Identification Of Validation Collaterals Using Architecture Specification

Submitted By

Jagruti Jamba

16MCEC05

DEPARTMENT OF COMPUTER ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2018

Identification Of Validation Collaterals Using Architecture Specification

Major Project

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology in Computer Science and Engineering

Submitted By

Jagruti Jamba

(16MCEC05)

Guided By

Prof. Rupal Kapdi

DEPARTMENT OF COMPUTER ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2018

Certificate

This is to certify that the major project entitled **"Identification Of Validation Collaterals Using Architecture Specification"** submitted by **Jagruti Jamba (16MCEC05)**, towards the partial fulfillment of the requirements for the award of degree of Master of Technology in Computer Science and Engineering (Specialization in title case, if applicable) of Nirma University, Ahmedabad, is the record of work carried out by her under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project part-I, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Rupal Kapdi
Guide & Assistant Professor,
CE / IT Department,
Institute of Technology,
Nirma University, Ahmedabad.

Dr. Priyanka Sharma
Professor,
Coordinator M.Tech - CSE (Specialization)
Institute of Technology,
Nirma University, Ahmedabad

Dr. Sanjay Garg
Professor and Head,
CE Department,
Institute of Technology,
Nirma University, Ahmedabad.

Dr. Alka Mahajan
Director,
Institute of Technology,
Nirma University, Ahmedabad

Statement of Originality

I, **Jagruti Jamba**, **16MCEC05**, give undertaking that the Major Project entitled "**Identification Of Validation Collaterals in Architecture Specification**" submitted by me, towards the partial fulfillment of the requirements for the degree of Master of Technology in **Computer Science & Engineering** of Institute of Technology, Nirma University, Ahmedabad, contains no material that has been awarded for any degree or diploma in any university or school in any territory to the best of my knowledge. It is the original work carried out by me and I give assurance that no attempt of plagiarism has been made.It contains no material that is previously published or written, except where reference has been made. I understand that in the event of any similarity found subsequently with any published work or any dissertation work elsewhere; it will result in severe disciplinary action.

Signature of Student
Date:
Place:

Endorsed by
Prof. Rupal Kapdi
(Signature of Guide)

Acknowledgements

It gives me immense pleasure in expressing thanks and profound gratitude to **Prof. Rupal Kapdi**, Associate Professor, Computer Engineering Department, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout this work. The appreciation and continual support she has imparted has been a great motivation to me in reaching a higher goal. Her guidance has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come.

It gives me an immense pleasure to thank **Dr. Sanjay Garg**, Hon'ble Head of Computer Engineering Department, Institute of Technology, Nirma University, Ahmedabad for his kind support and providing basic infrastructure and healthy research environment.

A special thank you is expressed wholeheartedly to **Dr. Alka Mahajan**, Hon'ble Director, Institute of Technology, Nirma University, Ahmedabad for the unmentionable motivation he has extended throughout course of this work.

I would also thank the Institution, all faculty members of Computer Engineering Department, Nirma University, Ahmedabad for their special attention and suggestions towards the project work.

- Jagruti Jamba
16MCEC05

Abstract

Enhancements in technology, to some extent, have empowered the progress of multi-million gate designs from very large printed circuit sheets to small-sized processor chip. With the consistently expanding complexities and contracting geometries, the difficulties with processor configuration have developed significantly. The expanded complexities of processors have provoked the need of increase in design, functional and performance verification efforts to meet the client and time-to-market demands. Thus, the traditional verification techniques no longer satisfy the needs of verifying complex architecture as a whole. Hence, it is necessary to upgrade the techniques for effectively carrying out the verification so that it can cover all the corner cases and analyse them if the given architecture design or instruction set behaves as per the expectations.

AVS is widely used test suite for carrying out verification of the ARM architecture. The grounds of verification or the parameters serve as the basis for carrying out any verification or validation. This report will be describing the steps to be performed prior to the verification process and also the motivation behind it. It also describes the verification technique that is conducted in ARM.

Abbreviations

A32	ARM Instruction Set
AVS	Architecture Validation Suites
DPLL	Davis-Putnam-Logemann-Loveland Procedure
ELR	Exception Link Register
FP	Floating Point
ISS	Instruction Set Simulator
PC	Program Counter
PE	Processing Element
PSTATE	Program State
RISC	Reduced Instruction Set Computer
SAT	Satisfiability
SIMD	Single Instruction Multiple Data
SMT	Satisfiability Modulo Theroies
SP	Stack Pointer
T32	Thumb Instruction Set

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The major issues with the processor are 1. scalability and 2. The complicated ISA (Instruction Set Architecture) with increased number of instructions to perform different kinds of computations. This in turn leads to complicated Verification Process which requires effective verification techniques and also the organized data-set to be fed into the various verification tools. AVS is one of the tools used at ARM for verification of ARM Architecture. We need to verify if ARM cores of same architecture version are software compatible with each other. Cores from Architecture licensees are s/w compatible with all other cores (ARM and others) of the same Architecture version. AVS is one such tool we have been using to meet our requirements. But there are some steps that need to be performed before actually carrying out the verification process as verification process itself requires some set of data (dependency data set) on which it can rely for carrying out verification as a whole.

1.2 Objective

1. Dependency set identification

   - Identify the architectural states that affect the outcome (set of dependencies) of a particular feature For eg. traps.

   - This could be used as a query mechanism by architects to check if the pseudocode actually meets the architectural intent.

- Reviewers to easily gauge the quality of testing.
- Test-writers to identify vector sets to feed into test-generation aspects.

2. Checkpoint set identification

- Identify the architectural states(direct and indirect) to be checkpointed for efficient feature validation which could feed into different tools performing verification (can be either determininstic or random instruction simulation).
- Identification of pseudo states is the challenge.

3. Interesting data set Collection: that can feed into verification tools.

Cover the corner scenarios and the ones which are prone to be misinterpreted.

1.3 Thesis Organization

- Chapter 2 gives the Introduction to ARM Arhictecture.
- Chapter 3 describes the Scalable Vector Extension and how it differs from the NEON.
- Chapter 4 is the literature survey describing various verification techniques that are used in ARM.
- Chapter 5 describes the various approaches for conducting the verification process.
- Chapter 6 describes the overall flow of the verification process and my contribution in the beginning steps.
- Chapter 7 describes the approach followed for collecting interesting datasets
- Chapter 8 is all about the conclusion and the future scope.

Chapter 2

ARM Architecture

2.1 ARMv8A Architecture

ARM Architecture is a RISC architecture which defines the behaviour of an abstract object which can be referred to as PE. A PE is basically the smallest unit of the architecture.[1]

The features of ARM architecture include the load/store operations performed on registers to access the data rather than directly accessing the memory for the same. It also has very simple addressing modes wherein the all the addresses of load/store are determined from the instruction fields and registers contents only. The architecture defines the interaction of the PE with memory, including caches, and includes a memory translation system. It also describes how multiple PEs interact with each other and with other observers in a system. The ARM architecture supports implementations across a wide range of performance points. Implementation size, performance, and very low power consumption are key attributes of the ARM architecture.[1]
The ARMv8 architecture supports:

- A 64-bit Execution state, AArch64.
- A 32-bit Execution state, AArch32, that is compatible with previous versions of the ARM architecture.

Both Execution states support SIMD and floating-point instructions:

AArch32 state provides:

- SIMD instructions in the base instruction sets that operate on the 32-bit general-purpose registers.
- Advanced SIMD instructions that operate on registers in the SIMD and floating-point register (SIMDFP register) file.
- Floating-point instructions that operate on registers in the SIMD FP register file.

AArch64 state provides:

- Advanced SIMD instructions that operate on registers in the SIMD and floating-point register (SIMD FP register) file.
- Floating-point instructions that operate on registers in the SIMD FP register file.

2.2 Execution States

The Execution State defines the execution environment of a Processing Element which includes the size of the registers and the instructions.

The Execution states are:

AArch64 The 64-bit Execution state. This Execution state:

- Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
- Provides a 64-bit program counter (PC), stack pointers (SPs), and exception link registers(ELRs).
- Provides 32 128-bit registers for SIMD vector and scalar floating-point support.
- Provides a single instruction set, A64.
- Defines the ARMv8 Exception model, with up to four Exception levels, EL0 - EL3, that provide an execution privilege hierarchy.
- Provides support for 64-bit virtual addressing.

- Defines a number of Process state (PSTATE) elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.[1]
- Names each System register using a suffix that indicates the lowest Exception level at which the register can be accessed.

AArch32 The 32-bit Execution state. This Execution state:

- Provides 13 32-bit general-purpose registers, and a 32-bit PC, SP, and link register (LR). The LR is used as both an ELR and a procedure link register. Some of these registers have multiple banked instances for use in different PE modes.
- Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides two instruction sets, A32 and T32.
- Supports the ARMv7-A exception model, based on PE modes, and maps this onto the ARMv8 Exception model,that is based on the Exception levels.
- Provides support for 32-bit virtual addressing.

2.3 ARM Instruction Sets

In ARMv8 the possible instruction sets depend on the Execution state:
AArch64 - AArch64 state supports only a single instruction set, called A64. This is a fixed-length instruction set that uses 32-bit instruction encodings.
AArch32 - AArch32 state supports the following instruction sets:

- ***A32*** This is a fixed-length instruction set that uses 32-bit instruction encodings.
- ***T32*** This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings.

Chapter 3

Scalable Vector Extension

3.1 Introduction

Design expansions are regularly to some degree moderate when they are first presented and are then extended as their potential turns out to be better comprehended and transistor spending plans increment.

A few objectives guided the plan of the design. In the first place was the need to broaden the vector processing ability related with the ARM AArch64 execution state to better address the register necessities in areas for example, superior figuring (HPC), information investigation, PC vision and machine learning.

Second was the desire to introduce an extension that can scale across multiple implementations, both now and into the future, allowing Architecture designers to choose the vector length most suitable for their power, performance and area targets. The engineering ought to abstain from forcing a product improvement cost as the vector length changes and where conceivable lessen it by enhancing the range of compiler auto-vectorization advancements.

It permits implementations to pick a vector register length between 128 and 2048 bits. It supports a vector length rationalist programming model which enables code to run and scale consequently over all vector lengths without recompilation. At long last, it presents a few imaginative highlights that start to conquer a portion of the conventional boundaries to auto-vectorization.

These augmentations productively target media and picture processing workloads, which regularly process organized information utilizing very much adapted DSP calculations. Nonetheless, as our accomplices keep on deploying ARMv8-An into new markets we have seen an expanding interest for more radical changes to the ARM SIMD engineering, including the presentation of surely understood advancements, for example, gather-load and scatter-store, per-lane predication and longer vectors.

However, this brings up the issue, what should that vector length be? The decision from over a time of research into vector preparing, and taking motivation from more traditional vector structures, is that there is no single length vector length.

Thus, SVE leaves the vector length as an optional decision (from 128 to 2048 bits, in increases of 128 bits). Vitally the programming model modifies progressively to the accessible vector length, with no need to recompile abnormal state dialects or to modify hand-coded SVE get together or compiler intrinsics. At an high level, the key SVE highlights empowering enhanced auto-vectorization support are:

- Scalable vector length - expanding parallelism while permitting usage decision.
- Rich addressing modes - support for the access of non-linear data.
- Per-lane predication - permitting vectorization of loops containing complex control stream.
- Predicate-driven loop control reduces vectorization overhead with respect to scalar code.
- A rich arrangement of flat operations applicable to more sorts of reducible loop-carried conditions.
- Vector partitioning and programming oversaw speculation empowering vectorization of loops with information subordinate ways out.
- Scalarized intra-vector sub-loops allowing vectorization of loops with more intricate loop conveyed conditions.

3.2 SVE Architectural State

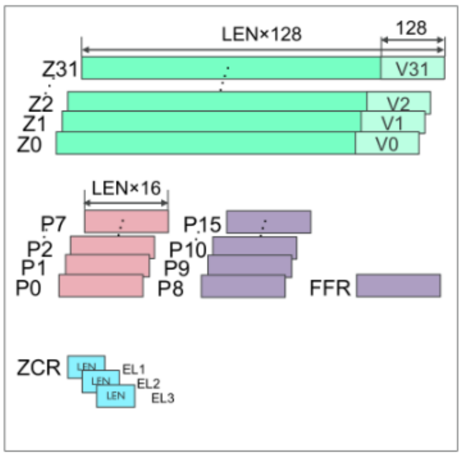
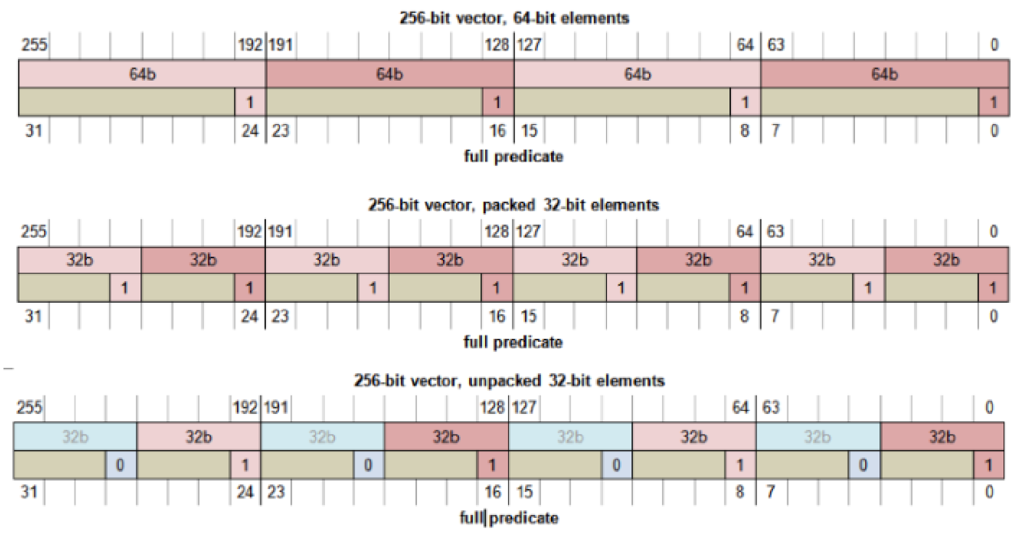


Figure 3.1: SVE Registers [2]

SVE presents new design state, appeared in Fig.2.1. This state gives thirty-two new adaptable vector registers (Z0 Z31). Their width is usage subordinate inside the previously mentioned territory. The new registers broaden the thirty-two 128-piece wide Advanced SIMD registers (V0 V31) to give adaptable holders to 64-, 32-, 16-, and 8-bit information components. Nearby the versatile vector registers, are sixteen scalable predicate registers (P0 P15) and a unique reason first-faulting register (FFR).



256-bit vector, 64-bit elements
255
192
191
128
127
64
63
0
64b
1
31
24
23
16
15
8
7
full predicate
256-bit vector, packed 32-bit elements
32b
256-bit vector, unpacked 32-bit elements

Figure 3.2: SVE predicate organization [2]

3.3 Scalable Vector Length

Within a fixed 32-bit encoding space, it isn't suitable to make an alternate direction set each time an alternate vector width is requested. SVE drastically leaves from this approach in being vector length skeptic, permitting every execution to pick a vector length that is any various of 128 bits in the vicinity of 128 and 2048 bits (the current design upper constrain). Not having a settled vector length enables SVE to address different markets with executions focusing on distinctive execution control territory enhancement focuses.

This novel part of SVE empowers programming to scale various vector lengths without the requirement for extra direction encodings, recompilation or programming porting exertion. SVE gives the capacities to programming to be vector length rationalist using vector partitioning while likewise supporting more traditional SIMD coding styles requiring settled length, different of-N or energy of-two sub-vectors.

3.4 Horizontal Operations

Another issue for conventional SIMD preparing is the nearness of conditions over various circle cycles. Much of the time, these conditions can be settled utilizing a straightforward even decrease operation. Not at all like ordinary SIMD directions, even operations are an exceptional class of directions that work over the components of the same vector enlist. SVE has a rich arrangement of even operations counting both sensible, whole number and floating point decreases as well as strictly-ordered reduction for floating-point.

3.5 Predication

Predicates are presented by an ordinary "if change" pass that replaces if-proclamations predicates and after that uses the proper predicate on each operation commanded by the control of the condition. This approach is reached out to deal with restrictive fans unaware of what's going on, by embeddings a 'brk' direction that produces a vector parcel where just those paths before the circle leave condition are dynamic.

3.6 Floating Point

Floating point is a test for compiler vectorization because vectorizing a lessening circle will change the request of floating point operations which may give an alternate outcome from the first scalar code. Software engineers at that point need to pick whether they need reliable outcomes by crippling vectorization or whether they can endure some variety to accomplish better execution. Vector length skepticism presents more variety because an alternate vector length could cause an alternate requesting and, in this manner, an alternate outcome. SVE mitigates this by giving 'fadda' that enables a compiler to vectorize those situations where the exact request of drifting point increases is basic to rightness.

3.7 SVE verus NEON

Throughout the years, extensive research has gone into deciding how best to extricate more information level parallelism from universally useful programming dialects, for example, C, C++ and Fortran. This has brought about the consideration of vectorization highlights, for example, scatter - store and gather load, per-lanepredication, and obviously longer vectors.

A key decision to make is the most fitting vector length, where many elements may impact the choice:

Current usage innovation and related power, execution and region tradeoffs.

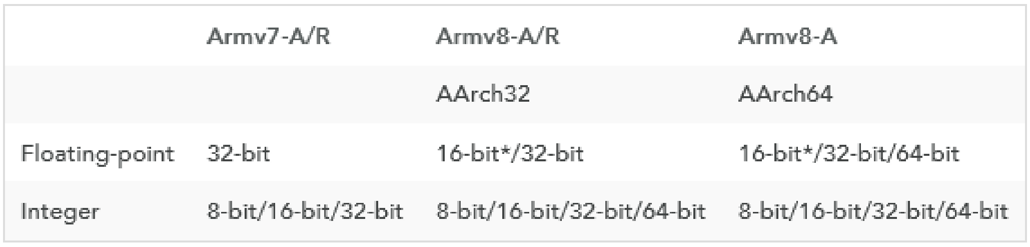
The particular application program attributes.

The market, which is HPC today; in the same manner as general patterns in PC design development, a developing requirement for longer vectors is normal in different markets later on.

3.7.1 NEON Instructions

The NEON technology is a packed SIMD architecture. NEON registers are considered as vectors of elements of the same data type. Multiple data types are supported by the technology. The following table describes data types as supported by the architecture version.[6]

The NEON instructions perform the same operations in all lanes of the vectors. The



	Armv7-A/R	Armv8-A/R	Armv8-A
		AArch32	AArch64
Floating-point	32-bit	16-bit*/32-bit	16-bit*/32-bit/64-bit
Integer	8-bit/16-bit/32-bit	8-bit/16-bit/32-bit/64-bit	8-bit/16-bit/32-bit/64-bit

Figure 3.3: Different Data Types supported By Different Architecture Versions[6]

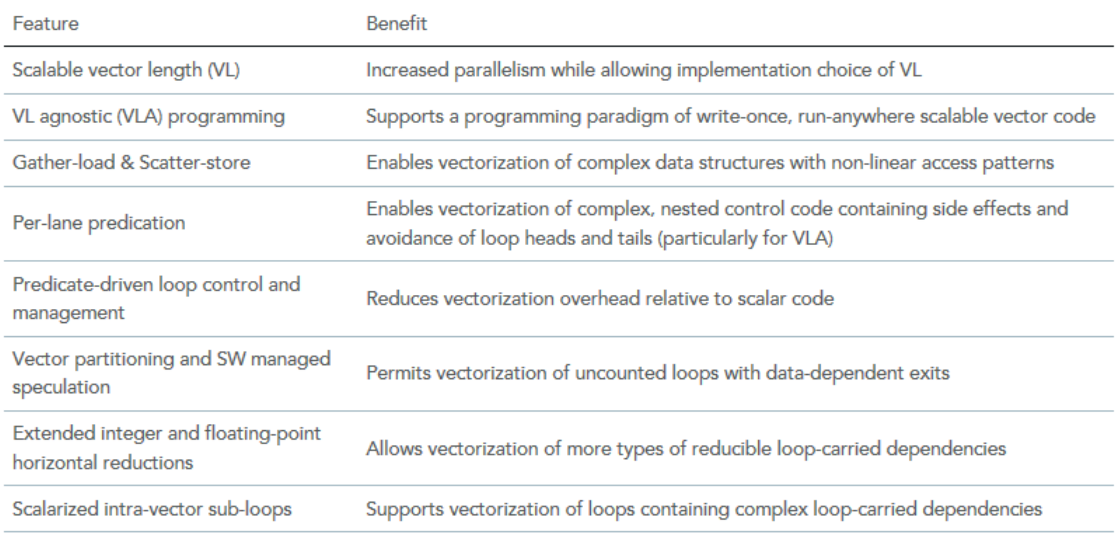
number of operations performed depends on the data types. NEON instructions allow up to: 16x8-bit, 8x16-bit, 4x32-bit, 2x64-bit integer operations 8x16-bit*, 4x32-bit, 2x64-bit** floating-point operations The implementation on NEON technology can also support issue of multiple instructions in parallel.

*Only in Armv8.2-A

**Only in Armv8-A/R [6]

3.7.2 How does SVE differ from Neon?

SVE is just an optional extension to NEON instructions which does not replace NEON, but provides the ability to vectorise the HPC workloads.



Feature	Benefit
Scalable vector length (VL)	Increased parallelism while allowing implementation choice of VL
VL agnostic (VLA) programming	Supports a programming paradigm of write-once, run-anywhere scalable vector code
Gather-load & Scatter-store	Enables vectorization of complex data structures with non-linear access patterns
Per-lane predication	Enables vectorization of complex, nested control code containing side effects and avoidance of loop heads and tails (particularly for VLA)
Predicate-driven loop control and management	Reduces vectorization overhead relative to scalar code
Vector partitioning and SW managed speculation	Permits vectorization of uncounted loops with data-dependent exits
Extended integer and floating-point horizontal reductions	Allows vectorization of more types of reducible loop-carried dependencies
Scalarized intra-vector sub-loops	Supports vectorization of loops containing complex loop-carried dependencies

Figure 3.4: SVE versus NEON [6]

Chapter 4

Arm Architecture Verification

4.1 What is Architecture Verification?

Functional verification of microprocessors is one of the most complex and expensive tasks in the current system-on-chip design methodology. Simulation using functional test vectors is the most widely used form of processor verification.[3] A major challenge in simulation-based verification is how to reduce the overall verification time and resources.[3]

Traditionally, billions of irregular and coordinated tests are utilized amid reenactment. Compared to random tests, directed tests can lessen general approval exertion fundamentally since shorter tests can get a similar scope objective. Nonetheless, there is an absence of computerized systems for directed test age focusing on small scale structural outline mistakes.

Verification is the way toward guaranteeing that the purpose of a plan is protected in its execution. Functional Verification can uncover practical rationale blunders in the equipment outlines which are depicted in behavioral model, enlist exchange level model, door level model, or switch level model. Useful mistakes are presented because of different elements including imprudent coding, error of the detail, microarchitectural plan unpredictability, corner cases. In the event that any utilitarian bug is found in a chip officially manufactured, the mistake should be revised and the changed adaptation of the outline should be manufactured once more, which is extremely costly. In the most noticeably awful case, bug settling after conveyance to clients will involve an expensive substitution and also re-creation costs.

In present day system-on-chip outlines, formal verification is one of the major bottlenecks because of the consolidated impacts of expanding outline unpredictability and diminishing time-to-advertise. Outline many-sided quality of current processors is expanding at a disturbing rate to adapt up to the required execution change for progressively perplexing applications in the areas of correspondence, sight and sound, systems administration and amusement. To suit such speedier calculation necessities, the present processors utilize numerous muddled small scale building highlights, for example, profound pipelines, dynamic booking, out-of-arrange and superscalar execution, and dynamic theory.

4.2 What is a Validation Collateral?

Verification of an Architecture can be performed on some ground parameters or data which serve as the base of verification process. Thus, Validation collaterals can be defined as the specifications or ground parameters or some data which can helpfully determine certain architecture states that are prone to easy misinterpretations. A validation collateral can be anything for example. a dependency data set, a checkpoint data set, some scenario sets or the data sets of interests.

4.3 Importance Of Validation Collaterals For Verification Engineer

Once a verification engineer has enough dataset to carry out the verification, the procedure can be continued very easily, since every verification process reqiures data of instructions in which each istruction can have certain corner cases, some psuedo states which can be misinterpretated, and some expected outcomes under certain scenarios. Thus, It is necessary to find out these relevant datasets as verification process cannot work actually without input dataset.

Chapter 5

Literature Survey

5.1 Verification Techniques

Existing processor verification methods are comprehensively arranged into formal procedures and reenactment based strategies The exchange off between formal procedures and reenactment based strategies is their ability and fulfillment in check. Formal confirmation methods give the culmination of check errand by demonstrating scientifically the rightness of a plan. Be that as it may, they experience issues in managing the vast plans because of the state space blast issue. Hypothesis demonstrating model checking SAT unraveling representative reproduction furthermore, equality checking are regularly utilized for formal confirmation of processor outlines.

Simulation based approval finds outline mistakes utilizing test vectors comprising of input jolts and expected yields. In spite of the fact that recreation based techniques can deal with complex processor outlines, they can't accomplish the fulfillment of check. For instance, for microchip check, all conceivable information direction successions are required keeping in mind the end goal to affirm the accuracy of a given chip outline. In any case, it is difficult to produce and mimic them in a sensible time. In this manner, formal strategies are more material to the confirmation of the little and basic parts, though simulation based strategies are more invaluable in approval of a predefined plan by giving a check.

The essential technique in simulation based processor approval comprises of creating

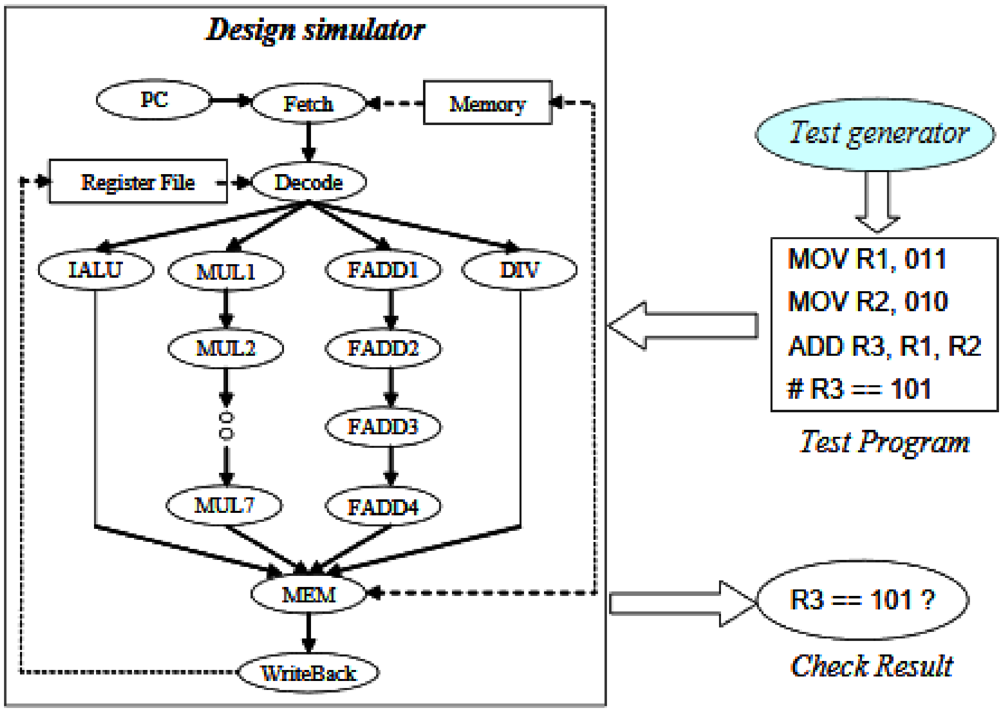


Figure 5.1: Simulation-based validation [4]

test programs, recreating a given processor plan with the test programs, contrasting the created yields with the normal outcomes, and adjusting plan mistakes if the simulation yields are unique in relation to the normal outcomes. A noteworthy test in processor approval is the manner by which to decrease the general approval time and assets. Since the test age and recreation for all information test programs is not feasible, a technique for choosing powerful tests is required to accomplish high certainty of the processor plan. In expansion, test age procedures must have the capacity to oblige complex processor outlines and additionally deliver tests in sensible time.

There are three sorts of test age methods: irregular, compelled arbitrary, also, co-ordinated. In the current modern practice , arbitrary and compelled irregular test age methods at engineering (ISA) level are most broadly utilized on the grounds that test projects can be delivered naturally and plan mistakes can be revealed ahead of schedule in the configuration cycle. Be that as it may, countless are required to accomplish high certainty of the outline rightness, and corner cases are barely noticeable. Moreover, building test age procedures experience issues in actuating small scale building target ancient rarities and pipeline functionalities since it isn't conceivable to create data with respect to pipeline communications or timing points of interest utilizing input ISA particular.

Compared to the arbitrary or compelled irregular tests, the coordinated tests can diminish general approval exertion essentially since shorter tests can get the same practical scope objective. Notwithstanding, there is an absence of robotized strategies for coordinated test age focusing on miniaturized scale engineering deficiencies. Accordingly, coordinated tests are regularly manually written by specialists. Because of manual advancement, it is infeasible to create all guided tests to accomplish extensive scope and this procedure is tedious and blunder inclined. Hence, there is a requirement for mechanized coordinated test age methods in light of miniaturized scale structural practical scope. Test age utilizing formal techniques has been effectively utilized because of its ability of programmed test age. Be that as it may, the customary test age procedures are inadmissible for substantial plans because of the state blast issue. To address these difficulties, my examination gives computerized test age methods utilizing disintegration of processor plan and property to make the formal strategies material practically speaking.

5.2 Coverage Driven Validation

A principle downside in simulation based approval is that a confirmation of the accuracy of the plan requires comprehensive reproduction which is conceivable just for little plans. At the end of the day, a specific level of certainty can be accomplished by recreating the configuration utilizing a huge volume of tests. Be that as it may, there is an absence of good measurements to evaluate his level of certainty and to qualify a test set. Subsequently, it is difficult to answer the question, ”At the point when is check done?” , because of trouble in measuring check advance furthermore, test viability.

A conventional stream of scope driven approval starts by characterizing scope metric, taken after by test age .A scope metric gives an approach to perceive what has not been confirmed and what tests ought to be included. Numerous scope measurements have been proposed for various sorts of plan blunders (e.g., control stream, information stream) and at diverse outline deliberation levels (e.g., behavioral, RTL, entryway level). In scope driven test age, tests are made to enact an objective scope point and it can viably diminish the quantity of tests contrasted with the arbitrary test age. Through reenactment, the scope is broken down by looking at whether target functionalities have been secured or on the other hand not. In the event that scope gaps are found, extra tests are created to practice them. In the event that higher level of certainty is required, the scope metric can be enhanced or utilization of extra scope measures can be made possible.

Verification architects can change the extension or profundity of scope amid the approval process. For instance, they can begin from basic scope measurements in the early check stage and utilize more perplexing scope measurements later on. In any case, existing scope measurements try not to have an immediate association with the outline usefulness, a scope metric is required in light of the usefulness of the outline. This paper gives an extensive practical scope metric by characterizing a useful blame model for pipelined processors. Albeit coordinated tests require a littler test set contrasted with arbitrary tests for the same utilitarian scope objective, the quantity of tests can in any case be amazingly substantial.In this manner, there is a requirement for useful test compaction procedures. My examination gives a useful test compaction strategy to lessen the coordinated test set.

The figure shows the overall procedure of the Coverage-Driven Architecture Verification Methodology. The initial step is to make a processor show and an functional fault model from the processor engineering particular. Next, it produces a rundown of all conceivable useful flaws in view of the blame model and the processor display under approval. Test compaction is performed before test age by disposing of the excess issues for the given plan imperatives. One of the rest of the shortcomings is chosen for test age. A test program for this blame is delivered consequently by formal onfirmation techniques, e.g., model checking.The fault then gets removed from the fault list.

Funnctional test verification is performed after this stream of test age. It is imperative to take note of that two stages of compaction procedures are connected prior and then afterward test age. This report makes three noteworthy commitments: I) improvement of effective fault models also, a scope metric for pipeline cooperation functionalities, ii) fault test age procedures utilizing formal strategies for current complex processor outlines, and iii) functional test verification.

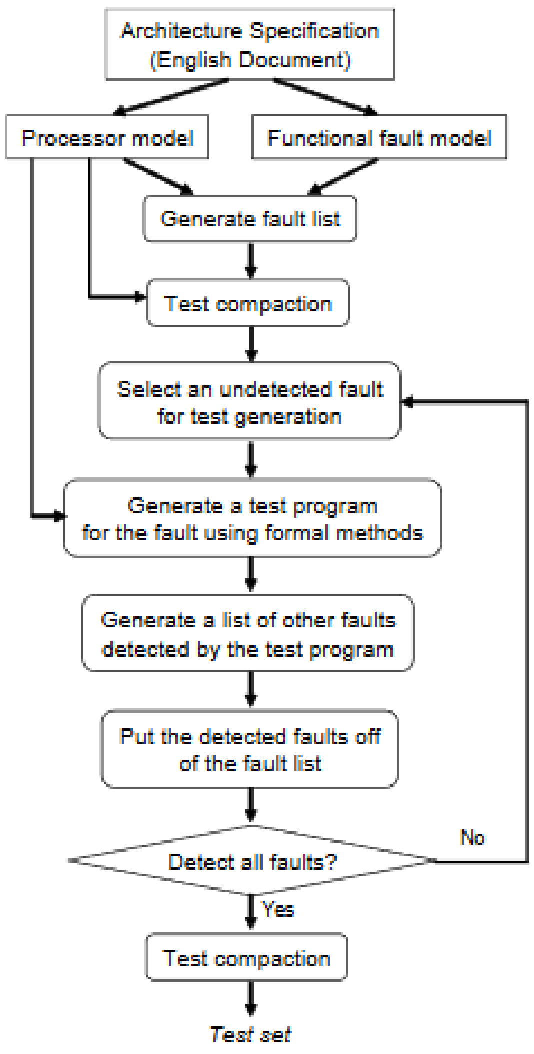

Architecture Specification (English Document)
Processor model
Functional fault model
Generate fault list
Test compaction
Select an undetected fault for test generation
Generate a test program for the fault using formal methods
Generate a list of other faults detected by the test program
Put the detected faults off of the fault list
Detect all faults?
No
Yes
Test compaction
Test set

Figure 5.2: Coverage-Driven Verification Methodology [3]

A pipeline connection fault model is characterised utilizing both diagram and FSM-based demonstrating of pipelined processors. The blame model is utilized to characterize a useful scope. The practical scope is utilized to gauge the approval advance by announcing the shortcomings that are secured by a given arrangement of test programs.

5.3 Model Checking Based Test Generation

A noteworthy bottleneck in processor approval is the absence of computerized instruments and methods for coordinated test age. Show checking-based test age has been presented as a promising methodology for pipelined processor approval because of its capacity of programmed test age. Nonetheless, conventional methodologies are unsatisfactory for extensive outlines because of the state blast issue in display checking. A proposed effective test age method utilizing both plan and property disintegrations to empower display checking-based test age for complex outlines.

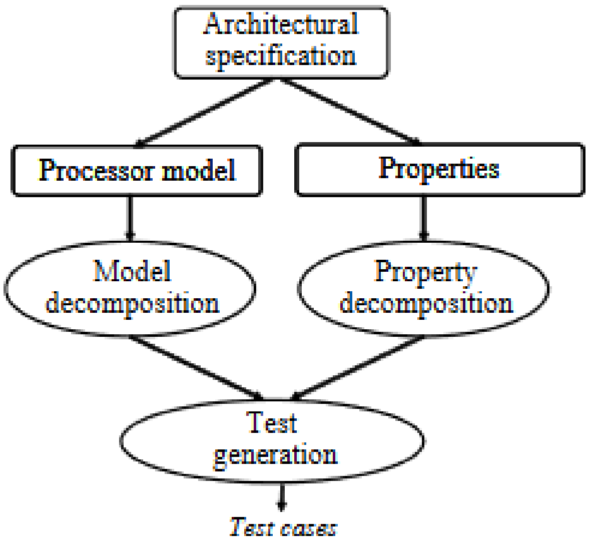


Figure 5.3: Test generation by Design Decompositions [3]

Processor model can be created from the engineering particular or can be produced by the architects. The properties can be produced from the particular in view of a useful scope, for example, diagram scope or pipeline communication scope. Extra properties can be included based intriguing situations utilizing joined pipeline arrange principles and corner cases. For proficient test age, the properties and the processor demonstration are disintegrated. Show checker and SAT solver are utilized to create incomplete counterexamples for he apportioned modules and decayed properties. These incomplete counterexamples are incorporated to develop the last test program.

The proposed technique makes three essential commitments: I) it builds up a technique for deteriorating a fleeting rationale property into different properties, ii) it introduces a calculation for combining the counterexamples created by disintegrated properties, and iii) it builds up an incorporated system to help both plan and design disintegrations for proficient test age of pipelined processors.

Model checking is a formal technique for confirming limited state simultaneous frameworks by demonstrating scientifically that a framework display fulfills a given particular.

The model is regularly gotten from an equipment or programming plan and the determination is ordinarily portrayed as fleeting rationale properties. Display checking additionally gives a computerized method for check contrasted with other confirmation strategies, for example, hypothesis demonstrating. Due to the capacity of finding even unpretentious plan blunders, show checking method has been effectively connected to numerous genuine framework plans and it has turned into a basic piece of modern configuration cycle. The confirmation system of model checking comprises of formal displaying of a plan, making formal properties, and demonstrating or invalidating by investigating the whole calculation space of the model thoroughly.

5.4 Test Generation Using Model Checking

Test generation using model checking is one of the most promising directed test generation approaches due to its capability of automatically producing a counterexample.[3]

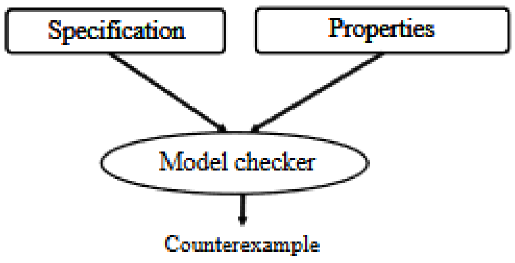


Figure 5.4: Test generation through Model Checking [3]

A processor model is described in a transient specification dialect and a desired behaviour is communicated as worldly rationale property. A model checker comprehensively seeks every single reachable condition of the model to check if the property holds (check) or not (misrepresentation), which is called unbounded model checking.[3] On the off chance that the model checker finds any reachable express that does not fulfill the property, it creates a counterexample.[3] This falsification can be adequately misused for test age. Rather than a coveted property, its nullified adaptation is connected to the model checker to create a counterexample. The counterexample contains a grouping of directions from an underlying state to a state where the negated form of the property fails.

Specification driven test age utilizing model checking has indicated promising comes about. It can produce test programs at early plan organize with no low-level usage learning. Figure demonstrates a determination driven test program age situation. A fashioner begins by indicating the processor engineering in an Engineering Description Language (ADL) that is utilized to catch both the structure and the conduct of the processor. A processor demonstrate is created from the ADL determination. Different properties (wanted practices) are created from the abnormal state microarchitectural processor determination. A model checker acknowledges the properties and the model of the processor to deliver test programs that are utilized for approval of the processor outline.

5.5 Deterministic And Random Instruction Sequence Simulation

It's clear to most intellectual property (IP) creators and users today that functional verification is one of the biggest problems facing the industry. As design complexity increases, the verification effort often over shadows the design effort. It becomes harder to detect obscure architecture functional errors and still provide the highest possible verification coverage. Most will agree that there is no single methodology or tool that solves what many are dubbing the "verification crisis". Thus an approach of multiple methodologies is required to tackle the problem from as many angles as possible. This is the approach that has been adopted at ARM , a mix of deterministic simulation, random stimulus generation, and automatic testbench generation with Verisity's (Mountain View, CA) Specman Elite.[7]

5.5.1 Deterministic simulation

The mainstay methodology that has been used since the early days of the first ARM Architecture design is deterministic simulation. This is a common and well understood methodology that offers a number of advantages, although it's limited by the amount of effort required to generate test cases and the performance of simulation tools. At ARM, test cases are developed as self-checking assembler sequences.These code sequences are then replayed on a simple simulation testbench consisting of the ARM CPU, a simple memory model, and some simple memory-mapped peripherals. The tests fall into two categories, AVS (Architecture Validation Suites) and DVS (Device Validation Suites).

All the AVS class tests check architectural functionality such as the instruction set architecture (32-bit and 16-bit Thumb), the exception model, and the debug architecture. The DVS tests focus on the behavior of specific cores and check corner cases arising from the particular implementation. An advantage of this type of testcase is that tests are self-contained and portable from ISS (Instruction Set Simulator) environments to Verilog or VHDL testbench environments, or to FPGA prototypes and eventually to silicon. Thus, the customers and the verification engineer can verify the functional equivalence of all these design views. These suites of tests are effectively the ARM architecture compliance suites.[7]

5.5.2 Random Instruction Simulation

Random stimulus generation is widely recognized as an effective approach for verifying corner cases that are hard to anticipate. It was found that, while most of design bugs are flushed out by the deterministic approach, random instruction sequences are also highly effective in hitting obscure cases, often finding bugs that may lay undetected for years in real-life applications. An internal tool generates targeted random code sequences known as RIS. With RIS, the self-checking tests are pre-generated using an ISS as the reference design.

Random Instruction Sequence (RIS) tools are broadly utilized over the methodology for architecture check and approval. These apparatuses are frequently used to discover verification bugs in moderately stable yet not yet develop RTL outline. RIS apparatuses are extremely successful in producing test situations that are difficult to imagine. Be that as it may, regularly totally irregular direction groupings are of little test an incentive for uncovering corner cases in the outline, particularly if the bug includes a grouping of occasions occurring in a restricted planning window. Macros can help improve the test nature of the produced guideline successions by giving controlled irregularity around a particular succession of directions focusing on a particular territory in the processor design.[7]

Chapter 6

Architecture Validation Suite

The architecture defines a common set of behaviours that software (such as OS Kernels) can rely on.

The objective of the AVS is to check adherence to that architecture definition so that:
1. ARM cores of same architecture version are software compatible with each other.
2. Cores from Architecture licensees are s/w compatible with all other cores (ARM and others) of the same Architecture version.

The Architecture Validation Suite (AVS) can be defined as : ”A set of examples of the invariant behaviours that are provided by the ARM Architecture Specification, so that implementers can verify if these behaviours have been interpreted correctly.”

Contractual requirement - ARM Compliant Cores must run and pass without modification of AVS test source code.A check that the implementer has not misinterpreted the architecture includes examples of areas of the architecture that are fundamental as well as known pitfalls and common misinterpretations.

There is often confusion between architecture validation and verification. Verification refers to the entire process of uncovering bugs in a design, with the ultimate goal of delivering the design to market as bug-free as possible. The scope of architecture validation is limited to ensuring that the architecture has been correctly interpreted.

6.1 AVK (Architecture Validation Kit)

1. Features of AVK:

- Self checking ARM Assembler tests.
- Organised as suites aligning to architecture areas/features.
- Supporting code (headers/include/macro files.

2. Infrastructure and Environment of AVK:

- Build, run, reporting scripts.
- Reference VAL library for AArch32 and AArch64.
- Non-synthesizable System Verilog reference testbench.
- Configuration files for abstracting the implementation.
- Capability to configure different architectural choices in model and environment.

AVK is the Kit containing several test generation suites that are intended to generate test cases for particular scenarios depicting a particular behaviour when the scenario is to be verified.

6.2 The Verification Flow

The verification flow can be listed as follows:
1. Gathering up the Validation Collaterals
2. Stacking up the Validation Collaterals into a Scenario Document
3. Writing templates by reading the scenario document
4. Feeding the templates into the Test cases generator which contains various libraries and support environment for writing the verification test cases.
5. The test cases are then fed into the AEM (Architecture Envelope Model i.e. a software model of designed architecture) to get the results.

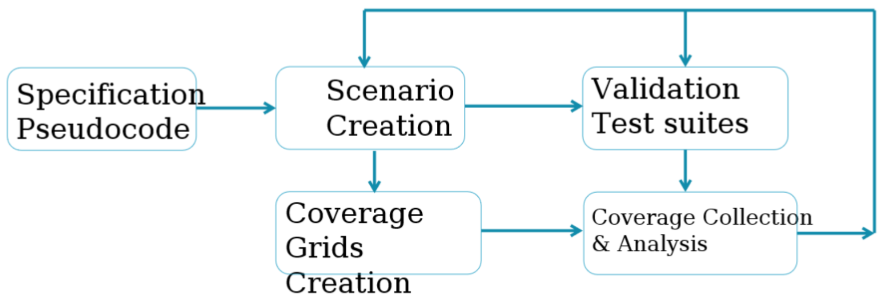


Figure 6.1: The Verification Flow

6.2.1 Gathering up the Validation Collaterals

Further breaking down the Step-1 procedure into following processes:

1. Specify the scenario into a tool for which the dataset has to be generated.

2. This tool performs symbolic execution and gives an intermediate output in the form of constraints.

3. The output received in step-2 is further fed into the SMT (Satisfiability Modulo theory) solver to give the actual scenarios in the form of register values. i.e. A reverse engineering process to find out the dataset which can create a scenario of interest (given as input to the tool).

4. Different scenario data or dependency dataset or checkpoint data thus gathers up to form the Validation Collateral Set.

6.2.2 Stacking up the Validation Collaterals into a Scenario Document

This process involves the following:

1. Input Sets:

- Input set can contain any of the data sets i.e Dependency Dataset, Scenario Dataset, Checkpoint Dataset.
- User provided set of rules to add up certain constraints on the Input Dataset.

2. Intermediate Tooling: This part contains the tools which renders the input set and converts into a Scenario document.

3. The output: The output contains the scenario document which further goes to some another test-case generation tool.

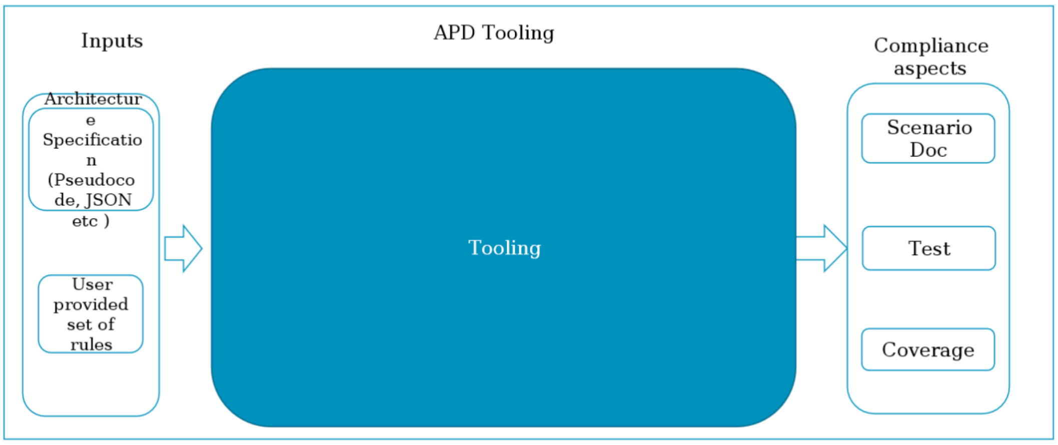

Inputs
APD Tooling
Compliance aspects
Architecture Specification (Pseudocode, JSON etc)
User provided set of rules
Tooling
Scenario Doc
Test
Coverage

Figure 6.2: Writing up the scenarios in Scenario document

Chapter 7

An approach for gathering Interesting Data-Sets (Step-2 Of the Verification Flow)

7.1 What are interesting datasets?

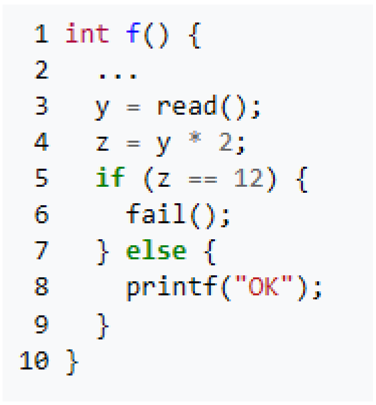
Interesting Datasets can be defined as the datasets which serve as the grounds for verification. The datasets that cover the corner cases, critical cases of the test scenarios. Such datasets should not be missed in verification methodologies. It becomes mandatory to cover all the scenarios of a particular test through these datasets to make sure that the architecture remains bug-free or with least possible number of bugs.

7.2 An Approach for collecting such datasets

7.2.1 Symbolic Execution

Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute. An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would, a case of abstract interpretation. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

Consider the program below, which reads in a value and fails if the input is 6.



```c
int f() {
  ...
  y = read();
  z = y * 2;
  if (z == 12) {
    fail();
  } else {
    printf("OK");
  }
}
```

Figure 7.1: A sample Code [8]

During a normal execution ("concrete" execution), the program would read a concrete input value (e.g., 5) and assign it to y. Execution would then proceed with the multiplication and the conditional branch, which would evaluate to false and print OK. During symbolic execution, the program reads a symbolic value (e.g.,) and assigns it to y. The program would then proceed with the multiplication and assign Λ * 2 to z. When reaching the if statement, it would evaluate Λ * 2 == 12. At this point of the program, could take any value, and symbolic execution can therefore proceed along both branches, by "forking" two paths. Each path gets assigned a copy of the program state at the branch instruction as well as a path constraint. In this example, the path constraint is Λ * 2 == 12 for the then branch and Λ * 2 != 12 for the else branch. Both paths can be symbolically executed independently. When paths terminate (e.g., as a result of executing fail() or simply exiting), symbolic execution computes a concrete value for Λ by solving the accumulated path constraints on each path. These concrete values can be thought of as concrete test cases that can, e.g., help developers reproduce bugs. In this example, the constraint solver would determine that in order to reach the fail() statement, Λ would need to equal 6.

7.2.2 Constraint Solvers - Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Examples of theories typically used in computer science are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays, bit vectors and so on. SMT can be thought of as a form of the constraint satisfaction problem and thus a certain formalized approach to constraint programming.

An SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable. In other words, imagine an instance of the Boolean satisfiability problem (SAT) in which some of the binary variables are replaced by predicates over a suitable set of non-binary variables. A predicate is basically a binary-valued function of non-binary variables. Example predicates include linear inequalities (e.g., 3 x + 2 y z $\geq$ 4) or equalities involving uninterpreted terms and function symbols (e.g., f(f(u, v), v) = f(u, v) where f is some unspecified function of two arguments.) These predicates are classified according to each respective theory assigned. For instance, linear inequalities over real variables are evaluated using the rules of the theory of linear real arithmetic, whereas predicates involving uninterpreted terms and function symbols are evaluated using the rules of the theory of uninterpreted functions with equality (sometimes referred to as the empty theory). Other theories include the theories of arrays and list structures (useful for modeling and verifying computer programs), and the theory of bit vectors (useful in modeling and verifying hardware designs). Subtheories are also possible: for example, difference logic is a sub-theory of linear arithmetic in which each inequality is restricted to have the form x y > c for variables and y and constant c. Most SMT solvers support only quantifier free fragments of their logics.

7.2.3 SMT Solving Approach

Early attempts for solving SMT instances involved translating them to Boolean SAT instances (e.g., a 32-bit integer variable would be encoded by 32 bit variables with appropriate weights and word-level operations such as 'plus' would be replaced by lower-level logic operations on the bits) and passing this formula to a Boolean SAT solver. This approach, which is referred to as the eager approach, has its merits: by pre-processing the SMT formula into an equivalent Boolean SAT formula we can use existing Boolean SAT solvers "as-is" and leverage their performance and capacity improvements over time. On the other hand, the loss of the high-level semantics of the underlying theories means that the Boolean SAT solver has to work a lot harder than necessary to discover "obvious" facts (such as x + y = y + x for integer addition.) This observation led to the development of a number of SMT solvers that tightly integrate the Boolean reasoning of a DPLL-style search with theory-specific solvers (T-solvers) that handle conjunctions (ANDs) of predicates from a given theory. This approach is referred to as the lazy approach.

7.2.4 Symbolic-execution based analysis

An important application of SMT solvers is symbolic execution for analysis and testing of programs (e.g., concolic testing), aimed particularly at finding security vulnerabilities. Important actively-maintained tools in this category include SAGE from Microsoft Research, KLEE, S2E, and Triton. SMT solvers that are particularly useful for symbolic-execution applications include Z3, STP, Z3str2, and Boolector.

7.2.5 The flow for gathering interesting datasets using the above

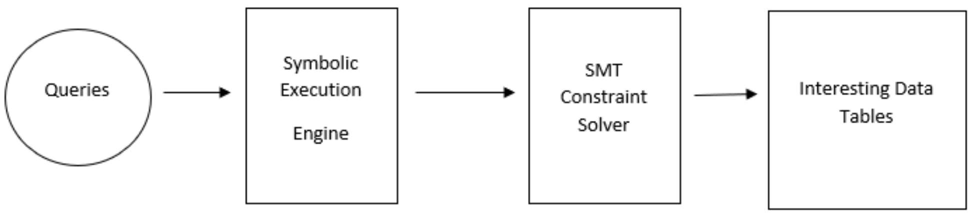


Figure 7.2: Collecting Interesting Data-Sets

The above figure depicts the approach for collecting interesting data-sets using a combination of Symbolic Execution and Constraint SMT Solver. It begins by passing the queries to the symbolic execution engine, which gives the output in the form of constraints by generating and analyzing the execution tree. Further these constraints are passed on to the Constraint Solver, which solves the equations (constraints) and provides the data-sets accordingly.

Chapter 8

Conclusion And Future Scope

8.1 Conclusion

AVK is widely used tool-kit (which contains various tools) for performing Functional Architectural Verification. Certain updates can help us touch the corner cases with a better hit rate. An automatically generated scenario document can minimize the human efforts for manually writing the Validation dataset into the same. It can reduce humanly mistakes which might be done while writing a huge amount of datasets into the scenario document. Since, its necessary to have a scenario document generated with minimal mistakes and in a proper order, an automated approach works better here.

8.2 Future Scope

- Contributed in the step of gathering interesting datasets to be further rendered in the form of a Scenario Document.
- Contributed in the automating the generation the scenario document.
- Looking forward to contribute in generation of templates for test-cases by automating as them as much as possible and having them execute on the AEM (software model of designed architecture).

Bibliography

[1] ARM Architecture Reference Manual (Beta) for ARMv8-A. Available: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architec-ture.reference/index.html`

[2] Stephens, Nigel, et al. "The ARM Scalable Vector Extension." IEEE Micro 37.2 (2017): 26-39.

[3] Koo, Heon-Mo. Coverage-driven Test Generation for Functional Validation of Pipelined Processors. Diss. University of Florida, 2007.

[4] Reid, Alastair, et al. "End-to-end verification of ARM processors with ISA-Formal." Proc. Computer Aided Verification (CAV).

[5] Prabhat Mishra and Nikil Dutt , Functional Coverage Driven Test Generation for Validation of Pipelined Processors , Proceedings of the conference on Design, Automation, and Test in Europe Volume 2. 2005

[6] `https://developer.arm.com/technologies/neon`

[7] `https://www.eetimes.com/document.asp?doc-id=1277271`

[8] `https://en.wikipedia.org/wiki/Symbolic_execution`

[9] `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories#cite_note-1`