on the Eclipse editor.

3.2.1 Rules File and Graph

Defined in the rules file is the stimulus we want to generated[22]. It is used by inFact when translating verification intent for the DUT into a set of hierarchical rules. The specified rules within the rules file reflect DUT behavior in the context of verification scenarios/stimulus sequences. The rules are written in a higher abstraction level than for example SystemVerilog code[23]. However, it does not force a single level of abstraction, as they can be as detailed as desired. SystemVerilog classes can be imported into inFact to automate the creation of a rules file.

From the rules file a graph, involving a compiled binary representation of the rules, is generated[22]. This graph represents the stimulus defined in the rules file in an abstract way, where each action in the rules translates to an action node in the graph. During simulation runtime, this graph is traversed to determine which values to generate.

3.2.2 Action Functions

The action methods (tasks/functions) created in the runtime graph for each graph node performs different operations[22]. They map to a specific task or function in the testbench. This typically consists of processing TLM transactions, generating firmware instructions, interacting with the testbench, and driving a signal. Stimulus is created from these actions during runtime by setting values in a transaction, driving stimulus directly through testbench components or interfaces, and calling existing embedded code.

3.2.3 Coverage Strategy

In inFact, stimulus is driven by a coverage strategy[24]. This makes coverage an active component during simulation that is opposite to coverage in SystemVerilog, where it is passive and only monitors the design. By default, random test sequences from the test component are generated by inFact[22]. The coverage strategy can be used to direct the stimulus to meet a specific coverage target. Upon meeting the target, the simulation is ended. Termination of the simulation can also be set after a number of iterations. This allows the test stimulus to be completely random or directed, or a combination of the two.

3.2.4 Test Components

In order to employ the graph, a methodology-specific container called test component is used[22]. This provides the necessary wrapper to instantiate the graph inside the testbench. This is achieved by a style-specific file facilitating the interface between the testbench and the graph. Test components are configurable and can be enabled/disabled by testbench code, and can be reactive to other graphs and testbench objects (transactions or function/method calls). The test components can be generated in two ways, either by importing a testbench source, or manually from a template. They are language and method specific, meaning UVM, SV class and C code uses a separate test component.

3.2.5 Integration Process

The test component integrates inFact into a testbench[22]. It can be created for an existing testbench or while developing a new. The integration process consists of creating the test component and instantiating it in the testbench. When the rules file, graph, coverage strategy, and test engine are created in the test component, it is ready to be instantiated. It is possible to use multiple test components within a testbench. The test component can be referenced wherever needed.

3.3 Defining a Rules File

From each test component type a rules file template is provided as a starting point[22]. If the test component is created from importing a testbench, rule segments are automatically generated. Inside the rules file are rules written involving what stimulus to create, where constraints can be used to remove illegal values. All rule keywords can be found in the inFact reference manual[25].

The rules are written in a **rule_graph**, which is used when the graph is generated[22]. Inside can rules be directly written or be imported from a **rule_segment**. A **rule_segment** can contain the same rules as in a **rule_graph**. Thus, the benefit of **rule_segments** are reuse in multiple **rule_graphs**.

Actions within a **rule_graph** declare action functions and becomes leaf nodes of the graph[22]. The actions *init* and *infact_checkcov* are standard in the **rule_graph**, where the latter checks progress toward coverage goals[22]. They are run by referencing them in a rule sequence. A rule sequence defines the rules of stimulus activity, i.e. what and when to run stimulus.

In addition are **struct** instances called here. A **struct** declares a hierarchical data structure, combining data, local constraints, and graph structure information into reusable and extendable objects. This can be compared to a class in SystemVerilog[24]. In UVM will a **struct** represent a transaction. Once a **struct** is declared, one or more instances of it can be created.

Inside the **struct** can **meta_actions**, **constraints**, **symbols** and coverage attributes be declared[22]. **Meta_actions** are variables[24]. A value, from the valid domain, will be selected by the inFact algorithms when traversing a **meta_action** in the graph. It is possible to use **meta_action_import** to import a value from the testbench, where the value will be set by the testbench and not by inFact. **Constraints** limit the values of the **meta_actions**, and work in the same way as in SystemVerilog. They can also be applied to the whole **rule_graph** or coverage strategy objects. Dynamic **constraints** can be created to be valid for a certain branch. **Symbols** introduce hierarchy to the rule graph, and forms sub-rules.

To be able to use the **struct** in the testbench an **interface** is needed[22]. This is not the same as an interface in SystemVerilog, but provides synchronization between the graph and the testbench[24]. Once an **interface** is declared for a specific **struct**, it can be run through a call in the rule segment. By default, the **meta_actions** will be traversed in the order they are specified, but this can be branched by referencing the **struct** inside itself. This can be done for example by adding OR or adding if-statements.

Attributes can be used to modify what is being generated[24]. This is useful when moving from different abstraction level, for example when the set of variables are not the same. These rules described in these sub-sections are the most basic[22]. Rules that are present are dependent upon application. Also, additional complex rules exist allowing more customization of the structure and actions of the graph.

3.3.1 Example: Defining Rules

Listing 3.1 demonstrates an example on how rules are defined. It begins with the **rule_graph** declaration, later being used to define the rule sequence at Line 25. *Init* is an action leaf node in this example that initializes the graph interface.

```
1  rule_graph comp_eng {
2      action init;
3
4      struct trans {
5          meta_action addr [unsigned 7:0];
6          meta_action data [unsigned 15:0];
7          meta_action size [1,2,4,8,16];
8          meta_action dir [enum READ, WRITE];
9
10         constraint mem_reg_c {
11         if (addr inside[0x10..0x1F]) {size <=8};
12         if (addr inside[0x40..0x5F]) {size inside[2,4]};
13         }
14
```

```
15          symbol mem_fields = size data;
16
17          trans = ((addr[0..7] dir[READ]) | (addr[8..255] dir))
                mem_fields;
18      }
19
20      trans tr1, tr2;
21
22      interface do_tr(trans);
23      constraint diff_addrs_c {tr1.addr != tr2.addr}
24
25      comp_eng = init repeat {
26          do_tr(tr1)
27          do_tr(tr2)
28      };
29  }
```

Listing 3.1: Rule constructs example[22].

The **struct** *trans* declares the four variables *addr*, *data*, *size* and *dir*. The *size* is constrained based on the *addr* in Line 10. At Line 15 is a **symbol** declared, grouping the **meta_actions** *size* and *data*. The rule at Line 17 declares the top-level rule, where the value of *addr* determines whether *size* and *data* are of read or write type.

The **struct** is declared twice at Line 20, and connected with the graph and testbench with the **interface** at Line 22. A **constraint** makes sure that the two **structs** cannot have the same address. Inside the rule sequence the *init* is called first, followed by a repetition of sending the first and second **struct** to the testbench.

3.4 Viewing the Graph

The IDE provides a visualization of the graph. The graph can either be generated by the IDE or compiled through commands[22]. The display shows the symbols as solid boxes that can be expanded to show their graphical definition in-line or in a separate viewer tab. The **meta_actions** are shown as circles, including the number of possible values. If the **meta_action** is of the **import** type it will be marked with a green arrow. An example is shown in Figure 3.2, representing the graph of the first **struct** from the example in sub-section 3.3.1.

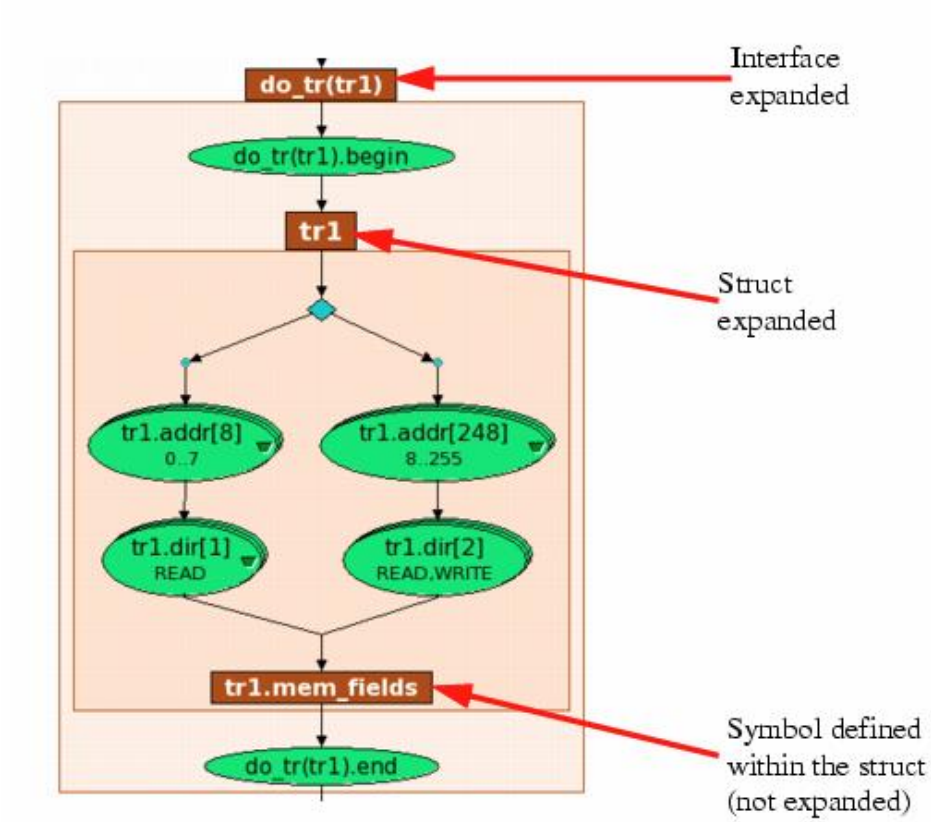

Figure 3.2: Example of graph from Example: Defining Rules in sub-section 3.3.1[22].

3.5 Specifying Coverage

As mentioned, stimulus is driven by coverage in inFact[22]. Except when specifying an explicit set of iterations. There are two types of coverage strategies, action coverage and path coverage. The first targets one or more individual node in the graph. This is done by creating a coverage object for a selected node in the graph. The default is to create a bin for each of the possible values, but this can be overridden by specifying its **bins**. A **bins** is specified by declaring it where the **action/meta_action** is specified, in the same way as in SystemVerilog: *bins* ***bins_target*** *bins_name* ***bins_definition****;*[25]. **Bins** can be imported from an existing testbench as well.

Path coverage target cross-coverage goals, and is created by specifying a start node and an end node[22]. Nodes in between can be defined as ”don’t care”. Path coverage can also be associated with **bins**, usually in a **bin_scheme** that is a collection of multiple **bins**. With both action and path coverage will each bin be hit once, unless else specified. This eliminates redundancy when it comes to simulating the same test case multiple times[24]. It is also possible to see how many tests each coverage goal will generate. This gives an idea about simulation time.

3.6 Simulation of inFact

Simulation can begin after the graph sequence has been integrated into the testbench[24]. The sequence file must be included in a package through a **include**-statement. If the inFact sequence replaces an old sequence, it can be overridden through *uvm_set_type_override*. Else, it can be directly instantiated by name in the testbench. Also, a path to the .ini file created by the inFact workspace is needed as a simulation argument. Next, the compilation and simulation of the testbench can begin in a ”normal” way (through vlog and vsim in Questa SIM).

3.7 inFact versus PSS Specification

The rule language used in inFact forms a basis for the PSS DSL language[23]. Further DSL constructs will be added to inFact during the year, and when the PSS 1.0 is released, inFact will support a true subset of the DSL. The C++ part of PSS will also be supported. Today, some inFact terminology may change as they have the same semantics under a different name in the PSS specification. Examples are **meta_actions** that are the same as flow objects, and **attributes** that are the same as **exec blocks**. **Resources** and **Pools** are not supported by inFact at the moment. This means that the entire system cannot be yet modelled in inFact, as described in the Theory and Background chapter.

CHAPTER 4

Portable Stimulus Verification Infrastructure

A universal asynchronous receiver/transmitter (UART) was chosen for creating scenarios in PSS. The reason for choosing the `UART` is because it is possible to create complex scenarios by using direct memory access (DMA), as well as simpler scenarios. It was desired to test all the levels, including IP- and sub-system level at simulation, and the whole SoC, with both simulation and emulation. This chapter presents how this is achieved through creating a UVM framework, a framework for running C code, creating PS code, and finally using the generated results from the PS code in the frameworks. All code is located online in a digital appendix, see Appendix A for information. The DUTs and compilation and simulation frameworks used in this thesis are from existing Nordic Semiconductor material.

4.1 UART Description

The `UART` is used to receive (RX) and transmit (TX) a byte of information. The frame format consists of a start bit, 8 random bits (payload), an optional parity bit, and one or two stop bits. The LSB of the payload is received/transmitted first. This is configured in a register. Reading and writing of the registers are done through a `PAR` interface, that consist of the signals: Address, Write data, Read data, Write enable, and Read enable.

Figure 4.1 represents a waveform of the `UART` receiving a frame. This is done by changing the serial input *rxd* from '1' (idle state) to '0', creating the start bit. The following 8 bits are stored in a register *rxData*, that can be read by the external environment. A '1' signals the stop bit. Data on *rxData* is ready when the *rxDataReady* signal is set, and the external environment signals that the data is read by sending a pulse on *rxDataReq*.

The `UART` transmits data on the serial output *txd*, as shown in the waveform in Figure 4.2. First, the payload needs to be written to *txData*. This is done by an external environment that sets the *txDataReady* signal when the `UART` can read the payload. The `UART` signals that the data is read by sending a pulse on *txDataRead*. After this,the `UART` will transmit a frame containing the written payload.

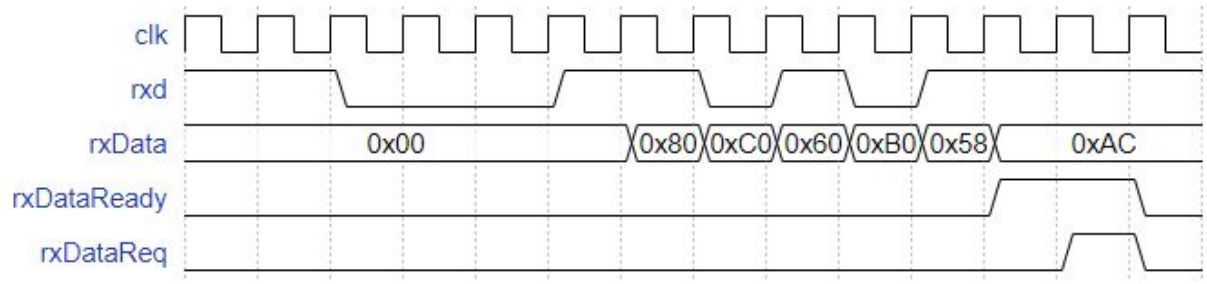

Figure 4.1: Waveform of receiving a payload to `UART`.

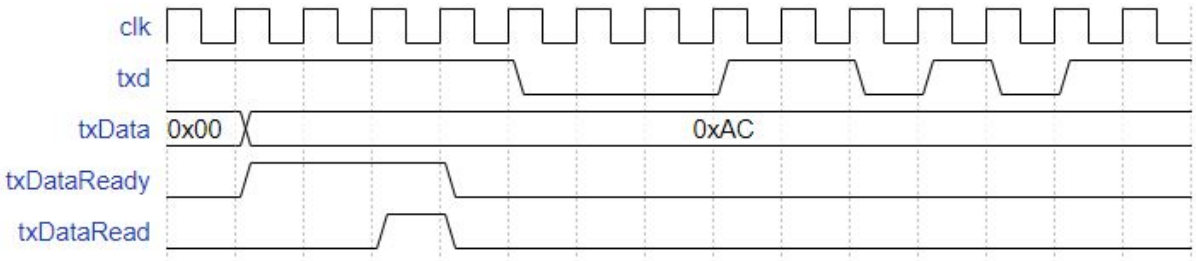

Figure 4.2: Waveform of transmitting a payload from `UART`.

4.2 UVM Framework

The first step is to have UVM framework in place. This is because the generated code in SystemVerilog from PS code is UVM sequences. Thus, the other parts of a UVM framework are essential for stimulus execution. Nordic Semiconductor does not use UVM for their verification. Therefore, UVM frameworks had to be created from scratch, in addition to learing UVM. First, a UVM framework was created at IP level. In addition, to test the framework, a set of simple sequences were created.

At the sub-system level, the UVM monitor can be reused for monitoring the `UART` inside the sub-system. The same transaction, sequences and sequencer were reused. However, the rest of the UVM framework had to be created, as there are different stimulus signals driven at IP level and sub-system level. The same sequences were used to test it.

The UVM framework at SoC level (top-level) could copy/reuse most parts of the previous UVM frameworks. The transaction, sequences and sequencer were reused, and the environment and agent were copied as they only had to be renamed. Most of the structure from the testbench at sub-system level was adapted. This UVM framework was also tested with the same sequences. In addition, the UVM monitors at IP level and sub-system level were reused for monitoring the `UART` and sub-system. This demonstrated greatly the benefits of reuse in UVM, as previous work at IP level and sub-system level verified SoC behavior.

4.2.1 UVM at UART IP Level

A UVM testbench at UART IP level is created as seen in Figure 4.3. Simple verification has been done, including reading and writing of registers, and simple receive and transmit. Only valid frames with one frame format were tested, without parity and with one stop bit. The testbench follows the UVM structure as described in sub-section 2.1.1, starting with a top module, *top_uart*, where the UART is instantiated and *run_test* is called. Also, the interfaces are instantiated and connected here, and registered with the UVM factory as seen in Listing 4.1. The structure, apart from the UVM components, comes from existing setup from Nordic Semiconductor.

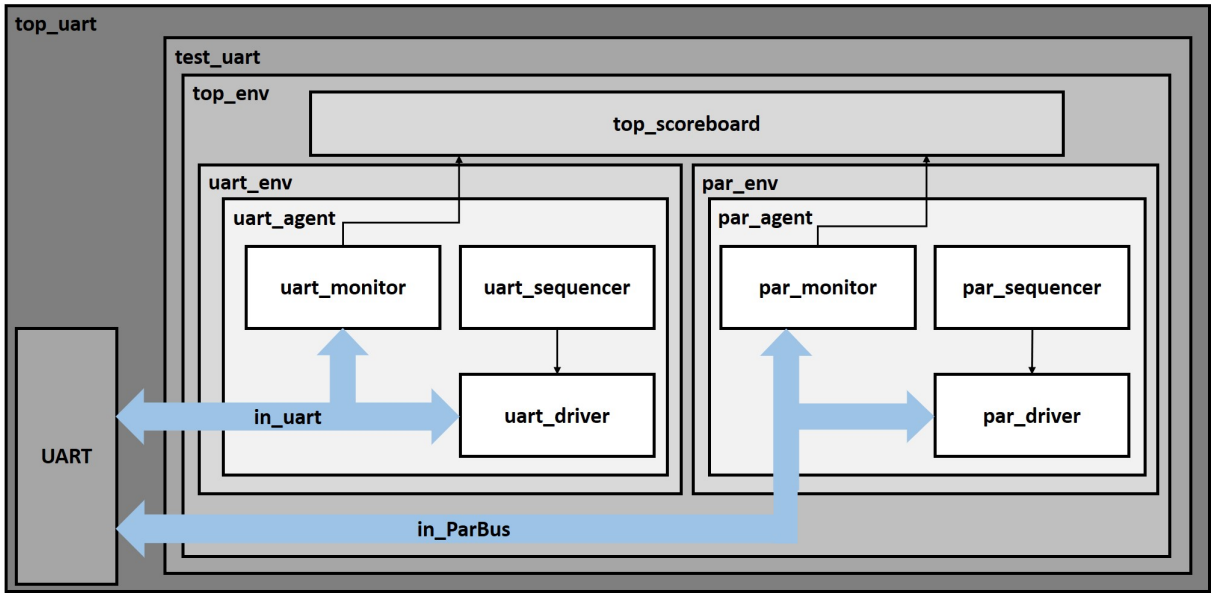

Figure 4.3: Block diagram of the UVM testbench for UART at IP level.

```
1 uvm_config_db#(virtual in_uart)::set(uvm_root::get(),"*","
    uin_uart",uin_uart);
2 uvm_config_db#(virtual in_ParBus)::set(uvm_root::get(),"*","
    uin_ParBus",uin_ParBus);
```

Listing 4.1: Registering interfaces with the UVM factory.

The first part of the UVM framework is *test_uart*, where the top UVM environment is instantiated together with different tasks, starting UVM sequences. The tasks consist of a write-read task, triggering sequences for *in_ParBus*, and a receive- and transmit task triggering sequences for *in_uart*. Listing 4.2 demonstrates how the receive sequence is triggered, but it is the same method for all of them.

```
1 uart_sequence_receive uart_rec_seq;
2 uart_rec_seq = new();
3 uart_rec_seq.start(env.uart0.agent.sequencer);
```

Listing 4.2: Calling the receive sequence from *test_uart*.

Inside the top environment, are separate environments for UART and PAR instantiated together with a top scoreboard. Listing 4.3 shows these instances, and how they are built in the build phase. The UART and PAR monitors are connected to the scoreboard in the connect phase. The UART and PAR environments builds their respective agents in a build phase with the same method as the UVM components in the top environment.

```
1  uart_env uart0;
2  par_env  par0;
3  top_scoreboard scoreboard;
4
5  function void build_phase(uvm_phase phase);
6    super.build_phase(phase);
7    uart0 = uart_env::type_id::create(.name("uart0"), .parent(
       this));
8    par0  = par_env::type_id::create(.name("par0"), .parent(this)
       );
9    scoreboard   = top_scoreboard::type_id::create(.name("
       scoreboard"), .parent(this));
10 endfunction
11
12 function void connect_phase(uvm_phase phase);
13   super.connect_phase(phase);
14   uart0.agent.monitor.item_collected_port.connect(scoreboard.
       item_collected_export_uart);
15   par0.agent.monitor.item_collected_port.connect(scoreboard.
       item_collected_export_par);
16 endfunction
```

Listing 4.3: Creating and connecting UART and PAR environments with scoreboard in top environment.

The UART and PAR environments use different transactions and compare functions, but both use the same scoreboard. Usually, a monitor and a scoreboard are connected through an analysis port. A write-function, with the sent transaction as an input, will be called in the scoreboard when the analysis port is written. This is demonstrated in Listing 4.4. The scoreboard used in the IP level testbench have been modified to handle two different analysis ports as seen in Listing 4.5. The *'uvm_analysis_imp_decl* macro creates a new *uvm_analysis_imp* and a write-function, where they get extended names based on the input to the macro. Thus, allowing multiple *uvm_analysis_imp* and write-functions to be declared.

```
1  uvm_analysis_imp#(transaction_name, scoreboard_name) item_name;
2
3  virtual function void write(transaction_name pkt);
4    compare_function(pkt);
5  endfunction
```

Listing 4.4: Analysis port and write-function in scoreboards.

```
1  `uvm_analysis_imp_decl ( _uart )
2  `uvm_analysis_imp_decl ( _par )
3
4  class top_scoreboard extends uvm_scoreboard;
5    uvm_analysis_imp_uart#(uart_transaction, top_scoreboard)
       item_collected_export_uart;
6    uvm_analysis_imp_par#(par_transaction, top_scoreboard)
       item_collected_export_par;
7
8    virtual function void write_uart(uart_transaction pkt);
9      uart_trans(pkt);
10   endfunction
11   virtual function void write_par(par_transaction pkt);
12     par_trans(pkt);
13   endfunction
```

Listing 4.5: Modified analysis ports and write-functions in scoreboard for IP level testbench.

4.2.1.1 UART UVM Environment

The UART agent builds the UART sequencer, driver and monitor in the build phase as seen in Listing 4.6. It also connects the driver and the sequencer in the connect phase. The driver and sequencer are only built and connected if the UVM agent is active, meaning it should be used for driving stimulus.

```
1  function void build_phase(uvm_phase phase);
2    super.build_phase(phase);
3    monitor = uart_monitor::type_id::create("monitor", this);
4    if(get_is_active() == UVM_ACTIVE) begin
5      driver = uart_driver::type_id::create("driver",this);
6      sequencer = uart_sequencer::type_id::create("sequencer",
         this);
7    end
8  endfunction
9
10 function void connect_phase(uvm_phase phase);
11   if(get_is_active() == UVM_ACTIVE)begin
12    driver.seq_item_port.connect(sequencer.seq_item_export);
13   end
14 endfunction
```

Listing 4.6: UART agent creating the sequencer, driver and monitor.

A transaction is specified for the UART, shown in Listing 4.7. This consist of a random variable *payload*, used for receiving and transmitting a random byte. Other variables are also created, such as data received, and data transmitted, where each is a byte wide. It also has an enumerator consisting of *RECEIVE* and *TRANSMIT* to separate the two modes. An enumerator for choosing between one and two stop bits is also present, but this is constrained to one bit as the testbench

is not created to support two bits. A delay variable is made so that frames are sent with different time in between.

```
1  typedef enum{RECEIVE, TRANSMIT} kind_e;
2  typedef enum{TWO_STOP_BITS, ONE_STOP_BIT} stop_e;
3
4  class uart_transaction extends uvm_sequence_item;
5    // Data and control fields
6    rand bit[7:0] payload; //rxd or txData
7    rand bit start_bit;
8    rand bit[1:0] stop_bits;
9    // Receive
10   bit rxDataReady;
11   bit[7:0] rxData;
12   // Transmit
13   bit[7:0] txd;
14   bit txDataRead;
15   // Control
16   rand kind_e kind;
17   rand stop_e stop_type;
18   rand int delay;
19   int uart_id;
20   // Constraints
21   constraint c_delay       { delay >= 1; delay < 20; }
22   constraint c_start_bit   { start_bit == 1'b0; }
23   constraint c_stop_bits   { stop_bits == 2'b11; }
24   constraint c_stop_bits_t { stop_type == ONE_STOP_BIT; }
```

Listing 4.7: The UART transaction.

Two sequences have been created, one for receiving and one for transmitting. Because these sequences were going to be imported to Questa inFact, inline constraints could not be used. This means that the transaction is requested and randomized with an input specifying a variable. For example, "*req.randomize() with {req.kind == RECEIVE;}*". As a result, two transactions. extending the *uart_transaction*, were created. These two transactions include a constraint for the *kind* variable, constraining it based on desired test case. Thus, the two sequences call a different transaction. This can be seen in Listing 4.8. The body task contains a repeat to request the transaction 5 times, to test more than one value. A sequencer is created to send transactions to the driver. This is connected with the base transaction, *uart_transaction*, but does also work with all extensions of it.

```
1  // Receive sequence
2  class uart_sequence_receive extends uvm_sequence#(
       uart_receive_transaction);
3    `uvm_object_utils(uart_sequence_receive)
4    function new(string name = "uart_sequence_receive");
5      super.new(name);
6    endfunction
7
8    task body();
9      repeat(5) begin
10       uart_receive_transaction rec_req =
           uart_receive_transaction::type_id::create("rec_req");
11       start_item(rec_req);
12       assert(rec_req.randomize());
13       finish_item(rec_req);
14     end
15   endtask
16 endclass
17
18 // Transmit sequence
19 class uart_sequence_transmit extends uvm_sequence#(
       uart_transmit_transaction);
20   `uvm_object_utils(uart_sequence_transmit)
21   function new(string name = "uart_sequence_transmit");
22     super.new(name);
23   endfunction
24
25   task body();
26     repeat(5) begin
27       uart_transmit_transaction trans_req =
           uart_transmit_transaction::type_id::create("trans_req"
           );
28       start_item(trans_req);
29       assert(trans_req.randomize());
30       finish_item(trans_req);
31     end
32   endtask
33 endclass
```

Listing 4.8: Receive and transmit UART sequences.

The driver requests a transaction from a sequence in the run phase. A receive- or transmit task is called based on the enumerator in the transaction, as seen in Listing 4.9. The *in_uart* interface is used to generate stimulus to the UART. The stimulus is applied to the UART as described in sub-section 4.1. During receive, the payload is written to the RX serial input, *rxd*, one bit at a time, and for TX, the payload is written to the transmit data byte input, *txData*.

```
task run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    case(req.kind)
      RECEIVE:  receive();
      TRANSMIT: transmit();
    endcase
    seq_item_port.item_done();
  end
endtask
```

Listing 4.9: Run phase of the UART driver.

The monitor monitors the *in_uart* interface. These signals are assigned to a transaction, and then sent to the scoreboard, as demonstrated in Listing 4.10. When receiving, the RX serial input bits are gathered into a payload, and RX data is set from the byte output. During transmission, the TX byte input is set as the payload, and the TX serial output bits are gathered into TX data. The scoreboard compares the random payload with the received/transmitted data as seen in Listing 4.11.

```
//Instantiate analysis port and uart transaction
uvm_analysis_port#(uart_transaction) item_collected_port;
uart_transaction trans_collected;
...
//Send transaction to scoreboard through analysis port
item_collected_port.write(trans_collected);
```

Listing 4.10: Sending a transaction though a analysis port in the UART monitor.

```
if(uart_pkt.kind == RECEIVE) begin
  // Check if bit stream input equals output RX data
  if(uart_pkt.payload == uart_pkt.rxData)
    `uvm_info(get_type_name(),$sformatf("- :: RECEIVEMatch :: -
      "),UVM_LOW)
  else
   `uvm_error(get_type_name(),"- ::RECEIVE MisMatch:: -")
end
else if(uart_pkt.kind == TRANSMIT) begin
  // Check if input TX data equals bit stream output
  if(uart_pkt.payload == uart_pkt.txd)
    `uvm_info(get_type_name(),$sformatf("- :: TRANSMITMatch ::
      -"),UVM_LOW)
  else
   `uvm_error(get_type_name(),"-::TRANSMIT MisMatch::-")
end
```

Listing 4.11: Comparison of expected received/transmitted data with actual received/transmitted data.

4.2.1.2 PAR UVM Environment

The elements of the UVM environment for PAR was created with the same methods as the UART UVM environment. Listing 4.12 shows the transaction, consisting of random variables for address, write data, write enable and read enable. An output variable for read data is also present. An enumerator, consisting of *READ* and *WRITE*, is used to separate reading and writing. Constraints are added for stopping read enable and write enable from being set at the same time.

```
1  typedef enum{READ, WRITE} inst_t;
2
3  class par_transaction extends uvm_sequence_item;
4    rand inst_t    inst;
5    rand bit[31:0] parDo;
6    rand bit[11:0] parAddr;
7    rand bit[3:0]  parWe;
8    rand bit       parRe;
9    bit[31:0]      parDi;
10   // Constraints
11   constraint C_read  { inst==READ -> parRe; inst==READ -> parWe
       ==4'h0; };
12   constraint C_write { inst==WRITE -> parWe==4'hF; inst==WRITE
       -> parRe=1'b0; };
```

Listing 4.12: The PAR transaction.

There are two different sequences, one read, and one write sequence. The read sequence constrains the enumerator to *READ* and has an address input for constraining the address to a specific value. The same applies to the write sequence, but with *WRITE* instead, in addition to a data input constraining data to be written. Listing 4.13 shows a sequence calling the write sequence, followed by the read sequence, together with the inputs. This is the sequence used by the test program. The address and data inputs are added in order to write and read specific registers.

```
1  class par_sequence_write_read extends uvm_sequence#(
     par_transaction);
2    `uvm_object_utils(par_sequence_write_read)
3    rand bit[11:0] addr;
4    rand bit[31:0] data;
5
6    function new(stringname = "",bit[11:0] addr =   0 ,bit[31:0]
       data =   0 );
7      super.new(name);
8      this.addr = addr;
9      this.data = data;
10   endfunction
11
12   par_sequence_write write_seq;
```

```
13    par_sequence_read  read_seq;
14    task body();
15      `uvm_do_with(write_seq, {write_seq.addr==addr;write_seq.data
          ==data;})
16      `uvm_do_with(read_seq, {read_seq.addr==addr;})
17    endtask
18  endclass
```

Listing 4.13: First write then read PAR sequence.

The driver requests a transaction in the run phase, likewise in the UART driver. The *in_ParBus* interface is set according to the values in the transaction. During a read operation, the address is set together with the read enable signal. When writing, the address is set, and data is written together with write enable.

The interface is monitored by the monitor, where values from *in_ParBus* are collected into a PAR transaction and sent to the scoreboard. During a read, the read data is stored in the transaction together with the address, and during a write, the written data is stored in the transaction with the address. The scoreboard stores the written value in an array corresponding to the address and compares this value with the read value. This is demonstrated in listing 4.14.

```
1  if(par_pkt.inst == WRITE)
2    sc_mem[par_pkt.parAddr] = par_pkt.parDo;
3  else if(par_pkt.inst == READ) begin
4    if(sc_mem[par_pkt.parAddr] == par_pkt.parDi)
5      `uvm_info(get_type_name(),$sformatf("- :: READ DATAMatch ::
           -"),UVM_LOW)
6    else
7      `uvm_error(get_type_name(),"- :: READ DATA MisMatch:: -")
8  end
```

Listing 4.14: Comparison of written of read value with the PAR transaction.

4.2.2 UVM at UART Sub-System Level

The UVM testbench for the UART at sub-system level can be seen in Figure 4.4. The sub-system is called PeripheralSubSystem and gathers all the peripheral IPs, connecting them together and allowing them to be used by an external environment. The *UART* is located inside a smaller sub-system, SerIOBox. The same structure of the UVM testbench, like at the IP level, has been used as well as the same simple receive and transmit sequences. The UART monitor from the IP level testbench is reused for monitoring the UART. The structure, apart from the UVM components, comes from existing setup from Nordic Semiconductor.

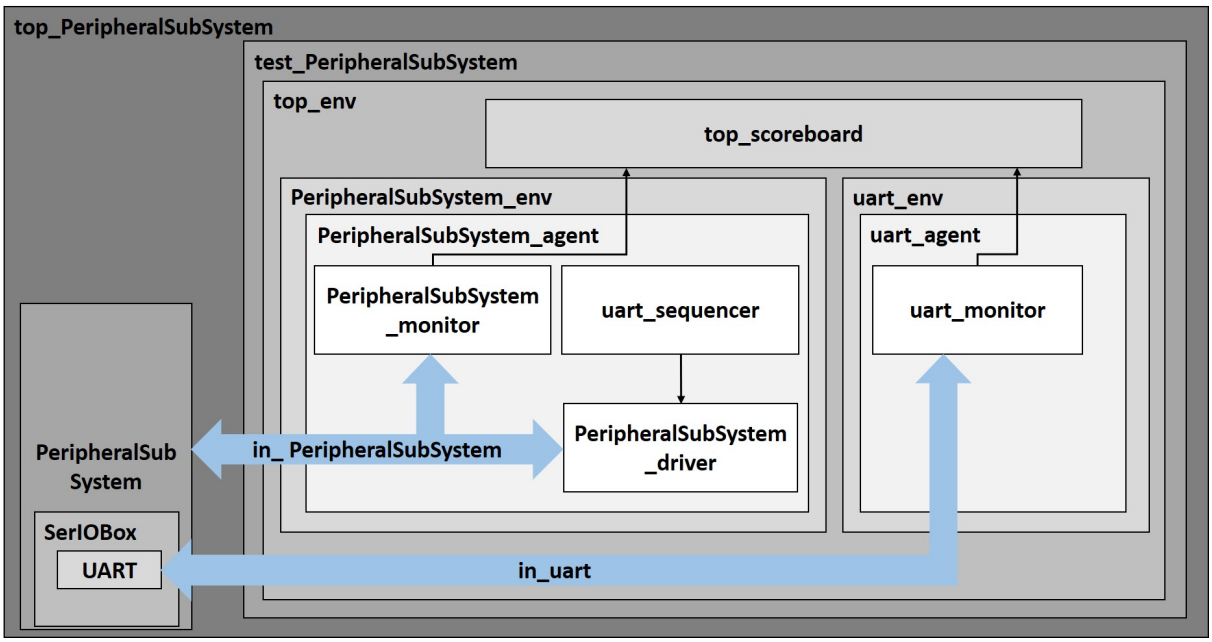

Figure 4.4: Block diagram of the UVM testbench for UART at sub-system level.

The top module, *top_PeripheralSubSystem*, instantiates the PeripheralSubSystem together with an instance mimicking the RAM, and calls *run_test*. Inside *test_PeripheralSub-System*, is the top environment instantiated, and registers are written before sequences for the interface *in_PeripheralSub-System* are called. The RAM model, and tasks for writing registers, are previously created by Nordic Semiconductor. The PeripheralSubSystem environment and the UART environment are created inside the top environment together with the scoreboard, like the top environment at IP level. The UART agent only builds the monitor by using *uvm_config_db* to set the UART environment as passive, i.e. not generating stimulus, as seen in listing 4.15.

```
1 uart0=uart_env::type_id::create(.name("uart0"), .parent(this));
2 uvm_config_db#(int)::set(this, "uart0", "is_active", 0);
```

Listing 4.15: Creating UART environment and setting it passive.

A new scoreboard has been created that collects transactions from the two monitors. The compare function for the UART monitor is the same as used at the IP level scoreboard. The compare function for the PeripheralSubSystem is a copy of the UART compare function. The reason for having two separate compare functions, that are equal, is because it is easier to debug when one can see which monitor that fails. Two different analysis ports and write-functions are achieved in the same way as with the scoreboard at the IP level testbench.

The sub-system contains four different UARTs. Therefore, the testbench has been extended to cover testing for all of them through the same UVM testbench. This involved adding a variable for the number of UARTs to be tested. The variable is instantiated in the environment, and can be configured in the top module through *uvm_config_db*, as shown in Listing 4.16. Inside the environment, this variable is used to create multiple PeripheralSubSystem agents. In addition, to keep track of which UART is being verified, an identifier variable was created inside

the driver and monitor. This is demonstrated in Listing 4.17, where each agent is given the same name, with a number suffix, and the identifier variable inside the monitor, and driver, is set to this number. The *in_PeripheralSubSystem* interface was extended with separate signals for each UART. Thus, only one interface is needed.

```
1 parameter NUM_SERIOBOX = 4;
2 uvm_config_db#(int)::set(null,"*",$sformatf("num_serIOBox"),
     NUM_SERIOBOX);
```

Listing 4.16: Setting the number of UARTs/SerIOBoxes.

```
1  PeripheralSubSystem_agent agent[];
2  function void build_phase(uvm_phase phase);
3    ...
4    agent = new[num_serIOBox];
5    for(int i = 0; i < num_serIOBox; i++) begin
6      $sformat(inst_name, "agent[%0d]", i);
7      agent[i] = PeripheralSubSystem_agent::type_id::create(
          inst_name, this);
8      void'(uvm_config_db#(int)::set(this,{inst_name,".monitor"},
           "PeripheralSS_id", i));
9      void'(uvm_config_db#(int)::set(this,{inst_name,".driver"},
          "PeripheralSS_id", i));
10   end
11 endfunction
```

Listing 4.17: Creating multiple PeripheralSubSystem agents.

As a result, a test, where two PeripheralSubSystem drivers are active at the same time during a receive, was added. However, in theory, it is possible to test all the UARTs at the same time. To achieve a more complex test, this test was created to receive exactly at the same time and with synchronous baud rate clocks (internal clock used when sampling received bits). This way, both UARTs write to the RAM simultaneously. Thus, it is possible to extend the testbench to cover different RAM priorities. However, the testbench only test with default priority setting, the lower the UART number, the higher the priority. As a result, the new test required a new sequence, where the delay is fixed. The new sequence was created likewise the two other sequences, in addition to a new transaction where *kind* is constrained to *RECEIVE* and *delay* to *1*.

These two variables also had to be created in the UART UVM framework, which required modifications of the IP level testbench. The number of UARTs are set in the IP top environment equal to the number in PeripheralSubSystems top environment, and the agents are created with the same method. Inside the top module, the UART interface had to be replicated equal to the number of UARTs, where the signals in each interface are assigned to the correct UART. This is done through a generate loop as shown in listing 4.18. Also, all the interfaces had to be registered with the UVM factory, where the method by adding a number suffix to the name was followed. Inside the monitor, each UART is monitored using the interface with the corresponding number suffix. All the created monitors could use the same port to the scoreboard. Thus, the same compare method could be used for all of them.

```
1  generate
2    for(genvar i = 0; i < NUM_SERIOBOX; i++) begin
3      // Uart interface
4      in_uart uin_uart();
5
6      assign uin_uart.rxd = u_PeripheralSubSystem.la_SerIOBox[i].
           u_SerIOBox.u_Uart.rxd;
7
8      initial begin
9        uvm_config_db#(virtualin_uart)::set(uvm_root::get(),"*",
             $sformatf("uin_uart%0d",i), uin_uart);
10     end
11   end
12 endgenerate
```

Listing 4.18: Creating multiple UART interfaces. (Only one assignment of many are shown)

Before the sequencer is started, the DMA size and address registers are set for both RX and TX. Then, the UART is enabled, before either starting RX and the receive sequence, or starting TX and the transmit sequence. This will enable the UART to receive bits or start transmission of RAM content through DMA.

4.2.2.1 PeripheralSubSystem UVM Environment

The PeripheralSubSystem agent, built inside the environment, builds the UART sequencer, PeripheralSubSystem driver and monitor. Thus, the UVM environment is created to support the same transaction and sequences as the UART environment. This is done in the same manner as with the UART agent.

Upon a sequence request in the driver, are either a receive- or transmit task called, in the same way as for the UART driver. The receive task controls a smaller pad bus interface, located inside *in_PeripheralSubSystem*, mimicking GPIO connections. The pad bus is signalled according to the payload and according to which UART is being tested. The transmit task only writes the payload to the RAM. When all calls to the transmit sequences are done, will the content of the RAM be transmitted by starting transmission.

RX is monitored through the written data on the pad bus. After a frame has been sent, data on the bus going to the RAM, written through DMA, are monitored. The data written to RAM is stored as RX data inside the transaction, while the pad bus values are stored as the payload. Then is the transaction sent to the scoreboard in the same way as at IP level. For TX, the pad bus is monitored, and its values are stored as the TX data. The value read through DMA, on the RAM bus, is monitored and set as payload. This is then sent to the scoreboard.

4.2.3 UVM at UART SoC Level

Figure 4.5 demonstrates the UVM testbench at SoC-level. It should be noted that the `PeripheralSubSystem` lies several layers down in the hierarchy than what is shown in the figure. This testbench used a previously created framework by Nordic Semiconductor to create the DUT and the environment around it. This includes instantiating the SoC, and a helper-framework around this with access to different interfaces and different tasks to interact with the SoC. The UVM framework replaces the test program, while the rest is made up of the previously existing framework. The same method with configurable number of `UARTs`, as in the sub-system level, was followed. The monitors from IP and sub-system level were reused for monitoring `UART` and sub-system behavior.

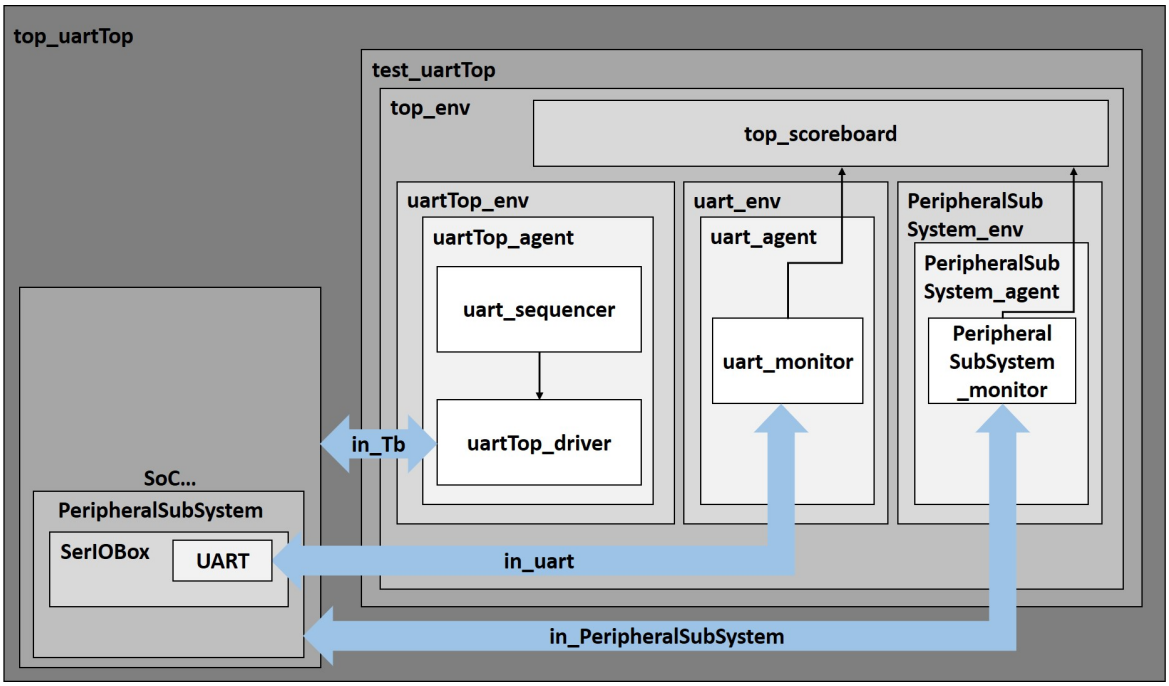

Figure 4.5: Block diagram of the UVM testbench for `UART` at SoC level.

Inside the top module *top_uartTop*, the SoC is instantiated together with a top-level interface. The same method for multiple `UART` interfaces in the sub-system testbench is followed. The `PeripheralSubSystem` interface is also instantiated here. *run_test* is called from this module. *test_uartTop* instantiates the top environment, and calls a receive- and transmit task, in addition to the task with two `UARTs` receiving simultaneous. Inside the top environment, the `UART` `PeripheralSubSystem` and the SoC environment are created. The `UART` and `PeripheralSubSystem` environments only consists of a monitor, while the SoC environment does not have a monitor. The top scoreboard is reused from sub-system level.

Each task in *test_uartTop* begins with configuring the `SerIOBox`/`UART` through writing of registers. This involves configuring which GPIOs to be used and enabling the `UART` with DMA. This is followed by calling a task for RX or TX depending on what to test. These tasks are created in the same way as at sub-system level.The testbench runs the receive- and transmit task for all the different `UARTs`. This is achieved through an input parameter to change between them. A separate receive task where two `UARTs` are driven at the same time is then called.

4.2.3.1 UART top UVM Environment

The SoC UVM environment for the `UART` consists of a UVM agent that contains a driver and the `UART` sequencer. Thus, there is no monitor. This is reasoned as the goal is to verify the `UART`, which is being achieved through the `UART` and `PeripheralSubSystem` monitors.

This UVM environment is also built around the `UART` transaction. The driver works in the same way as in the sub-system level, but instead of using a pad bus mimicking the GPIOs, the actual GPIOs are used. Separate GPIOs are set for each of the `UARTs`, and the identifier variable selects which GPIO to be toggled. For the transmission, the actual RAM in the SoC is written with the payload.

4.3 C Code Framework

The C code generated from inFact only generates the stimulus (what is being tested), but not how to test it on the SoC. Therefore, a C code framework, where the stimulus is applied to the SoC, is necessary. A method for connecting generating C code, and the framework, is to have the generated code call a function in the framework with the stimulus as inputs. This is similar to a UVM sequence being sent to a driver, where the driver handles the received transaction. Thus, the C code framework needs a function where it either calls a receive- or transmit function based on the **kind** enumeration, just as in the UVM driver.

The C code is run in a co-simulation environment by loading the FW into a SystemVerilog testbench. This means that the C code is run on the CPU with an RTL description of the SoC. The created C code consists of equivalent receive- and transmit tests as the UVM testbenches. Keil, a software development environment made for arm processors, is used for compiling the C code. This is because it generates a *.axf.vhx.32* file, which will be used in the testbench.

All scenarios testing the `UART` in the UVM testbenches do not entirely make sense in C code. This is because one has greater control over the hardware with RTL simulation, as well as writing hardware specific tests. For example, one can drive simulation by monitoring edges of relevant signals, such as clocks. While with software, this is not the case. Software relies on using the CPU or GPIO pins to communicate with the SoC and does not have any internal access.

The way transmission is tested in the SoC UVM testbench is equivalent to how one would do it in C code, as the RAM is written by the CPU, and transmission is self-driven after being stated. This is not the case with the receiving test. First, the receive test does not make sense in C code. Who writes a software program triggering GPIO pins to receive information from itself, instead of just using the CPU to write it? In the UVM testbench, the test relies on applying a new bit at each clock tick. This is not easily done in C code. The receiving should be done by transmission from another SoC. However, the intention of this thesis is to do a proof of concept. Thus, the C code is created to imitate the UVM testbench at SoC as much as possible. This way, the exact same PS code in inFact could be used.

4.3.1 UART C Code

The C code framework consists of a main-function as seen in Listing 4.19. The UART is here configured in two initialize functions, *UartRxInit()* and *UartTxInit()*. All functions are included from the *uart_RxTx.h* header file. Next, a timer is initialized. The timer is used to notify the FW at each clock tick in order to know when to apply a new bit when receiving. This is a workaround for the unusual coding styles to receive from itself. After initialization, the is UART enabled and RX is started. Now, the UART can receive information, and the RAM will be written before transmission. This will be performed through a call to the inFact code. Last, the transmission will begin, and when finished, the UART will be disabled, and the test is done. The while-loop is added to prevent the FW from restarting when finished, as otherwise happens when run in the SystemVerilog testbench.

```
1  int main(void){
2    //Initialize UART and TIMER
3    UartRxInit();
4    UartTxInit();
5    TimerInit();
6
7    //Enable UART and start RX
8    StartRx();
9
10   //Start Tx and disable UART when finished
11   SendTx();
12
13   //Forever loop to prevent C code from restarting in TB
14   while(true){}
15 }
```

Listing 4.19: main-function of the C code framework.

Listing 4.20 demonstrates the implementation of the UART initialize-functions. *UartRxInit()* configures GPIO pins, *GPIO_RXD* and *GPIO_CTS*, as well as RAM location and size of receive buffer. The GPIO number for these macros are defined in the header file. An additional GPIO pin is configured. This will be used by the FW when applying bits to be received. The receive pin used by the UART is assigned to this pin in the SystemVerilog testbench. The reason for this is that the pin used for receiving by the UART, cannot be configured to be directly set by the FW. This is because the FW relies on CPU writes to toggle the GPIO pins.

```
1  void UartRxInit(void){
2    //Pin select
3    NRF_UARTE0->PSEL.RXD = (GPIO_RXD << UARTE_PSEL_RXD_PIN_Pos) |
         (UARTE_PSEL_RXD_CONNECT_Connected <<
        UARTE_PSEL_RXD_CONNECT_Pos);
4    NRF_UARTE0->PSEL.CTS = (GPIO_CTS << UARTE_PSEL_CTS_PIN_Pos) |
         (UARTE_PSEL_CTS_CONNECT_Connected <<
        UARTE_PSEL_CTS_CONNECT_Pos);
5
```

```
6   //Set DMA pointer and buffer size
7   NRF_UARTE0->RXD.PTR = (0x20038000 << UARTE_RXD_PTR_PTR_Pos);
8   NRF_UARTE0->RXD.MAXCNT= (0xF << UARTE_RXD_MAXCNT_MAXCNT_Pos);
9
10  //Set up GPIO for setting RX
11  NRF_GPIO->PIN_CNF[GPIO_RX] = (GPIO_PIN_CNF_DIR_Output <<
       GPIO_PIN_CNF_DIR_Pos) | (GPIO_PIN_CNF_INPUT_Connect <<
       GPIO_PIN_CNF_INPUT_Pos) | (GPIO_PIN_CNF_PULL_Disabled <<
       GPIO_PIN_CNF_PULL_Pos) | (GPIO_PIN_CNF_DRIVE_S0S1 <<
       GPIO_PIN_CNF_DRIVE_Pos) | (GPIO_PIN_CNF_SENSE_Disabled <<
       GPIO_PIN_CNF_SENSE_Pos);
12  NRF_GPIO->OUT = (0x1 << GPIO_RX);
13 }
14
15 void UartTxInit(void){
16  //Pin select
17  NRF_UARTE0->PSEL.TXD = (GPIO_TXD << UARTE_PSEL_TXD_PIN_Pos) |
        (UARTE_PSEL_TXD_CONNECT_Connected <<
       UARTE_PSEL_TXD_CONNECT_Pos);
18  NRF_UARTE0->PSEL.RTS = (GPIO_RTS << UARTE_PSEL_RTS_PIN_Pos) |
        (UARTE_PSEL_RTS_CONNECT_Connected <<
       UARTE_PSEL_RTS_CONNECT_Pos);
19
20  //Set DMA pointer and buffer size
21  NRF_UARTE0->TXD.PTR = (0x20038100 << UARTE_RXD_PTR_PTR_Pos);
22  NRF_UARTE0->TXD.MAXCNT= (0xA << UARTE_RXD_MAXCNT_MAXCNT_Pos);
23
24  //Set up ENDTX event to trigger GPIO through PPI
25  NRF_UARTE0->PUBLISH_ENDTX = (UARTE_PUBLISH_ENDTX_EN_Enabled
       << UARTE_PUBLISH_ENDTX_EN_Pos) | (0x1 <<
       UARTE_PUBLISH_ENDTX_CHIDX_Pos);
26  NRF_GPIO->PIN_CNF[GPIO_ENDTX] = (GPIO_PIN_CNF_DIR_Output <<
       GPIO_PIN_CNF_DIR_Pos) | (GPIO_PIN_CNF_INPUT_Connect <<
       GPIO_PIN_CNF_INPUT_Pos) | (GPIO_PIN_CNF_PULL_Disabled <<
       GPIO_PIN_CNF_PULL_Pos) | (GPIO_PIN_CNF_DRIVE_S0S1 <<
       GPIO_PIN_CNF_DRIVE_Pos) | (GPIO_PIN_CNF_SENSE_Disabled <<
       GPIO_PIN_CNF_SENSE_Pos);
27  NRF_GPIOTE->CONFIG[1] = (GPIOTE_CONFIG_OUTINIT_Low <<
       GPIOTE_CONFIG_OUTINIT_Pos) | (
       GPIOTE_CONFIG_POLARITY_LoToHi <<
       GPIOTE_CONFIG_POLARITY_Pos) | (GPIO_ENDTX <<
       GPIOTE_CONFIG_PSEL_Pos) | (GPIOTE_CONFIG_MODE_Task <<
       GPIOTE_CONFIG_MODE_Pos);
28  NRF_GPIOTE->SUBSCRIBE_OUT[1] = (
       GPIOTE_SUBSCRIBE_OUT_EN_Enabled <<
       GPIOTE_SUBSCRIBE_OUT_EN_Pos) | (0x1 <<
       GPIOTE_SUBSCRIBE_OUT_CHIDX_Pos);
```

```
29    NRF_DPPIC->CHENSET = (DPPIC_CHENSET_CH1_Set <<
          DPPIC_CHENSET_CH1_Pos);
30  }
```

Listing 4.20: RX and TX initialize functions.

UartTxInit() configures GPIO pins, *GPIO_TXD* and *GPIO_RTS*, as well as RAM location and size of transmit buffer. Here is also an additional GPIO pin configured. This is connected through a Programmable Peripheral Interface (PPI) channel. A PPI allows peripherals to communicate without the CPU. The *ENDTX* event, generated when the TX buffer has been transmitted, is sent to a task on the GPIO, which will trigger the configured GPIO. This is achieved by publishing the *ENDTX* event and having the GPIO tasks and events controller (`GPIOTE`) subscribe to this. They are configured to use PPI channel one and `GPIOTE` channel one, that will trigger GPIO *GPIO_ENDTX*.

In *TimerInit*, the timer is configured with a *4 us* clock, which has the same rate as the clock used by the `UART`, and it is configured to generate an event after one clock tick. This can be seen in Listing 4.21. This is also achieved through the PPI. When the `TIMER` has counted to one, a compare event (CC) will publish on PPI channel zero, which the `GPIOTE` is subscribing to. Then will *GPIO_TM* be set. This is followed by starting the timer.

```
1  void TimerInit(void){
2    //Configure and setup TIMER0 to generate event each 4us (brg
         clock-rate)
3    NRF_TIMER0->PRESCALER=(0x6 << TIMER_PRESCALER_PRESCALER_Pos);
4    NRF_TIMER0->CC[0] = 0x1;
5    NRF_TIMER0->PUBLISH_COMPARE[0] = (
         TIMER_PUBLISH_COMPARE_EN_Enabled <<
         TIMER_PUBLISH_COMPARE_EN_Pos) | (0x0 <<
         TIMER_PUBLISH_COMPARE_CHIDX_Pos);
6    NRF_GPIO->PIN_CNF[GPIO_TIM] = (GPIO_PIN_CNF_DIR_Output <<
         GPIO_PIN_CNF_DIR_Pos) | (GPIO_PIN_CNF_INPUT_Connect <<
         GPIO_PIN_CNF_INPUT_Pos) | (GPIO_PIN_CNF_PULL_Disabled <<
         GPIO_PIN_CNF_PULL_Pos) | (GPIO_PIN_CNF_DRIVE_S0S1 <<
         GPIO_PIN_CNF_DRIVE_Pos) | (GPIO_PIN_CNF_SENSE_Disabled <<
         GPIO_PIN_CNF_SENSE_Pos);
7    NRF_GPIOTE->CONFIG[0] = (GPIOTE_CONFIG_OUTINIT_Low <<
         GPIOTE_CONFIG_OUTINIT_Pos) | (
         GPIOTE_CONFIG_POLARITY_LoToHi <<
         GPIOTE_CONFIG_POLARITY_Pos) | (GPIO_TIM <<
         GPIOTE_CONFIG_PSEL_Pos) | (GPIOTE_CONFIG_MODE_Task <<
         GPIOTE_CONFIG_MODE_Pos);
8    NRF_GPIOTE->SUBSCRIBE_OUT[0] = (
         GPIOTE_SUBSCRIBE_OUT_EN_Enabled <<
         GPIOTE_SUBSCRIBE_OUT_EN_Pos) | (0x0 <<
         GPIOTE_SUBSCRIBE_OUT_CHIDX_Pos);
9    NRF_DPPIC->CHENSET = (DPPIC_CHENSET_CH0_Set <<
         DPPIC_CHENSET_CH0_Pos);
```

```
10    NRF_TIMER0->TASKS_START = 0x1;
11  }
```

Listing 4.21: Initialize timer to generate event each clock tick.

Listing 4.22 demonstrates the functions used for the RX part. *StartRx()* enables the UART, and start the RX. Restarting of the timer is done in *waitForTimer()*. Here, the FW waits for the timer GPIO to be set, and when this happens, the timer and the GPIO will be cleared. This function is utilized in the receive function *SetRx()*. Receiving is performed by waiting for the timer and then apply information, first the start bit, then the payload and last the stop bit. Equivalent as in the SoC UVM testbench.

```
1  void StartRx(void){
2    //Enable UARTE0 with DMA
3    NRF_UARTE0->ENABLE = (UARTE_ENABLE_ENABLE_Enabled <<
       UARTE_ENABLE_ENABLE_Pos);
4
5    //Start RX
6    NRF_UARTE0->TASKS_STARTRX = 0x1;
7  }
8
9  void waitForTimer(void){
10   while(NRF_GPIO->PIN[GPIO_TIM].IN == 0x0){}
11   NRF_TIMER0->TASKS_CLEAR = 0x1;
12   NRF_GPIOTE->TASKS_CLR[0] = 0x1;
13 }
14
15 void SetRx(uint32_t payload, uint32_t delay){
16   char RxBit;
17   waitForTimer();
18
19   //Start bit
20   waitForTimer();
21   NRF_GPIO->OUT = (0x0 << GPIO_RX);
22
23   //Payload
24   for(uint32_t i = 0; i < 8; i++){
25     waitForTimer();
26     RxBit = (payload >> i);
27     NRF_GPIO->OUT = (RxBit << GPIO_RX);
28   }
29
30   //Stop bit
31   waitForTimer();
32   NRF_GPIO->OUT = (0x1 << GPIO_RX);
33
34   //Delay
```

```
35    for(uint32_t j = 0; j < delay; j++){
36      waitForTimer();
37    }
38  }
```

Listing 4.22: Functions for RX.

Transmission is carried out by writing a payload to the RAM location specified in the initialize function, as shown in function *SendTx()* in Listing 4.23. After each write, an *offset* variable is incremented, so that the next write will happen at the next RAM location. *SendTx()* starts transmission of the written RAM content, and wait for it to finish before disabling the UART.

```
1  void WriteTx(uint32_t payload){
2    //Write to RAM
3    *(volatile int *) (0x20038100 + offset) = payload;
4
5    offset++;
6  }
7
8  void SendTx(void){
9    //Start TX
10   NRF_UARTE0->TASKS_STARTTX = 0x1;
11
12   //Wait for completion
13   while(NRF_GPIO->PIN[GPIO_ENDTX].IN == 0x0){}
14
15    //Disable UART
16   NRF_UARTE0->ENABLE = (UARTE_ENABLE_ENABLE_Disabled <<
       UARTE_ENABLE_ENABLE_Pos);
17 }
```

Listing 4.23: Function for TX.

The *do_RxTx* function is what the generated C code from inFact will call. Here, either the receive- or transmit function will be called, based on the *kind* enumerator. Just as in the UVM testbenches.

```
1  void do_RxTx(uint32_t payload, enum kind_e kind, uint32_t delay
     ){
2    if(kind == RECEIVE){
3      SetRx(payload, delay);
4    }
5    else if(kind == TRANSMIT){
6      WriteTx(payload);
7    }
8  }
```

Listing 4.24: Function called in PSS generated C code.

4.3.2 Testbench Calling FW

The SystemVerilog testbench created to run the FW can be seen in Listing 4.25. The first lines assign the FW-controlled GPIO to the GPIO used for receiving. This testbench consists of one task, which loads the FW, resets the device to load it into the CPU model, and waits for it to finish.

```
1  //GPIO used by RXD in UART, assigned to GPIO in FW applying
       receive information
2  initial begin
3    uin_DevA.GPIO[0].ctrl.en_control = 1'b1;
4  end
5  assign uin_Tb.uin_DevA.GPIO[0].ctrl.in = uin_Tb.uin_DevA.GPIO
       [5].ctrl.out;
6
7  task ta_TransferRxTxDataWithDma();
8    // Load firmware
9    uin_DevA.uin_Tasks.APPMCU.uin_AppFlashMem_Tasks.ta_LoadROM(.
       filename(fwuart));
10
11   // Reset device to load FW
12   #10ns;
13   begin
14     -> uin_DevA.uin_Ctrl.reset;
15     #0;
16     wait(uin_DevA.uin_Ctrl.resetDone);
17   end
18
19   fork : la_Timeout
20     begin
21       wait(`DevA.`AppPer_SS.la_SerIOBox[0].u_SerIOBox.u_Uart.
           uartEnable);
22       wait(!(`DevA.`AppPer_SS.la_SerIOBox[0].u_SerIOBox.u_Uart.
           uartEnable));
23     end
24     begin
25       #3000us;
26       $error("Test timed out");
27     end
28   join_any
29   disable la_Timeout;
30 endtask
```

Listing 4.25: SystemVerilog testbench starting the FW.

4.4 Portable Stimulus Code

The rule language in inFact, which will be referred to as PS code, is not familiar from before. The first task was to get familiar with the tool. This was done by reading and understanding the inFact documentation. Also, a workshop with Mentor on behalf of Nordic Semiconductor was conducted. The workshop consisted of a presentation and explanation of inFact, as well as learning the tool from three labs/tasks. Afterwards, the PS code could be created.

inFact supports importing of a testbench, so that the content in the rules file will be automatically created. The rules file content is created through existing transactions (*uvm_sequence_item*) and sequences. Since the UVM frameworks were tested with sequences already made, these sequences could be used. These were imported through the inFact IDE.

A part of this thesis is to get an idea on how much effort it requires to create a PS code. Therefore, a PS code was created from scratch. It was created as a single sequence, mixing receiving and transmission. This resulted in a modification to the UVM testbench. Afterwards, the imported sequences were tested with the UVM testbenches. Then followed by testing the code generated from the PS code written from scratch with the sub-system and SoC UVM testbenches, and with the C code.

4.4.1 Using Testbench Import

A new workspace in inFact was created and opened in the IDE. In the IDE the testbench is imported through a work-directory (compiled testbench), followed by picking the package where the transaction and sequences are included. The transaction to be imported should then be defined. An inFact UVM sequence will be created from this by specifying a base sequence. These steps were followed for all the three different transactions and sequences. As a result, three test components were made with a separate rules file and inFact sequence.

4.4.1.1 Generated Rules file

The **rule_segment** created from the `UART` transaction is presented in Listing 4.26, and is used as a base by all of the sequences. Here, the **rand logic/int** variables have been translated into **meta_action**. The **constraints** are the same.

```
1  rule_segment{
2    import"paTest_uart.rseg";
3    struct uart_transaction {
4      meta_action payload[unsigned 7:0];
5      meta_action start_bit[unsigned 0:0];
6      meta_action stop_bits[unsigned 1:0];
7      meta_action kind<paTest_uart::kind_e>[kind_e];
8      meta_action stop_type<paTest_uart::stop_e>[stop_e];
9      meta_action delay[signed 31:0];
10
11     constraint c_delay {delay < 20; delay >= 1;};
12     constraint c_start_bit {start_bit == 1'b0;};
13     constraint c_stop_bits {stop_bits == 2'h3;};
```

```
14      constraint c_stop_bits_t {stop_type == ONE_STOP_BIT;};
15    }
16  }
```

Listing 4.26: Rule_segement created from the UART transaction.

A second **rule_segment**, extending the base **rule_segment** of the UART transaction, is created based on the receive transaction. This involves adding a **constraint** specifying it to receive, just as in the receive transaction. Listing 4.27 demonstrates this. The generated **rule_segment** from the imported receive, with fixed delay transaction, are identical, but with an added **constraint**, setting *delay* equal one. For the imported transmit transaction, this **constraint** is set to *TRANSMIT*.

```
1 rule_segment{
2   import"uart_transaction.rseg";
3   struct uart_receive_transaction extends uart_transaction{
4     constraint c_receive {kind == RECEIVE;};
5   }
6 }
```

Listing 4.27: Rule_segement created from the UART receive transaction.

The generated sequence, based on the imported transaction- and receive sequence, will be used by the UVM frameworks. The sequence is called *infact_uart_receive_transaction_gen*. This sequence overrides the **body**-function of the imported receive sequence. A set of tasks are created, containing behavior for assigning variables, starting and ending a transaction and checking coverage. For example, each of the **meta_action** variables are set through an action task as seen in Listing 4.28. This input value is set by inFact during simulation.

```
1 virtual task action_uart_receive_transaction_inst__delay(
     longint meta_val);
2   m_uart_receive_transaction_inst.delay = meta_val;
3 endtask
```

Listing 4.28: inFact receive sequence created from the UART receive sequence.

The complete **rule_graph** is shown in Listing 4.29. Here, the receive **rule_segment** is imported and declared, together with using **attributes** to override inFact sequence variables. The **attributes** are set to default. The **actions** *init* and *infact_checkcov* are declared and called in the rule sequence. They are default **actions** generated by inFact. The inFact receive sequence that is instantiated and connected through an **interface**, before being called in the rule sequence.

```
1  rule_graph infact_uart_receive_transaction_gen {
2    attributes infact_uart_receive_transaction_gen {
3      call_pre_post_randomize =true;
4      unroll_array_max = 256;
5      unroll_dynamic_arrays =false;
6    }
7    import "uart_receive_transaction.rseg";
8    attributes infact_uart_receive_transaction_gen {
9      base_class="paTest_uart::uart_sequence_receive";
10   }
11   interface do_item(uart_receive_transaction);
12   uart_receive_transaction uart_receive_transaction_inst;
13   action init;
14   action infact_checkcov;
15   infact_uart_receive_transaction_gen =
16     init
17     repeat{
18       do_item(uart_receive_transaction_inst)
19       infact_checkcov
20     };
21 }
```

Listing 4.29: rule_graph created from the UART receive sequence.

4.4.1.2 Generated Graph

From the **rule_graph** for the receive sequence, the graph is generated as seen in Figure 4.6. The graph looks the same for the other sequences. Here, the receive transaction is represented, where each **meta_action** has become a node. First, the *init* is traversed, followed by the transaction, though the *do_item()* call. This involves randomizing values for all the **meta_actions**. After, the coverage goal is checked, and if the goal is not met will the transaction be traversed again.

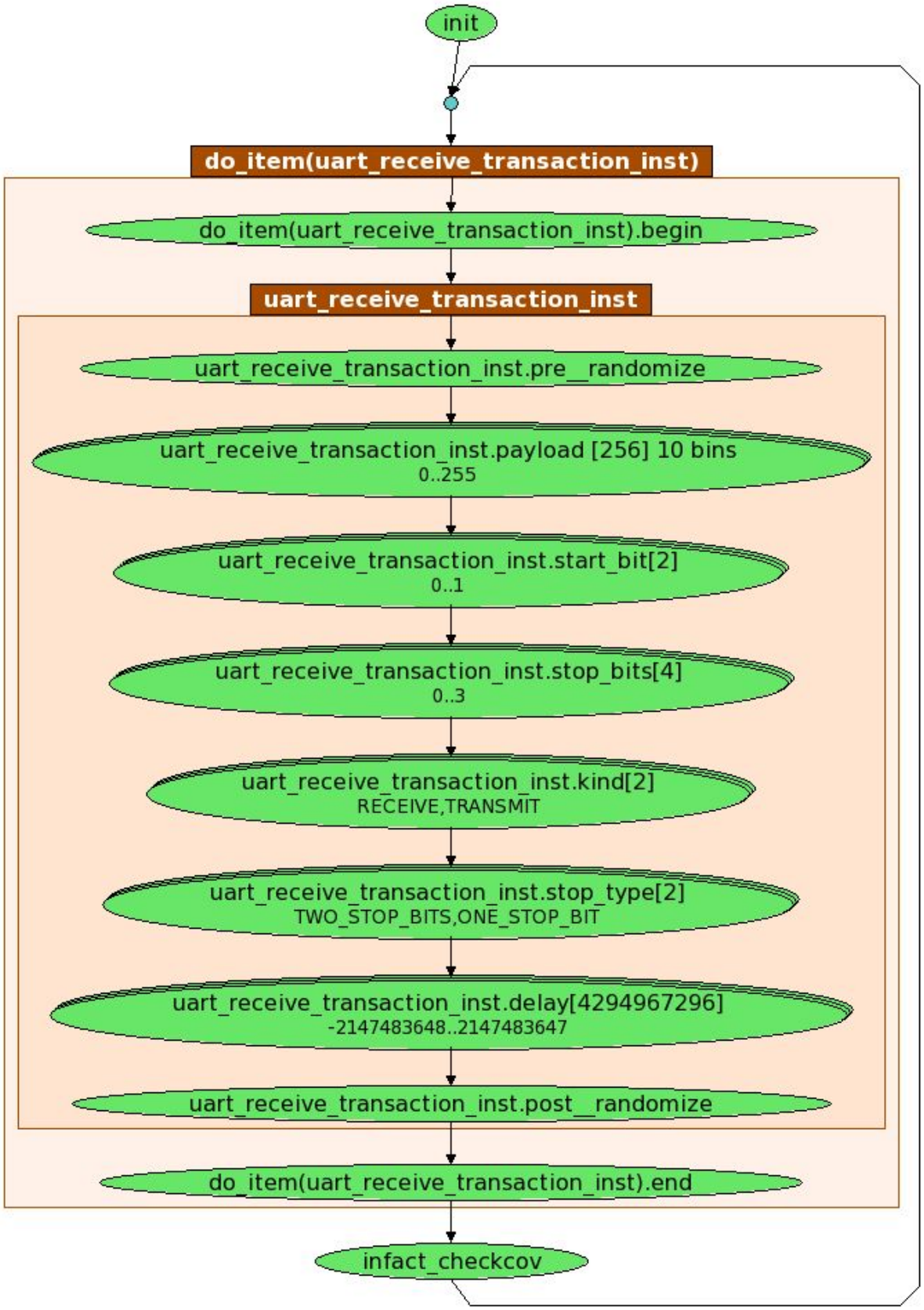

Figure 4.6: Graph generated from receive sequence.

4.4.1.3 Defining Coverage Strategy

The coverage strategy is the same for all the three sequences. Of all the **meta_action** variables, only the *payload* is of interest regarding coverage. The others are constrained to a certain value, while the value of *delay* is not important as it is not `UART` specific. Thus, there is no necessity to create a path coverage (cross coverage).

An action coverage is specified for the *payload* through the IDE. This is done by opening a coverage strategy window of the graph, and right-clicking on the *payload* node and selecting action coverage. This automatically creates a bin for each possible value, which is 256. It is not necessary to simulate all possible values, so a **bins** for the *payload* is specified in the **rule_segment** for each of the test components.

The *payload* **bins** is defined to contain the three lowest and highest values possible, in addition to splitting the values in between into four bins. This is done by adding the **bins** statement "*bins payload [0] [1] [2] [3..252]/4 [253] [254] [255]*". Thus, a total of ten bins will be created that can be seen in the graph for the *payload* node in Figure 4.6. The simulation will be terminated after each bin is hit once, i.e. calling the sequence ten times. The bin is defined this way because it covers corner cases, the lowest and highest values, in addition to "normal" values.

4.4.2 Creating from Scratch

Listing 4.30 shows the contents of a **rule_graph** that is automatically generated when creating a new test component. The names are based on the name given to the test component, which is here "*infac_RxTx_seq*". This means that the only part that needs to be filled out is the content of the **struct**. This involves adding **meta_actions**, **constraints** and **bins**. It should be noted that inFact only generates sequences, so a transaction and sequencer need to be created in the UVM framework.

```
1  rule_graph infact_RxTx_seq {
2    action init, infact_checkcov;
3
4    struct infact_RxTx_seq_item_c {
5      // TODO: Declare meta-actions
6    }
7
8    infact_RxTx_seq_item_c infact_RxTx_seq_item_c_inst;
9
10   interface do_item(infact_RxTx_seq_item_c);
11
12   infact_RxTx_seq = init repeat{
13     do_item(infact_RxTx_seq_item_c_inst)
14     infact_checkcov
15   };
16 }
```

Listing 4.30: rule_graph automatically generated when creating a new test component.

Because a transaction is already specified, the relevant fields can be copied manually. The **rand** variables becomes **meta_actions** and the **constraints** can be copied. The variables used for storing outputs are ignored as inFact only cares about generating (input) stimulus. The result can be seen in Listing 4.31. The **struct** is renamed to the same name as the transaction used in the UVM testbenches and can be used directly in them. However, from experimenting, it does not seem like this rename is necessary. The **kind** variable is not constrained, as this sequence should be a mix of receive and transmit. For some reason, the enumerators cannot be directly specified in the **struct**, as this is not compatible with the existing testbenches. Therefore, **set** is used to determine the enumerators. The generated code contains the same functions as the generated code from the imported sequences.

```
1  rule_graph infact_RxTx_seq {
2    action init, infact_checkcov;
3
4    set stop_e[signed enum TWO_STOP_BITS, ONE_STOP_BIT];
5    set kind_e[signed enum RECEIVE, TRANSMIT];
6    struct uart_transaction {
7      meta_action payload [unsigned 7:0];
8      meta_action start_bit [unsigned 0:0];
9      meta_action stop_bits [unsigned 1:0];
10     meta_action delay [signed 31:0];
11     meta_action kind <kind_e>[kind_e];
12     meta_action stop_type <stop_e>[stop_e];
13
14     constraint c_delay{delay < 20; delay >= 1;}
15     constraint c_start_bit{start_bit == 1'b0;}
16     constraint c_stop_bit{stop_bits == 2'h3;}
17     constraint c_stop_bits_t{stop_type == ONE_STOP_BIT;}
18
19     bins payload [0] [1] [2] [3..252]/4 [253] [254] [255];
20   }
21
22   uart_transaction uart_transaction_inst;
23   interface do_item(uart_transaction);
24
25   infact_RxTx_seq = init repeat {
26     do_item(uart_transaction_inst)
27     infact_checkcov
28   };
29 }
```

Listing 4.31: rule_segment created based on the transaction used in the UVM frameworks.

4.4.2.1 Modifying to C Code

The same rules file can be used for the C code, but not the same test component. A new test component of the type "*SDV OVM/UVM Sequence*" is created, where SDV stands for Software Driven Verification. This test component is intended for simultaneous verification of hardware and software in embedded software applications. However, it can also be used to create regular C code without using the hardware part.

The rules file from Listing 4.31 is copied to this test component. However, the start bit- and stop bit **meta_actions**, as well as the stop bit enumerator, are removed. This is because they are specified to a specific value in inFact and might as well be hard-coded in the C code. As a result, the code is less cluttered. An **action** and **attributes** are specified in the **rule_graph** as seen in Listing 4.32. This is required by inFact in order to communicate with the C code framework. The action creates a new node in the graph, which generates a function call. Inside this function, the *do_RxTx()* function is called, as specified by the **attributes**. The function is called with input values set by inFact. An **attributes** is specified for the **rule_graph**, which states to include the header file, where *do_RxTx()* is declared. The generated code contains the same set of functions for setting values and checking coverage as the UVM generated code, in addition to create and run functions.

```
1  struct uart_transaction {
2    ...
3    action do_RxTx;
4    attributes do_RxTx {
5      sw_action_stmt = "do_RxTx(fields->
           infact_uart_RxTx_item_c_inst.payload, fields->
           infact_uart_RxTx_item_c_inst.kind, fields->
           infact_uart_RxTx_item_c_inst.delay);";
6    }
7  }
8
9  attributes infact_uart_RxTx{
10   include += "#include \"uart_RxTx.h\"";
11 }
```

Listing 4.32: Modifications to rules file when used for C code.

4.4.2.2 Generated Graph

The generated graph, by the PS code for UVM, is equal to the graph generated from the imported sequences. However, the graph generated by the PS code for the C code, is different as some **meta_actions** are removed, and an action is added. This is demonstrated in Figure 4.7. After *init* will the **strcut** be traversed, where values for the **meta_actions** will be set followed by a call to the *do_RxTx* **action**, which will call the function with the selected **meta_actions** values.

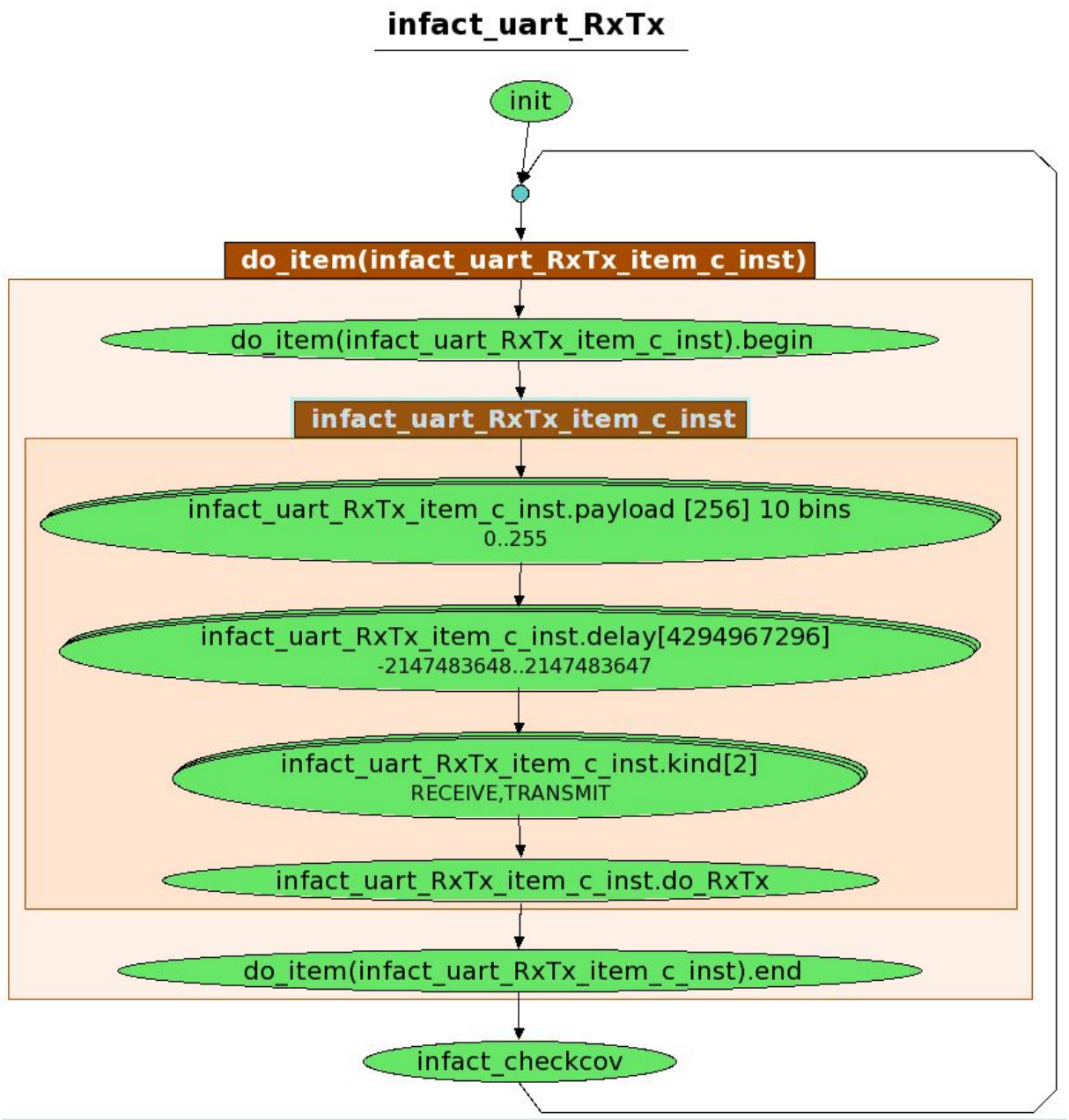

Figure 4.7: Graph generated from C code **rule_graph**.

4.4.2.3 Creating a Coverage Strategy

The bin for the *payload* is reused from the imported test components. However, this time, cross coverage is of more interest. This is because we want the payload values to be matched with both *RECEIVE* and *TRANSMIT*. This is done by specifying a coverage strategy and selecting **cross coverage** from the *payload* node, and setting the *kind* node as stop node. Because only these two values are of interest, the other nodes in between are set as ”don’t care”. Thus, excluded. This is done for both test components, and the graph for the C code test component, containing the cross coverage, can be seen in Figure 4.8. As a result, the graph will be traversed 20 times, 10 times for each **kind**.

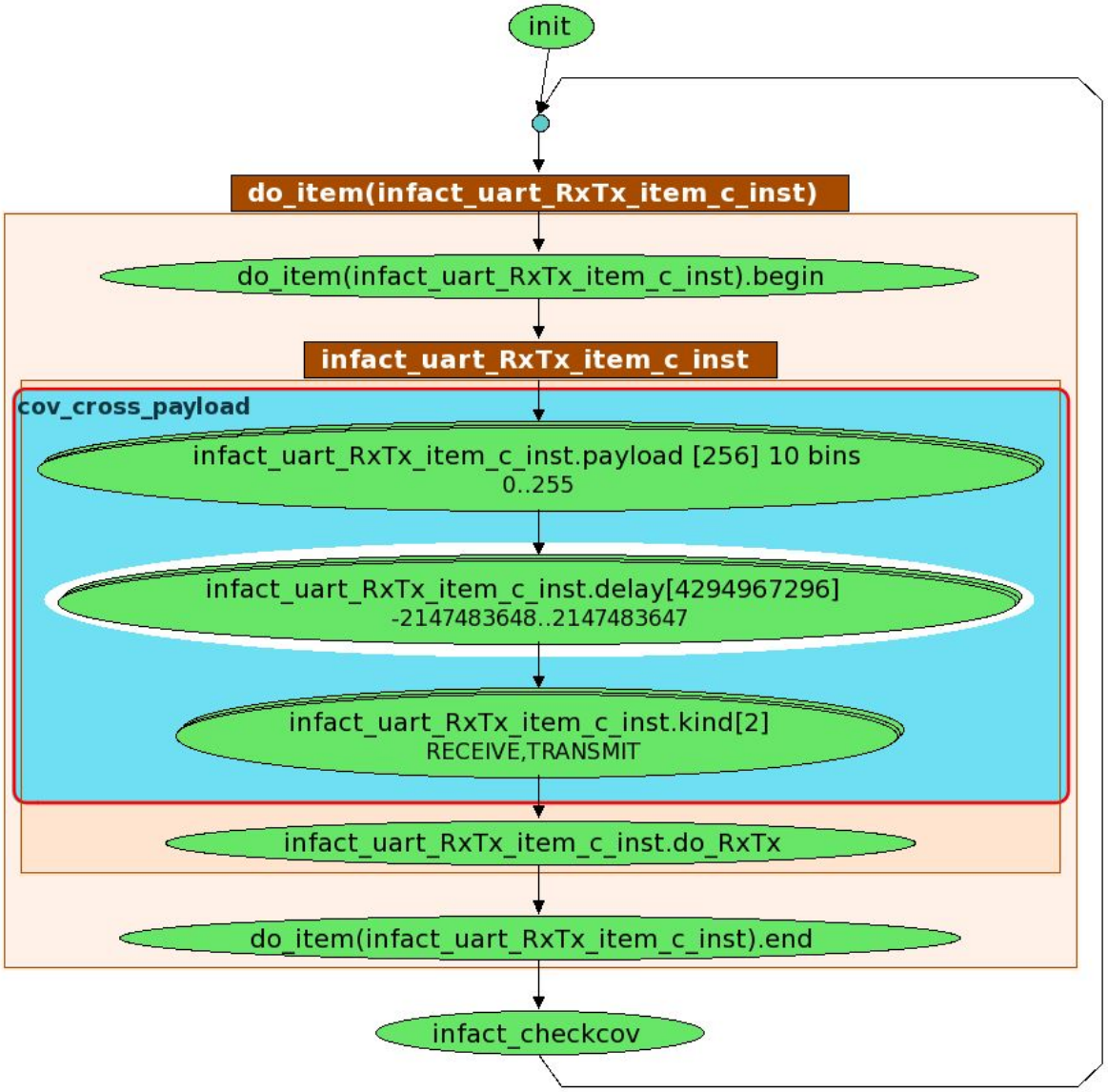

Figure 4.8: Graph generated from C code **rule_graph**.

4.5 UVM Simulation with Generated Code

There are two methods for using a inFact sequence in a simulation. One method is to instantiate and call it directly in a testbench. Another method is to override an existing sequence with the inFact sequence by adding this line to a compilation/simulation file: *+uvm_set_type_override=original_sequence_name, infact_sequence_name*. Also, the component files, including the sequence, needs to be included in a package, and the path to the component files should be specified in the compilation file through *+incdir+path_to_test_component*. The inFact project consists of a *.ini*-file, that is used to specify settings for inFact to use during the simulation session. This file needs to be passed to inFact at runtime via the simulator, which is done by adding *+infact=path_to_ini_file* to the simulation file. The component files are included in the `UART` package as seen in Listing 4.33. Their paths are specified in the compilation file. The *.ini*-file is added in the simulation file.

```
1  //Added to package
2  `include "infact_uart_receive_transaction_gen.svh"
3  `include "infact_uart_transmit_transaction_gen.svh"
4  `include "infact_uart_receive_delay_transaction_gen.svh"
5
6  //Added to compialtion file
7  +incdir+${VC_WORKSPACE}/ip/Uart/uvm/infact_uart/uart_infact/
      infact_uart_receive_transaction_gen
8  +incdir+${VC_WORKSPACE}/ip/Uart/uvm/infact_uart/uart_infact/
      infact_uart_transmit_transaction_gen
9  +incdir+${VC_WORKSPACE}/ip/Uart/uvm/infact_uart/uart_infact/
      infact_uart_receive_delay_transaction_gen
10
11 //Added to simulation file
12 +infact=${VC_WORKSPACE}/ip/Uart/uvm/infact_uart/uart_infact/
      uart_infact.ini
```

Listing 4.33: Modifications for inFact in UVM testbench, and compilation and simulation files

4.5.1 UVM testbench Modifications

The UVM testbenches are expanded to run three different sets of sequences, ”normal” sequences, the imported sequences from inFact, and the inFact sequence created from scratch. The selection is based on defines, where default is the normal sequences, *INFACT_IMPORT* for the inFact imported sequences, and *INFACT_RXTX* for the inFact sequence from scratch. The first two use the same sequence names and the same tasks, only which sequence is different. The last use a separate sequence definition and name, and calls a separate task. Listing 4.34 demonstrates this for the sub-system, but the same method is used in all three testbenches. The IP level testbench does not call the inFact sequence from scratch. A *RUN_ALL* script is modified to run all three scenarios by adding the defines.

```
1  `ifdef INFACT_IMPORT
2    infact_uart_receive_transaction_gen PeripheralSS_rec_seq;
3    infact_uart_transmit_transaction_gen PeripheralSS_trans_seq;
4    infact_uart_receive_delay_transaction_gen
        PeripheralSS_rec_delay_seq0;
5    infact_uart_receive_delay_transaction_gen
        PeripheralSS_rec_delay_seq1;
6  `else
7    uart_sequence_receive              PeripheralSS_rec_seq;
8    uart_sequence_transmit             PeripheralSS_trans_seq;
9    uart_sequence_receive_fixed_delay PeripheralSS_rec_delay_seq0
        ;
10   uart_sequence_receive_fixed_delay PeripheralSS_rec_delay_seq1
        ;
11 `endif
12 // Mixed Rx and Tx sequence
```

```
13 infact_RxTx_seq PeripheralSubSystem_RxTx_seq;
14
15 task run_phase(uvm_phase phase);
16   ...
17   `ifdef INFACT_RXTX
18     ta_ReceiveAndTransmit();
19   `else
20     ta_Receive();
21     #10us;
22     ta_Transmit();
23     #10us;
24     ta_ReceiveDmaPriority();
25   `endif
```

Listing 4.34: Modification done to sub-system UVM testbench to switch between sequences. (Equvivalent modifications is done to the other two testbenches)

4.5.2 Simulation Results

During a simulation, a coverage report is printed by inFact each time a sequence is called, stating the number of times being called out of total goal. An example is shown in Listing 4.35. This also states which sequence is being called, and through which sequencer. All the three testbenches were completed with success with all the different defines. And of course, all the coverage bins are hit with the inFact simulation, and only once.

```
1 ------------------------------------------------
2 - infact_uart_transmit_transaction_gen_cov Coverage Report: "
     uvm_test_top.env.uart0.agent.sequencer._item."
3 ------------------------------------------------
4 Path Coverage uvm_test_top.env.uart0.agent.sequencer._item.
     cov_payload_trans: 1/10 paths covered
```

Listing 4.35: Goal status printed by inFact during simualtion.

4.6 C Simulation with Generated Code

The inFact generated C code can be integrated in two ways, either as a dynamic test or as a static test. In a dynamic test the inFact sequence communicates with software running on the processor via shared memory, to execute software functions during simulation. In a static test, a static code has specified the order the processor executes the software functions during simulation. The latter is desired for this C code, as it is not wanted to use inFact during simulation. The static test contains selected values for the **meta_actions**, meaning the C code framework, together with the inFact code, can be compiled together, and simulated without using inFact. Static testing is fairly new, so it is not yet supported by the inFact IDE. Therefore, the static C code is generated through a bash command, in this case "*infact cmd sdv_static_gen -data workspace -max_iterations 30 \infact_uart_RxTx/infact_uart_RxTx.tmd*". This specifies a static test to be generated for the *infact_uart_RxTx* test component, with a maximum iteration of 30,

which means stop at 30 iterations, or when coverage goal is met. The static code contains functions for selecting values and sending them to the other C generated code. Only the inFact sequence created from scratch is tested with C code.

4.6.1 C Code Modifications

The inFact generated C code from the **rule_graph**, and the static C code, can be integrated to the C code framework by including their header files. Next, they can be linked in the main function as seen in Listing 4.36. An inFact test-engine manager is instantiated and created, and the inFact sequence is instantiated and created with the test-engine manger. This way the static C code will provide **meta_action** values to the inFact sequence. The generated C code is run by adding "*infact_uart_RxTx_run(seq)*" between starting RX and starting TX functions. Also, the inFact SDV library are linked in the compile statement.

```
1   //Instantiate test engine manager and inFact sequence
2   infact_sdv_te_mgr te_mgr;
3   infact_uart_RxTx_t *seq;
4
5   //Create test engine manager with static C code
6   infact_uart_RxTx_static_init(&te_mgr);
7   //Create inFact C code sequence
8   seq = infact_uart_RxTx_create(&te_mgr, "seq");
9
10  ...
11
12  //Run inFact sequence
13  infact_uart_RxTx_run(seq);
```

Listing 4.36: Modification to C code ti run inFact sequence.

4.6.2 Simulation Results

The compiled *.axf.vhx.32* file is loaded as the FW in the SystemVerilog testbench. The simulation completes without errors. The behavior is not monitored by the testbench, nor the FW in any way, but the waveform is examined, and shows correct behavior. Usually when writing C code, it is not normal to add any monitor functions. In addition, this is to check if stimulus is portable, and not to check correctness of SoC. Thus, it was reasoned to not include any monitoring.

CHAPTER 5

Discussion

5.1 Portable Stimulus

Portable Stimulus brings a new way of thinking, with a new high-level representation of a system, expressed with a new language, thus, leading to a change in verification. PSS describe the system in an abstract way with a high-level language. This might be unfamiliar to a verification engineer working with hardware, as they usually work at a low-level. With PSS, they must alter their way of thinking. Change is not always easy, thus getting used to PSS can be challenging. On the other hand, architects and software engineers are more used to working on a higher level of abstraction, so they will more easily adjust to PSS. Nevertheless, PSS might be more beneficial for hardware engineers as this is where most of the verification is done, as they deal with a lot of different levels. Also, the language is recognizable for them as it reminds of SystemVerilog.

One of the main ideas of PS is to move the verification from thinking of how to test it, to what to test. This involves specifying what stimulus should be tested and how they relate to each other. As a result, the engineers are forced to think about which values and combinations should be tested. It is possible to forget test scenarios when the focus lies on how to do it. For example, when writing a test verifying an operation, a corner case may be forgotten. Or, when using random generation, maybe not all ranges get tested. Regardless, this might only concern a few cases as it is also important today to check that everything is tested. However, another benefit is that it is easier to get an overview of what is tested because it is all in one place. As a result, it is easier to see what is missing, and check that everything is covered. Shifting of this focus will require changes in verification, which might take time and become difficult in the beginning. Nevertheless, the end-result will provide these benefits, which can be in favour of PS.

There does not exist any other standardized method for making stimulus portable between platforms, mainly from SystemVerilog to C code in this thesis. However, the usability of this is doubtful. The verification target is different, as SystemVerilog simulation is probes based looking for specific signal behavior, while C code looks at black boxed functional checks using instructions. On the other hand, they have similarities as well, because both at SoC want to

verify that everything has been integrated the correct way. For example, the transmission is equivalent in the SoC UVM testbench and the C code. The RAM is written, and transmission is started. Nevertheless, the scenarios for the `UART` were created with regard to a IP point of view. The first part was to make sure the received bits and the receive buffer is equivalent, and that the transmit buffer and the transmitted bits are equivalent. Then, focus was shifted to comparing these bits with RAM values for the sub-system and SoC. This shifts the view from looking at one block to multiple blocks communicating correctly. How this is achieved in SystemVerilog and in software is different because of change in focus between specific signals and instructions. This creates a gap in which scenarios can be applied. Regardless, there are scenarios that is relevant for both targets. Selecting the correct scenarios on the different abstraction levels are the key to creating test intent for both worlds.

An example where the portability between UVM and software verification is restricted is the receive-test. It is desired to verify that the SoC can function correctly, both in UVM and software. However, in software, it is strange to program a device to receive using pin loop back. In the real world information received would come from an external device. Thus, this should have been the intended test stimulus and test harness. At the SoC level in RTL, it makes sense to test receiving without an external device because the testbench harness provide necessary external hook up or behavioral model when applying the receive information. Therefore, the same scenario as at IP and sub-system level also applies here. Regardless, the workaround used in this thesis to get the software to receive by a pin loop back does fulfill the receive verification. It achieves the desired goal and proves the point of portable stimulus, even though it is in a strange way, but as the saying goes, ”if it looks stupid, but works, it ain’t stupid”.

5.2 Portable Stimulus and inFact

The name ”Portable Stimulus” states that the stimulus is portable. It does not state ”Portable Tests”, that test cases should be portable. In that sense, inFact does what it promises. The stimulus is reused at the different abstraction levels, IP, sub-system and SoC with UVM testbenches, as well as with C code. The portability requires little effort, as the same PS code can be used between the UVM testbenches, while the PS code for the C code requires small adjustments. Even though only the inFact PS code, that were written from scratch, was used between two different platforms, the imported code could just as well have been adjusted for C code. However, this was not necessary as the point was a proof of concept. Portability between different abstraction layers as well as different platforms were proven by working examples.

It is difficult to say which of the three available tools supports most of the PS standard. However, based on reading their descriptions, they support various parts, but it looks like inFact supports the most. In addition, inFact has been around the longest. Nevertheless, at this stage, PS from inFact can to some extent be seen as an extended sequence generator. In a sense, what it does is to select values for a list of specified variables and send it to a testbench. On the other hand, the documentation on PSS states that it creates UVM sequences, like inFact does now. PS does create UVM sequences, but then again, it is not difficult to create a sequence manually, and then reuse it on different abstraction levels. This thesis demonstrates this.

The benefit of PSS from inFact is a bit questionable, ”why bother when it can more easily be done manually?”. However, there are some counter arguments. inFact is created for efficient verification. Even though the efficiency gained from porting the stimulus, rather than doing it manually, is limited, the value selection for the stimulus is beneficial. Especially the simulation time can be more efficient than ”normal” random verification. inFact provides coverage driven stimulus. This means that every coverage bin, unless otherwise specified, will be hit once, i.e. you can be sure everything is tested, and time is not spent covering it multiple times. Comparing this with a huge test, where either a lot of iteration is performed, or passive coverage driven stimulus is used (waiting for all bins to be randomly hit), inFact is a lot more efficient. In addition, you can easily make sure this is the case on all the abstraction levels with its portability. The generated stimulus will cover all the verification goals. As a result, inFact can be considered a method for efficient verification, which is one of the main goals of PSS. This also helps bug detection when everything is exercised. As mentioned, writing a UVM sequence is easy. However, writing a UVM sequence with coverage goals is more difficult. Based on this, the PS code from inFact provides verification value. Another benefit is that the coverage driven verification by specifying bins and action and path coverage applies the new thinking in PSS of what to test.

5.3 Usability of inFact

Writing rules files in inFact is fairly easy. Most of the rules files are automatically generated. What is left is to know which inputs the DUT has, and create them as meta actions. Then, they can be constrained to dismiss illegal values or select specific values. After this, it is ready to go, and integrating the test component is simple. It is also easy to reuse a rules file between different test components, only small adjustments are necessary. Also, as mentioned in the Theory and Background chapter, PSS and SystemVerilog have a lot of reuse possibilities. This is greatly supported by inFact and its import tool. Only a transaction is required, importing a sequence is optional. The transaction has to be written anyway, as this have to be registered with the rest of the testbench. Thus, it requires almost no effort to import the transaction and create sequences from it. This supports the argument that it is worth the trouble to use inFact to create UVM sequences.

The C code on the other hand was a bit difficult. A reason for this is that inFact does not include a functionality to directly generate ”clean” C code. There are multiple options to create C code, either through a SystemC test component or an SDV UVM sequence test component. The first was tried out, but the generated code are in C++ and classes are a huge part of it. Therefore, a copy of the C code was created in C++. However, it was impossible to use as the Keil Arm compiler did not support *pthreads*. This is required as the generated code are SystemC code that must be adopted into C++ code. Also, it was not easy to understand how this would be connected to the C++ code. It seems like a component need to be instantiated and invoke a fill function which returns all the meta actions, which then must be distributed to the correct corresponding values in the C++ code. Therefore, the SystemC solution was discarded, and the SDV solution preferred.

The SDV test component is not perfect. For this use case, the dynamic test was not used as this required inFact to be run during simulation, in addition to having no need for a shared memory. On the other hand. the static test works well in the C code. The test component produced C code, including the static test, which could be used without inFact during simulation. Only an SDV library was required from inFact when compiling. In addition, the method by having the generated code call a function in the C code is better than the SystemC solution as it is simpler and does not clutter up the code. All the information on how to integrate the generated code is documented. Nevertheless, it is a bit scattered and could have been better explained as it is not easy to understand each step when doing it the first time. It would have been nice to have a more thorough explanation on how to do it, but also why the steps are necessary. With that being said, after having done this one time, it will be no problem to do it again.

Creating and using an action to call the C code was easy. There were some problems in understanding how this worked in the beginning, but it was not difficult to expand the rules file using such. However, there were some trouble with using **attributes**. It worked when using it to specify the C function the action should point to, but using to add a include statement with the header file containing the implementation did not work. Having tried different methods, but the *+include* inside the **attribute** for the **rule_graph** would not work. Even though it was done exactly as the documentation specifies. However, the reason might have been that the header file was not located inside the test component workspace, but I do not see how this should be necessary as it is about adding a line of information. Regardless, this was no big deal as the include statement was easy to add manually. Of course, this disappeared when regenerating the test component, but it required little effort to put it back.

5.4 Future of inFact

inFact focuses only on inputs, while the PSS specification cares about both. For example, the PSS specification states to specify flow objects, which are inputs and outputs to actions. This demonstrates how infact is only a subset of PSS. If the outputs were to be specified too, the potential for generated code will grow. When both inputs and outputs are specified, there is nothing that separates the transaction in UVM and the **struct** in inFact. As a result, both the transactions and sequences can be generated. Also, the sequencer might as well be generated. The sequencer usually has a basic structure containing a constructor. Nevertheless, if someone needs a more complex sequencer it could either be by more configuration in inFact, or it can be extended with complex behavior in the UVM testbench.

As mentioned, inFact is limited when it comes to the PSS description. Regardless, inFact is a large contributor to the upcoming PSS, and may in the future support everything. The largest difference lies in not specifying the entire system. When analyzing the PSS description, it seems like the idea is to specify how each block communicates. Especially resources and pools are a large contributor to this, as well as specifying inputs and outputs in flow objects. Compared to this, the current PSS version in inFact is effortless to use. But then again, the generated code from inFact is limited. With the complete PSS, it would be expected that the generated code is greater. Maybe, not only is the stimulus generated, but also how to apply it in some way. If PSS becomes better, the productivity and quality will increase. This is in the form of more through explanation on what to test and getting better generated code, but also through faster simulation.

Nordic Semiconductor should adopt UVM to utilize inFact. PSS is established with regards to UVM, which has become a industry standard. As inFact begins to support more and more of PSS, the development towards UVM will increase and will have the highest yield. An alternative would be to move the input stimulus to classes in order to benefit from inFact. However, this would only allow for coverage driven simulation, and not the same level of benefit from reuse.

5.5 The Proof of Concept

The thesis goal was making a proof of concept to see what is possible with PSS. However, the tests created are not advanced. No more than the basic operations of the `UART` is tested, even though more functionality is supported. For example, all the frame formats as well as error frames could have been tested. Regardless, this would have taken a lot more time. Also, the goal was to use PSS at all different stages, therefore a more complex verification scenarios for completeness were sacrificed. However, a benefit from having created a more complex test intent would be that it will more extensively showcase what inFact can do. The graph for example is directed, while more complex structures are possible such as branches.

The `PAR` environment at IP level is unnecessary. It is never used any other place, and not used with inFact. The other levels use a different task for IO access, which uses the existing Nordic Semiconductor framework utilities. However, the reason for having the `PAR` environment was that it was a part of learning UVM. Also, there is no reason to remove it when it is already there.

CHAPTER 6

Conclusion

Portable Stimulus has a lot of potential, describe test intent once, and reuse this within multiple abstraction levels and platforms, thus increase both quality and productivity in verification efforts. However, the question is how much of a complete test will PSS generate. It is called Portable Stimulus, which might suggest only the stimulus is generated. Nevertheless, based on the PSS specification, presentations and interviews by the working group, it sounds like PSS also should generate how to apply it to some degree. This can be argued based on the possibility to describe the entire system. When the PSS specification is completed and fully supported by tools, we will know for sure.

Not everything of PSS is supported by inFact during this thesis, only a subset. This involves stimulus that is portable. Applying the generated stimulus needs to be done manually. The coverage driven model is a huge asset to employ PSS by making the verification engineer think about what to test. This is already a part of verification today, but with PSS it will have a larger role. In inFact one thinks about what to test when writing coverage bins and specifying action and path coverage. As a result, it might become easier to get an overview of what is verified and what is missing when making sure everything is covered. Also, the simulation time will decrease with inFact's choice of stimulus values, as nothing gets verified more than once, making the verification more efficient.

The created UVM and C code frameworks are proven to work with inFact's PSS. Based on a single specification, the same stimulus is applied to three different abstraction layers, IP-, subsystem and SoC-level, as well as between different platforms with the UVM simulation and the C code. It is not difficult to write the rules files, as they are mostly automatically generated, and the information needed already lies in the transaction. The import functionality is an asset, as you can create the complete rules file automatically though a transaction that must be created anyway. However, the C code generation is a bit tricky, but after having learned how to do it, it will not be a problem.

Based on the results, it can be said that PSS is heading in the right direction, but it still has a long way to go. A goal with PSS is to increase productivity and quality of verification, which inFact does achieve. This is achieved through lowering the simulation time and having to describe what we want to test. If Nordic Semiconductor wants to utilize the benefits from inFact and PSS, they should adopt UVM. UVM is the standard methodology adopted within industry, in addition to PSS creating UVM. However, there are methods for generating stimulus for regular classes as well, but then they must alter their verification to get stimulus from classes.

6.1 Future Work

When the PSS specification is complete and inFact supports everything, a new proof of concept should be conducted. The frameworks are in place, making it easier to examine and analyze the benefits of PSS. However, they may have to be altered and become more complex, but this thesis provides a starting point.

Bibliography

[1] Universal verification methodology (uvm) working group. http://accellera.org/activities/working-groups/uvm/.

[2] UVM Working Group. *Universal Verification Methodology (UVM) 1.2 Class Reference*. Accellera, June 2014. http://accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf.

[3] UVM Working Group. *Universal Verification Methodology (UVM) 1.2 Users Guide*. Accellera, October 2015. http://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[4] Defining the verification environment. https://colorlesscube.com/uvm-guide-for-beginners/chapter-2-defining-the-verification-environment/. Accessed: 11.03.18.

[5] Accellera PSWG. Pss dvcon us tutorial. In *DVCon US*, 2017.

[6] Matthew Ballance. Automating test from ip to soc levels with portable stimulus. http://www.techdesignforums.com/practice/technique/automating-test-from-ip-to-soc-levels-with-portable-stimulus/. Accessed: 22.01.18.

[7] Gabe Moretti. Portable stimulus: The making of a standard. *Accellera*, 2017. http://www.accellera.org/resources/articles/portable-stimulus-the-making-of-a-standard.

[8] Gabe Moretti. Developing the portable stimulus standard. *Chip Design*, 2017. http://eecatalog.com/chipdesign/2017/10/17/developing-the-portable-stimulus-standard/.

[9] Automating tests with portable stimulus from ip to soc level. https://www.mentor.com/products/fv/resources/overview/automating-tests-with-portable-stimulus-from-ip-to-soc-level-79635dab-61df-47c3-99fd-0411e11c934b. Accessed: 29.01.18.

[10] Portable stimulus. http://www.techdesignforums.com/practice/guides/portable-stimulus/. Accessed: 22.01.18.

[11] Portable stimulus specification working group. http://accellera.org/activities/working-groups/portable-stimulus.

[12] Press releases. http://accellera.org/news/press-releases/244-accellera-portable-stimulus-early-adopter-specification-now-available-for-public-review.

[13] Matthew Ballance. *Making Legacy Portable with the Portable Stimulus Specification*. Mentor Graphics Corp., 2017.

[14] Brian Bailey. Portable stimulus status report. *SEMICONDUCTOR ENGINEERING*, 2017. https://semiengineering.com/portable-stimulus-status/.

[15] Portable Stimulus Specification Working Group. *PSS Early Adopter (EA) Portable Test and Stimulus Standard*. Accellera, July 2017. http://www.accellera.org/images/downloads/drafts-review/PSS_Early_Adopter_Release.pdf.

[16] Portable Stimulus Specification Working Group. *PSS Early Adopter II (PSS EA II) Portable Test and Stimulus Standard*. Accellera, February 2018. http://www.accellera.org/images/downloads/drafts-review/PSS_EAII_Feb_28_2018_Public_Review.pdf.

[17] Boosting test creation productivity with portable stimulus. https://www.mentor.com/products/fv/resources/overview/boosting-test-creation-productivity-with-portable-stimulus-2305bbf4-81cb-493f-95ba-93a2726e26fa.

[18] Mentor verification is first to deliver portable stimulus technology across the full enterprise verification platform. https://www.mentor.com/company/news/siemens-mentor-verification-first-to-deliver-portable-stimulus-tech-across-full-enterprise-verification-platform.

[19] Questa infact, intelligent testbench automation. https://www.mentor.com/products/fv/infact/.

[20] Breker verification systems to demonstrate implementation compliant with accellera portable stimulus draft standard during dac. http://www.marketwired.com/press-release/breker-verification-systems-demonstrate-implementation-compliant-with-accellera-portable-2222164.htm.

[21] Perspec system verifier is #1 in portable stimulus in 2017 user survey. https://community.cadence.com/cadence_blogs_8/b/sd/posts/perspec-system-verifier-is-1-in-portable-stimulus-in-2017-user-survey?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+cadence%2Fcommunity%2Fblogs%2Fsd+%28Cadence+System+Design+and+Verification+Blogs%29. Accessed: 04.02.18.

[22] Mentor Graphics Corporation. *Questa inFact User's Manual*, 10.7a edition, 2013-2018.

[23] Mentor Graphics Corp. *Portable Stimulus Specification and inFact*, Presentation. Staffan Berg, 2018.

[24] Mentor Graphics Corp. *Questa inFact Fundamentals*, Presentation. Staffan Berg, 2018.

[25] Mentor Graphics Corporation. *Questa inFact Reference Manual*, 10.7a edition, 2018.

APPENDIX A

Source Files

The Appendix is online and can be located here: `https://github.com/kariannekk/Portable-Stimulus`

The folders in this appendix include the following content:

- UVM_Uart - All the testbench files for the IP level UVM testbench
- UVM_PeripheralSS - All the testbench files for the sub-system level UVM testbench
- UVM_UartTop - All the testbench files for the SoC level UVM testbench
- FW_Uart - The C code and SystemVerilog testbench running it
- Infact code - The rules files for all 5 test components