TU WIEN Informatics

Yale

Automated large scale fault injection in ACT

FINAL REPORT

submitted in partial fulfillment of the requirements for

Marshall Plan Grant

by

Fabian Philipp Posch, BSc.
Registration Number 01456625

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Rajit Manohar, B.S., M.S., Ph.D.
Univ.Ass. Dipl.-Ing. Dr.techn. Florian Ferdinand Huemer, BSc.

Vienna, January 1, 2001

Fabian Philipp Posch

Andreas Steininger

TU WIEN Informatics

Yale

Automatisierte breit-gestreute Fehlerinjektion in ACT

ENDBERICHT

verfasst als Teil des

Marshall Plan Stipendiums

eingereicht von

Fabian Philipp Posch, BSc.
Matrikelnummer 01456625

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Rajit Manohar, B.S., M.S., Ph.D.
Univ.Ass. Dipl.-Ing. Dr.techn. Florian Ferdinand Huemer, BSc.

Wien, 1. Jänner 2001

Fabian Philipp Posch	Andreas Steininger

Contents

CHAPTER 1

Introduction

When a new product is released, focus is often exclusively put on its performance improvements over the previous generation. Unsurprisingly however, pure performance is never enough for a product to be actually useful. All the performance in the world does not help if the underlying system is unreliable. While most off the shelve electronics will be stable enough for home use under lab conditions, venturing out into a more challenging environments quickly opens up some major challenges. The obvious question to come to mind here is *what happens, when something goes wrong?* A question which has a quick answer so common, it has found its way into pop culture: Just turn it off and on again. Digging deeper into the details however, a slew of unknowns starts to rear their ugly head. What if the age old advice is not possible anymore, or worse, does not actually fix the issue? One cannot simply walk up to a satellite and hit the power button.

The main goal of this research is to take what we have learned about finding these answers and improving upon it - by building a set of tools which can be used by anyone. Their capability is not necessarily to find answers but to at least point out the problems.

The logic families we talk about in this work are not limited to synchronous logic. In fact, most of them are what is called asynchronous logic. While currently only very rarely employed in industry, asynchronous logic does away with the complexity of a clock net and replaces it with localized handshaking, enabling the logic to time itself. Even better, it completely rids the designer of thinking about the question of when the clock should actually hit and data be moved onwards. This introduction shall present a brief starting point into how asynchronous logic works - with a special focus possible consequences of unexpected environmental effects.

Following that we will go over the ACT toolchain, a fully open source set of tools for asynchronous VLSI, headed by Rajit Manohar at the Yale AVLSI Group [Man19]. It is a significant effort to enable an end-to-end tool flow, as VLSI tools for asynchronous logic are few and far in between.

1.1 Synchronous Logic

A look at any microprocessor datasheet will quickly draw attention to the clock frequency with which the device is guaranteed to run. The signal net behind this specification, often just referred to as *clock* tries to solve several problems of digital design, the biggest of which is synchronization. In synchronous digital designs, data is moved through a *pipeline* between *stages* whenever a positive clock edge appears. The combinational logic inside these stages is designed to finish calculating the next result at least faster than one clock period. When the clock hits, all stages have finished their calculations and no incomplete data is moved to the next stage. This way, no per-stage control logic is needed to detect completion of the current calculation.

This approach comes with a number of drawbacks. As the clock synchronizes all stages within the current clock domain, it has to reach all of them at the same time, making it a high-speed (thus low-slew), low-jitter, low-delay-variance, global signal, which is about the worst set of requirements to be attached to a signal. For this reason, a lot of engineering effort in a large synchronous design is dedicated to tuning the clock net. Additionally, the requirement for a short rise time often results in driving the clock net with relatively high power. This amounts to the clock net consuming an estimated $30 - 40\%$ of a chip's power budget [Moa17; SBS10] on its own.
To add insult to injury, clocked designs are by definition worst case designs. The clock cannot run faster than the slowest stage within a given clock domain. Even if this *critical path* is part of a fork in the pipeline and only used 50% of the time, the clock cannot run faster as it is not necessarily known whether or not this path is currently in use. Additionally, as chip characteristics change with voltage and temperature, a certain design point has to be assumed and the clock frequency calculated accordingly. This design point is often conservative and therefor further slows the chip.

1.2 Asynchronous Logic

In contrast to a synchronous approach, asynchronous logic does not employ a clock in order to synchronize between stages. The pipeline and stage concepts still exist, but now these stages are able to communicate with each other by adding handshaking logic to each stage. This takes the shape of a ready/valid protocol, which communicates state between two neighboring stages. Chained together, this creates back-pressure, halting subsequent stages as the pipeline fills up. Not only does this do away with the high-power clock net, but it also alleviates the issue of worst-case bound performance. If a given pipeline branch is not taken, it never contributes to the speed at which a given data token can move through the pipeline as a whole. This leads to average-bound rather than worst-case bound performance which often has a big real-world impact. In addition, if current environmental conditions allow for it (higher voltage, ...), transistors switch faster and the designs gains additional performance.

While rare, asynchronous designs have been explored in industry. In Davies et.al.

[Dav+14] the design of a 72 port 10G Ethernet switch is explored. Merolla et.al. [Mer+14] presents a neuromorphic chip to process machine learning workloads using spiking neurons.

CHAPTER 2

Asynchronous circuit description

This project aims to unify the workflows from TU Vienna and Yale University. Both sets of tools use different ways of describing asynchronous circuits on different abstraction levels. We will briefly describe these levels of abstraction and their usefulness towards our goals, as well as contrast the two against each other.

`act` [Man19], the toolchain produced by the Yale University AVLSI lab under the direction of Rajit Manohar is an attempt at and end-to-end solution. While it has the ability to synthesize to Verilog for FPGA acceleration [DM22], this is actually emulation rather than native execution. The system auto-generates a hardware-level simulator for the circuit and runs it synchronously on FPGA.

`pypr` [Hue22], the equivalent by ECS TU Vienna is less an attempt to create a full ASIC toolchain. Its main purpose is to quickly create and modify mostly quasi delay-insensitive (QDI) circuits and examine their robustness. In contrast to `act`, `pypr` possesses the capability to directly synthesize to VHDL and Verilog.

Both toolchains feature sub-languages for different levels of abstraction. `pypr` uses Python as its environment. Circuits are represented by Python objects and rules are added by calling functions on them. These objects can then be connected into a hierarchical design [Hue22].
In contrast, `act` has its own custom meta-language, in which processes can be defined, instantiated and connected. Rules and functionality are then added by sub-language blocks within these processes.

For both languages, design units have inputs and outputs, which can be channels or rails. A channel refers to a bundle of signals which jointly carry data as well as their respective control signals. These control signals carry handshaking between two stages.

Since the abstraction levels are similar between both toolchains, we will discuss them in their respective groupings and point out differences.

2.1 Abstract Serial Description / CHP

This level of abstraction only features in `act` as the CHP sub-language. In addition, it is currently of little use to this particular workflow, as there is of yet no way to synthesize CHP into arbitrary logic families.

CHP can be used as a sub-language block in `act`, follows a serial execution paradigm, and features directly accessible channel semantics. A simple adder can be seen in Listing 2.1.

```
defproc chp_adder (chan?(int) A, B; chan!(int) RES)
{
    int a_buf, b_buf;

    chp {
        // receive the input
        A?a_buf, B?b_buf;

        // send the output
        RES!(a_buf + b_buf)
    }
}
```

Listing 2.1: Simple Adder in CHP

? and ! represent the channel operations *send* and *receive* respectively. The process reads from the input channels concurrently (as specified by their `,` separator), then sends the addition result onto the *RES* channel, once both values have been received (using `;` as the serial execution separator).

2.2 Dataflow

The dataflow representation of a circuit can be viewed as a graph of building blocks with a defined functionality. This graph represents a pipeline and thus the flow of data in the design. The individual design blocks are treated as black boxes. When a dataflow design is synthesized into a specific logic family, the design blocks are substituted by a specific implementation for the target logic family. This makes dataflow as of yet the most powerful abstraction level. As long as a library exists, any logic family can theoretically be targeted. This includes synchronous families as well asynchronous ones.

`pypr` does have dataflow functionality, however it is mainly used to assemble user defined blocks into pipelines [Hue22]. In contrast, `act` already has a powerful library for base components.

A basic dataflow adder in `act` dataflow is shown in Listing 2.2. Note the absence of channel control compared to Listing 2.1. Connections between instances are inherently assumed to be channels handshaking is left completely up to the connected blocks.

```
defproc dflow_adder (chan?(int) A, B; chan!(int) RES)
{
    chan(int<DATA_WIDTH+1>) intermediary;

    dataflow {
        B + A -> RES;
    }
}
```

Listing 2.2: Simple Adder in Dataflow

2.3 Production Rule Set

Production Rule Sets (PRSs), as introduced by [Mar89; Mar91] describe conditions for signal transitions using guards of the form $G \mapsto S$. If G evaluates to true, transition S fires. A subset of production rules is directly CMOS implementable, as guards represent pull-up and pull-down stacks of logic gates.

Both frameworks support production rules using differing containers. An example of an `and`-gate in `pypr`'s PRS language can be seen in Listing 2.3 and the equivalent in `act` PRS in Listing 2.4.

Of note is the absence of channel semantics in the PRS language. As it is supposed to be directly describing transistor stack behavior, PRS does not handle channel operations implicitly anymore.

```
prs simple_and_gate is
inputs a : Bit; b : Bit;
outputs out : Bit;
begin out := rule(a and b);
end prs;
```

Listing 2.3: Simple `and`-gate in `pypr` PRS

```
defproc prs_and (bool a, b, out)
{
    prs {
        a & b => out+
```

```
    }
}
```

Listing 2.4: Simple `and`-gate in `act` PRS

`act` has the ability to handle direct transitions between the Communicating Hardware Processes (CHP) and PRS abstraction level. This is achieved by an internal channel library. Channel operations are split into certain operational phases, which trigger specific signal transitions on the PRS side and vice versa. This transition point also allows us to extract communication errors for failure analysis.

2.4 Benefits and Drawbacks of Description Levels

As the level of abstraction decreases, development and simulation effort increases. Focusing on `act` as our target ecosystem, CHP is the easiest to write and simulate. Dataflow is a middle ground for implementation, but depends highly on library support - completely new functions have to either be assembled from base components or written from scratch using PRS for every target logic family. PRS is the closest level to actual hardware and thus presents itself as the only useful abstraction level for fault injection testing. This level was also targeted by [Beh21].

In summary, the best workflow for the current state of the toolchain is for the testing harness to be written and simulated in CHP, the design under test (DUT) written in Dataflow, then mapped and simulated in PRS.

There are ongoing efforts to synthesize CHP designs to any arbitrary logic family, however these tools are not finished as of writing of this report.

2.5 Important Circuit Elements

In asynchronous logic, data is often modeled as a token moving through the pipeline. While this model is just as valid for the synchronous domain, here the tokens will keep moving with every clock tick and interactions between the tokens are mostly irrelevant. As asynchronous is event rather than time driven, data moves only when there is data to be moved, and tokens can interact of block each other from proceeding.

It is helpful to define certain circuit building blocks, which often appear in asynchronous circuits - especially in verification setups.

- **Token source.** A block of logic generating new tokens, which are fed into an input.
- **Token sink.** A block of logic which continuously consumes tokens from an output.

Verification environments often consist of a design under test (DUT) as well as a verification harness. This harness (even in the synchronous domain) usually contains a token source (referred to as a sequences in UVM) as well as token sinks.
In addition, two more parts enable the evaluation of a DUT:

- **Model.** Usually a higher abstraction level implementation with the same behavior as the DUT. It is assumed that the re-implementation leads to fewer or at least different errors. The output of both the DUT and the model are compared by the
- **Scoreboard.** It takes the output from both the model and the DUT and compares the outcome, flagging mismatches as failures.

CHAPTER 3

Related work / starting point

Over the past years, TU Wien has contributed a significant amount towards understanding resilience of asynchronous circuits, specifically QDI pipelines. For this purpose, several works before this one have contributed to the tool DiFit, which performs failure analysis through large scale fault injection. In essence, this is the same approach as the one we have chosen for our new tool. The main difference is the reliance of custom internal tools, which we aim to reduce in our own implementation. The following is an overview of the capabilities of DiFit, as presented by Behal [Beh21] and Schwendinger [Sch22].

3.1 Capabilities

DiFit provides a framework for distributed injection into asynchronous logic circuits. It uses a the already described Python based circuit description library `pypr` under the hood (see Chapter 2). This means the circuit can theoretically be synthesized into synchronous and asynchronous logic families, depending on library support by `pypr`. [Beh21] has used this capability to analyze the multiple families of asynchronous circuits for their fault resilience.

To investigate fault resilience of a given circuit, DiFit generates a set of testcases, which inject faults into wires of the design, varying their parameters for every testcase. These testcase parameter variations are randomly chosen inside a given parameter space, which tries to establish realistic conditions of circuit state as well as limit the amount of simulation time required.

3.2 Internal Tool Flow

3.2.1 Circuit description framework

DiFit uses `pypr` to provide circuit operation capability. A DUT is written in `pypr` as Python object and then mapped to a set of gate implementations. The current version as presented by Schwendinger [Sch22], DiFit can target either VHDL or PRS representations to enable gate level simulation. We will talk more about the latter in Section 3.5. To validate the output of the DUT, the output is compared against a model written in VHDL or CHP respectively. The testbench and scoreboard to perform these checks are generated using the OSVVM library for VHDL, an open source toolkit to validate designs written in VHDL, or using `tbgen.py` with a custom CHP testbench for PRS.

As already mentioned, the now fully assembled test harness is then either compiled into VHDL or flattened PRS. For the Python based circuit description, this means a more or less direct mapping through a circuit library, as already discussed in Chapter 2.

3.2.2 Testcase generation

After the harness for the DUT has been assembled, testcases are generated. DiFit uses certain key metrics in its generation and reporting which it determines through exploratory simulations. Testcase generation is separated into several steps:

1. **Measure pipeline depth.** The sink delay at the end of the pipeline is set to a high value and the input fed until no further input can be accepted. The amount of tokens fed to the system is assumed to be the pipeline depth.
2. **Measure source / sink timing for a given load scenario.** The target load scenario of the DUT can be configured using the YAML configuration file. The tool then tries to trim sink and source delays to hit this load point. This is only really possible for pipeline layouts without feedback or forks. We will return to this issue in Section 3.4.1.
3. **Determine the wait time after injection.** After an injection has occurred, a potential fault needs time until it becomes visible at the output.
4. **Perform a golden run.** The system performs a golden run to determine timing of all events in the circuit. This is used as a reference to generate timing failure reports.
5. **Maximum simulation time is determined.** This time boundary is used to determine if the circuit has potentially reached a deadlocked state.
6. **Determine the number of necessary injections.** DiFit automatically calculates the amount of necessary injections. Given the size of the search space, this is not an exhaustive search. The heuristic employed here is based on average injection density.

In order to gauge the parameter space and thus the number of needed injections, DiFit first runs parameter randomization many times. It then examines often randomly generated parameters collide with each other. Based on the inverse of the Birthday Problem [Von39], an estimation of the total parameter space is made. From this, the number of needed injections is calculated to reach a certain injection density.
Based on this metric, it is assumed that most possible failure scenarios will be triggered by the setup.

The victim signal for a single injection is chosen randomly from the list of all signals of the design as part of parameter randomization. There are no additional characteristics used for selection or distribution. Polarity and duration of the transient are also random. The original framework presented in [Beh21] can inject high or low polarity into the circuit, [Sch22] expanded the framework to also include metastability injections when using `prsim`.

3.2.3 Cluster

The setup application as well as the worker nodes connect to a MySQL database. When the setup part is run, the generated testcases are uploaded into this database. Worker nodes load these configurations and start processing the individual testcases, uploading their results back into the database. To save space, only runs which deviate from the golden run performed during setup are saved. The data includes the failure type, the seed for the pseudo-random generator (PRG) generating the input data, the generated logs, as well as the commands used to configure the simulation.

After completing simulating all testcases, this data can be processed for further investigation.

3.2.4 Simulator

DiFit uses Modelsim as the primary simulator. In later efforts [Sch22], `prsim` was added as an additional simulator (see Section 3.5).

3.3 Output

The generated data is manually extracted from the database and then further processed using Jupyter Notebooks. The framework does pre-categorize outcomes into failure modes, these are:

- **Deadlock.** The simulation took longer than the determined timeout limit.
- **Glitch.** A glitch was observed within the design.
- **Value.** The DUT output differed from the expected model output.

- **Coding.** A dual-rail encoded bit saw true and false rail in a high state simultaneously.
- **Timing.** The DUT timing varied from the golden run.

3.4 Shortcomings

The setup described here has a list of shortcomings, which we are important to point out. This list was the jumping-off point for the tool we developed during the length of this program.

In no particular order, the issues we found were:

3.4.1 Pipeline Load Factor

DiFit calculates the Pipeline Load Factor (PLF) by measuring the delay of acknowledgement for the token source as well as the waiting time between tokens at the token sink [Beh21]. It then defines PLF as

$$PLF = \frac{\delta_{\text{acknowledgement}}}{\delta_{\text{token-token}}}$$

Furthermore, it defines that a pipeline is *bubble limited*, if this factor is greater than 1 - we are waiting for acknowledgement longer than for new output tokens. If the factor is smaller than 1, the pipeline is *token limited* - the pipeline is outputting tokens faster than we are feeding it.

This definition deviates from the more common definition of the metric, with the advantage of being well defined for not just linear pipelines. It adds potential for confusion however, as classical definitions usually constrain it to $0 \leq PLF \leq 1$, where 1 corresponds to a pipeline with all buffers filled and 0 to all buffers empty.

Subsequently, DiFit attempts to steer the PLF towards its configured target by varying source and sink delays. These are determined in a multi-step process:

1. Increase the sink delay until the required simulation runtime increases proportionally with the number of injected tokens. Fix the delay as the corner value.
2. Repeat the process with the source delay. Now both ends are tuned to the circuit speed.
3. If the measured PLF is too low, increase the sink delay until the configured PLF value is reached. If the PLF is to high, increase the source delay until the configured PLF value is reached.
4. Repeat this process 100 times and set the final source and sink delays as the mean of these 100 runs. This aims to lower the influence of data dependent delay changes.

While dialing in a certain PLF could be useful under certain circumstances, we argue that a more realistic model of the logic feeding and consuming from the DUT would yield a more realistic testing analysis, as actual designs rarely target a specific PLF over targeted total throughput.

3.4.2 Number of Injections

Behal [Beh21] already talks about possible better solutions for determining the number of needed injections. In addition, Behal does not consider the potential impact a specific signal has on the rest of the circuit. A node with fanout of 1 will see as many injections as a signal with fanout of 5. We postulate that injecting based on the number of affected nodes would significantly cut down the number of simulations needed or potentially improve failure mode awareness.

3.4.3 Steady State Expectation

All designs tested with DiFit are expected to reach a steady state after 2 tokens. As argumentation for it, Behal only mentions empirical experiments without providing data. We feel this should simply be a configuration option to add flexibility to the system.

3.4.4 Pipeline Depth

After injection, the simulation needs to wait until potential faulty tokens have a chance to propagate to the output. To determine the number of tokens to wait for, DiFit performs an exploratory simulation during setup. The sink delay is set to a very high value and tokens are sent on the input until no more tokens can be consumed.

This leads to inaccuracy however, as the sink *is only set to a high delay, not fully halted.* A very long pipeline could see its value tainted by a token being absorbed by the sink. Even worse, a nonlinear pipeline might see one of several erroneous measurements. Assuming two branches of differing length:

1. Only the short branch gets filled: Measured pipeline length too short
2. Both branches get partially filled: Measured pipeline length too long

Due to its complexity, we feel this problem can simply be solved by making the post-injection wait time a configuration option instead of trying to measure it.

3.4.5 Class of possible circuits

DiFit expects circuits to have a singular input. Circuits with more than 1 input need to be wrapped accordingly. This inherently requires a custom testbench setup for DiFit and rules out code reusability from other verification tests. This potentially also negates the benefit of automatic testbench generation.

3.4.6 Simulator

Using Modelsim as a simulator inherently requires licensing of the software. As each installation - meaning each cluster node - needs a separate license, this solution is not easily financially scalable.
In addition, Modelsim does not support metastable circuit states, prohibiting the analysis of a potentially quite dangerous class of failures.

This issue was partially addressed with the extension to use `prsim`.

3.5 ACT tool flow integration

In a later efforts, initial work was performed by Schwendinger [Sch22] to integrate DiFit with the `act` toolchain by the Yale University AVLSI Lab [Man19]. The main addition is a translation layer from the Python based `pypr` description to PRS. This in turn enables two new capabilities:

1. Designs written in `act` can also be tested using DiFit
2. `act`'s `prsim` can be used as a simulator instead of Modelsim

A prospective design is translated from higher level `act` or `pypr` description to low level PRS, with the same circuit type restrictions still applying. The design hierarchy is then flattened using `aflat` as a requirement for `prsim`, including a CHP testbench instead of using OSVVM.

This does address the issues of effectively scaling a simulation cluster, but does not resolve the other issues.
`prsim` does not internally support randomizing delay of signals during runtime. Since DiFit requires this functionality, all input tokens are generated by their own source with static delay. Thus, significantly more overhead is generated as more hardware is introduced and simulated.

CHAPTER 4

Research questions

In light of creating a new tool flow for tackling this problem, we have set out to answer the following research questions:

How can we make the tool more general purpose? Right now, the tool is very limited in terms of which circuits can be simulated. DiFit only accepts setups with a singular input. We would like to expand this, so circuits with more than one input can also be evaluated.
We think that a new tool should not only be able to evaluate circuits with more than one input, but to reuse a general verification harness without modification as well.

Can we reduce the amount of overhead? Both Modelsim and `prsim` have to simulate the entire design on the same refinement level - meaning gate level. This creates a tremendous amount of overhead. Even worse, for `prsim` every injected data token uses its own source, to work around the lack of delay randomization support.
Switching to a simulator with support for simultaneous simulation of differing abstraction levels should reduce the performance impact of the surrounding harness. In addition, support for delay randomization is needed to reduce the amount of required dummy hardware.

Can we create a more streamlined workflow? The current workflow uses a number of workarounds.
As `act` is shaping up to become the new standard for asynchronous logic description, we want to create a more well integrated tool flow to ease development of future designs. In a perfect world, these tools are flexible enough to be used for more of the workflow than just fault resilience testing. Distributed simulation as well as on demand remote compute in general is a useful capability in any hardware development workflow.

Can we reduce the number of required injections? Currently, average injection density is used to determine the overall number of injections needed. We feel this is a poor representation of a signal's potential to cause failure conditions. Either signals

with high failure potential are underrepresented, or signals with low failure potential are overrepresented.
We need to find a metric which determines a signal's potential to cause harm. Based on this, we can estimate the number of expected possible failure modes, allowing us to calculate an estimation of required configurations to find all of them using the Coupon Collector's Problem. Then we need to distribute the total amount of injections over all signals based on their potential to cause harm.

CHAPTER 5

Cluster Build and Simulation System

The most development-heavy part of this project was implementing a new distributed system to compute simulation results. We chose to start from scratch, as this allowed us to widen the flexibility of the tool significantly. Nonetheless, the architecture of the first version is still heavily inspired by DiFit.

Hardware development requires a lot of compute, not just for this specific application. Verification in an industrial context is usually also achieved through subjecting a DUT to a large number of test inputs, thus also benefiting heavily from distribution. Outside of simulation, synthesis often requires more powerful compute and main memory than most client workstations provide.
For these reasons, we believe a more general case cluster based build system to be of use in a high performance toolchain like `act`. Since the end goal is a configurable shoot-and-forget solution, we chose the name `action`. Due to the limited scope of this project, only the initial distributed simulation capability has been implemented thus far, and robustness of the net-code has been sacrificed in the name of simplicity. We designed the underlying structure of the source to allow for simple modification, to extend the width of functions supported by the framework, as well as rework the communication between nodes for better robustness.

5.1 Basic Setup

`action` is configured through YAML files. These files are read by the engine internally converted into a pipeline. This pipeline consists of data ingestion, as well as two major processing phases. These phases are populated with *modules*, which are capabilities implemented in `action`. As soon as a module is instantiated in a pipeline, it is referred to as a pipeline stage (not to be confused with stages in the DUT).

The *prepare* phase is performed locally and allows for any major elaboration or preprocessing steps. Currently, these operations are performed serially, although this might change in the future. At this point in time only testcase generation uses this part of the pipeline, which was done to initially limit the scope of the project. This has the inherent downside that a synthesis-verification pipeline could currently not be run in the cluster alone, as synthesis results would have to be downloaded and re-uploaded (inherently breaking the workflow into two distinct pipelines). We plan on addressing this shortfall in the future (see Chapter 8). Results from every step are saved in variables, which are from hereon out referred to as *artifacts*. These artifacts can have different types, such as an `act` design object or a collection of testcases. The YAML itself does not contain type information for artifacts other than the ones loaded at the beginning, but type checking is automatically performed when the configuration is loaded.

The *deploy* phase happens after *prepare*. `action` determines which artifacts are needed in the cluster and uploads them to the central database alongside information regarding the tools which need to be run. Agents then perform these tasks and dump their results back into the database, from which it can be retrieved after the computation has finished. This has the added benefit of the client being able to disconnect while the simulation is in progress.

5.2 Configuration

As already mentioned, `action` is configured using YAML files. A very basic configuration can be seen in Listing 5.1. This configuration does three things:

1. Load the `act` design saved in `test.act`
2. Generate a testcase using the `simple` testcase generator
3. Simulate the simple testcase in the cluster

We shall first discuss common elements between these configuration elements and then break them down individually.

```
input_artifacts:
  - name: design_file
    type: act
    source: file
    path: test.act

prepare:
  - module: testcase_generation
    outputs:
```

```
      tests: simple_tests
    generator: simple

deploy:
  - module: actsim
    inputs:
      design_file: design_file
      sim_configs: simple_tests
    outputs:
      sim_outputs: sim_results
    top: test
```

Listing 5.1: Basic `action` configuration

5.2.1 Ports

All of the stages described in the configuration in Listing 5.1 have one or more inputs and or outputs, here generally referred to as *ports*. These are denoted per stage as `inputs:` and `outputs:` in the `prepare:` and `deploy:` sections. As `input_artifacts` only serve a single output, their artifact port is defined under the `name` property (see Section 5.2.3). Ports are typed and directional. `action` performs dependency checks when a pipeline configuration is loaded. These checks include making sure an artifact exists and is of the right type. Artifact names can be reused, the old data is overwritten.

Of note: Only artifacts which are needed during the *deploy* phase are actually uploaded to the cluster. In fact, artifacts unload from memory as soon as execution has passed the last point of use. As of writing of this report, there are known memory leaks within the original `act` framework, which might cause performance issues when unloading an `act` object. This is a known issue and will be mitigated in the future.

5.2.2 Modules

Modules are the building blocks from which a pipeline can be assembled. Internally, everything is treated as a module, even if this structure is simplified away for loading input artifacts for the sake of configuration readability. Given there are two main sections of pipeline execution, there are also two different lists of modules. *prepare*-modules can be accessed during the *prepare* stage of the pipeline and perform local computation. Currently, only the testcase generator module is supported in the *prepare* stage.
deploy-modules can be accessed during the *deploy* stage of the pipeline and less reflect tools available on the client machine as they do the capabilities of the cluster. Currently, only simulation tasks are implemented for deployment. Using a local artifact on a consuming port of a deploy stage will automatically cause `action` to upload it to the cluster during pipeline execution.

5.2.3 Input Artifacts

Input artifacts represent a single output port, generating a single output artifact. Since input artifacts can load multiple types of data, this type information can not simply be derived from the stage itself and has to be configured manually. An input artifact configuration always covers the following keys:

- `name`: The artifact name the data shall be saved in
- `type`: The type of data loaded (currently only `act` is supported)
- `source`: The data source from which the data shall be loaded (currently only `file` is supported)
- `path`: Required by `file` data source. The relative path on disk to the source file.

5.2.4 Pipeline Assembly

Both *prepare* and *deploy* section instantiate modules as stages through a YAML list. For both, the set of keys is identical. A stage instantiation always covers the following keys:

- `module`: The name of the module
- `inputs` (if applicable): Input port names and the artifacts they are mapped to
- `outputs` (if applicable): Output port names and the artifacts they are mapped to
- Local settings (if applicable)

5.2.5 Testcase Generation

The testcase generation module is used to generate procedural simulation setups for `actsim`. Instead of different modules for different types of test generation, only one testcase generator module exists with interchangeable generation engines. These engines can be selected using the `generator` key. Currently supported is the `simple` generator which only emits the `cycle` command for `actsim`, and `naive-set-injection`, which generates a very simple list of injection tests on a list of signals similar to Behal [Beh21]. This mode at this point does not pay attention to a signal's potential to cause different failure modes, and thus does not yet address the shortcomings of the older system as discussed in Section 3.4.2.

The testcase generation module has a variable set of ports, depending on which generation engine is used. By default, the module only has the output port `tests`, which is the collection of generated test cases.
`simple` has no additional ports, as its output is not dependant on any parameters.
`naive-set-injection` takes a list of victim signals as an additional input.

Generators can have an additional set of parameters.
`naive-set-injection` has the following parameters:

- `victim-iterations`: Multiplier for iteration generation
- `victim-mode`: Victim signal distribution selection (only `random` supported for now)
- `injection-windows` (list): Time windows in which to generate injections
- `injection-duration`: Maximum time an injection can last
- `inject-undefined`: Flag whether or not to inject metastability
- `random-seed`: Seed for the PRG to generate injections. Setting this makes results reproducible

5.2.6 Simulation

The `actsim` simulation *deploy*-module has two inputs:

- `design_file`: The `act` design to simulate
- `sim_configs`: The testcases generated by the testcase generator

The `actsim` module only has the `sim_outputs` output port, which emits the simulation results.

Finally, the module the local parameter `top`, which informs the simulator about which process it should treat as the design's top process.

5.3 Cluster Architecture

Once a pipeline configuration is executed and the *deploy* stage is reached, the client starts interacting with the cluster architecture. For the sake of simplicity, the cluster currently does only consist of a central PostgreSQL database as well as a set of worker nodes. The client as well as the workers connect to the central database and modify it to communicate activity. Workers then prefetch tasks, block them in the database, and queue them for simulation. Results from `actsim` are then placed in an upload queue and dumped back into the database. Communication is handled through a shared cluster library, which expands upon the `pqxx` library.

5.3.1 Shortcomings

The current setup does not handle worker crashes gracefully, in addition the database layout leaves to be desired. Much of the state handling and assertion checking automation is directly handled by the database, which speeds up execution but limits the scope of possible functionality. Additionally, certain client or worker crash modes exhaust PostgreSQL's maximum concurrent client threshold. All in all, the net-code implementation is not very robust. This is a clear consequence of the design decisions made for the layout of this cluster. We do intend to rework this part, see Chapter 8 for a more detailed discussion.

5.4 Database Layout

The database consists of several tables which hold the hierarchical data structure. The following shows their initialization code and talks about what information these fields hold.

```
CREATE TABLE jobs (
    id char(8) PRIMARY KEY DEFAULT random_string(8),
    job_status status_type NOT NULL DEFAULT 'halted',
    time_added timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    total_passes bigint NOT NULL CHECK (total_passes >= 0)
        ↪ DEFAULT 0,
    finished_passes bigint NOT NULL CHECK (finished_passes >= 0
        ↪ AND finished_passes <= total_passes) DEFAULT 0
);
```

Listing 5.2: Jobs table initialization code

When a pipeline is deployed, its specific invocation is referred to as a job. Every job has an alphanumeric string ID for easier handling. A job has a current status as well as some additional status information. One job consists of one or more passes, which have interdependencies. A pass is the cluster equivalent to a stage in the local pipeline configuration. The job table initialization code can be seen in Listing 5.2.

```
CREATE TABLE passes (
    id uuid PRIMARY KEY DEFAULT uuid_generate_v4(),
    job char(8) NOT NULL REFERENCES jobs(id),
    pass_type pass_type_type NOT NULL,
    pass_status status_type NOT NULL DEFAULT 'halted',
    total_tasks bigint NOT NULL CHECK (total_tasks >= 0)
        ↪ DEFAULT 0,
```

```
    finished_tasks bigint NOT NULL CHECK (finished_tasks >= 0
        ↪ AND finished_tasks <= total_tasks) DEFAULT 0,
    depends_on uuid[]
);
```

Listing 5.3: Passes table initialization code

Passes hold the dependency information of the pipeline graph. They themselves consist of one or more tasks. One simulation pass can contain one or more simulations. The code shown in Listing 5.3 is only the setup code for the generic pass table. The database makes heavy use of PostgreSQL's inheritance feature, which is mostly meant for database partitioning. We use it to avoid saving additional pass information fields for passes which do not require it. As an example, the setup code for the `actsim` pass table can be seen in Listing 5.4.

```
CREATE TABLE actsim_passes (
    design_file uuid NOT NULL,
    sim_configs uuid NOT NULL,
    top_proc text NOT NULL,
    outputs uuid NOT NULL DEFAULT uuid_generate_v4()
) INHERITS (passes);
```

Listing 5.4: `actsim` passes table initialization code

Finally, the smallest unit of work is a task. A task could be a single simulation or a synthesis run. Tasks do not require their own table and are instead represented as the artifacts they result in. Since the number of outputted artifacts is known before execution starts, their database entries are created at initial deploy-time. This way, progress of all stages can be tracked without a controller.
Artifacts once again use the inheritance feature. The setup code for the artifacts table as well as the simulator configurations artifact sub-table can be see in Listing 5.5.

```
CREATE TABLE artifacts (
    id uuid PRIMARY KEY DEFAULT uuid_generate_v4(),
    artifact uuid NOT NULL,
    source_pass uuid NOT NULL,
    part_status status_type NOT NULL DEFAULT 'in_progress'
);

CREATE TABLE sim_configs (
    id uuid PRIMARY KEY,
    sim_commands text[] NOT NULL,
    has_reference uuid
```

```
) INHERITS (artifacts);
```

Listing 5.5: Artifacts table and simulator configurations sub-table initialization code

5.5 Worker Nodes

A worker node is a program which directly connects to the database. Its main job is to execute the tasks contained in the database.

To save on networking delays, the worker node prefetches a set number of tasks locally and schedules them. Since there is no controller, these prefetched tasks are then locked in the database to prevent double execution. One worker node can spawn an arbitrary number of worker threads, which can be optimized towards the exact number of cores available on the machine.

Finally, the results from a simulation are placed into an upload queue, from which they are dumped back into the database. Should the node receive the signal to terminate and have the ability to shut down gracefully, all tasks remaining in the download queue are reopened.

5.6 Writing a *prepare* Stage

Stages in `action` are realized as C++ classes. A prepare stage has to publicly inherit from `PipelineModule` and has to implement the following functions:

- `consumes`: Returns a list of artifacts the stage consumes after being instantiated
- `provides`: Returns a list of artifacts this stage generates after being instantiated
- `execute`: Triggers the stage to perform the configured action
- `to_string`: Prints the configuration of the stage for debug purposes

An artifact list is simply list of artifact names as well as their types.

5.7 Writing a *deploy* Stage

Adding a *deploy* stage to `action` is similarly simple. It has to extend `DeployModule` instead of `PipelineModule`, and implement a similar set of functions. `consumes` and `provides`, and `to_string` behave like before.

- `execute`: Any functionality that can be performed before data is uploaded to the database should be implemented here.

- `commit`: Actually upload the generated setup to the database

This distinction is made to enable dry-run capability without affecting the state of the database. `commit` will only be called when the cluster is intended to be utilized. Data must not be transferred otherwise (and internal database connection is not handed over unless `commit` has been called).

5.8 Writing a Cluster Agent

Writing a new cluster agent for an additional tool requires slightly more effort than adding a new pipeline stage. This section is only a brief overview, as engineering a complete build tool for `act` is not the main focus of this work. Very roughly, the necessary steps are:

1. **Customize the downloader thread.** It pulls new tasks from the database and puts them into the execution queue. In the `actsim` agent, it pulls simulator configurations and makes sure required designs are in a design store. The dataset requested from the database must be changed and auxiliary artifact requirements need to be added.
2. **Change the task object.** It holds the information required for performing the task. This is what is created by the downloader thread and added to the queue.
3. **Adjust the worker process.** When a worker thread executes a task, it forks and executes the desired `act` tool in a separate process. This is a workaround for the current difficulty of calling `act` tools programmatically.
4. **Adjust the upload thread.** Similar to the download thread, the upload thread must be adjusted to upload the correct data to the corresponding table.

CHAPTER 6

Fault-Injection Framework

This chapter talks about the changes made to `actsim` to support our simulations, as well as the current state of our injection generation built into `action`.

6.1 Definitions

As definitions vary in this field, we want to clearly define some terms first.

6.1.1 Single Event Transient / Single Event Upset

Single event models categorize the illegal spontaneous change of bit values due to a passing particle or ionizing radiation. We make a clear distinction between Single Event Transient (SET) and Single Event Upset (SEU), depending on their location in the circuit, though it is of note that SEU is in some literature used for both types of errors. For this reason, we use the definitions made in [Alt13]:

We use Single Event Transient to refer to a temporary pulse or glitch in a logic circuit. This transient pulse may propagate through the circuit, but it does not necessarily get captured and stored in a memory element or register.

We use Single Event Upset to refer to a bitflip in memory, where one or more bits change state regardless of legality or write access.

Both of these error types are soft; this means they only cause a state change, ho hardware damage occurs, the fault is correctable. Note that these definitions differ slightly from the official JEDEC definitions.
This investigation mainly focuses on SET. [Alt13] does not limit SET to a single bit; we limit the scope of the model, as we will only look at cases where a single bit is affected.

Transferring this definition over to simulation gives us the following model:
A set of pull-up or pull-down transistors representing a logic gate is referred to as a *node*.

A node can fan out to one or more other nodes. An SET is simulated by forcing the output value of a node to a given state. If a real-world particle were to hit within the bounds of a node abstraction, two cases can occur:

1. **The output value is not affected:** This is represented by the absence of an SET injection.
2. **The output value is affected:** This is represented by the node output being forced to a given value.

Additionally, we assume a particle hit can affect the perceived value of leaf nodes differently. Figure 6.1 shows different leaf configurations. 2-forks can always be represented in our limited SET model (Figure 6.1a). If a node has more than two leafs, there are different cases:

1. **Only one leaf receives a different value and thus evaluates to an incorrect result** (Figure 6.1b)**.** This can be represented as an SET on the affected leaf node.
2. **More than one leaf receives a different value and thus evaluates to an incorrect result** (Figure 6.1c)**.** This can be represented as an SET on all affected leaf nodes. This is outside our limitation to 1 SET and thus will not be investigated here.

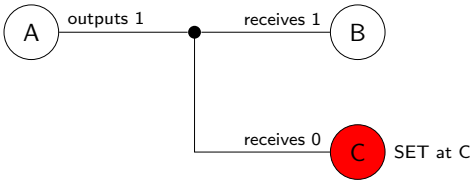
We thus argue that our simulation representation is complete within our limited SET model.

6.1.2 Pipeline Load Factor

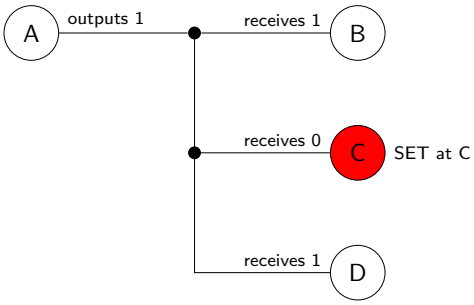
Pipeline Load Factor (PLF) is a measure of how busy the pipeline is. In contrast to [Beh21], we define PLF as the percentage of populated buffers within a pipeline. Consequently, we limit PLF to a value between 0 (all buffers empty) and 1 (all buffers full).

We find PLF an only tangentially useful metric. To show why, let the pipeline in Figure 6.2 fill up the buffers on the upper branch and then bypasses all further tokens on input A directly to its output. Only a certain magic number on input B triggers the saved tokens to be released to the output. Since we can make hitting the magic number by coincidence as unlikely as wel want, the pipeline has a PLF of 50% virtually most of the time (assuming uniformly random input data). While this example is rather artificial, a CPU's reorder buffer might produce a similar behavior.

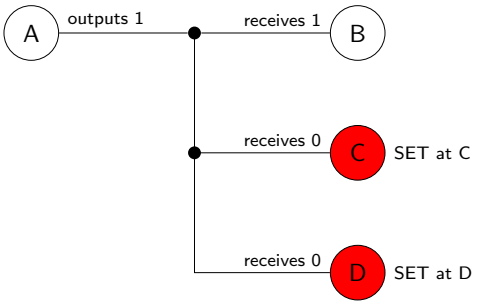
For this reason, we think PLF to be somewhat interesting, but ultimately not of great use when trying to validate a broader set of circuits. We think it much more important



(a) Node with 2-fork



(b) Node with 3-fork, only one leaf receives the transient



(c) Node with 3-fork, two leafs receive the transient

Figure 6.1: Fanout configurations of nodes and representation in model

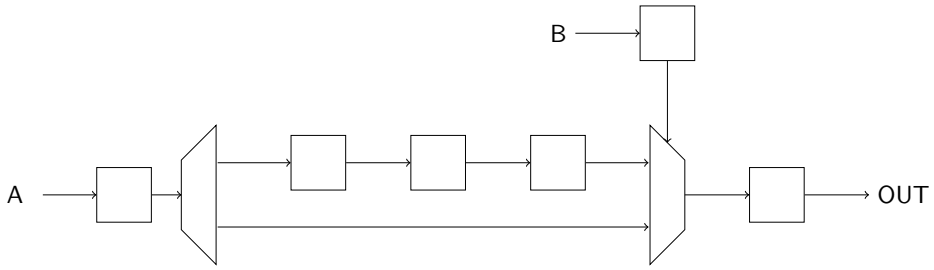


Figure 6.2: Pipeline without useful PLF

to model the input and output latency of the token source and sink realistically for the desired application.

6.1.3 Steady State

A similar argument can be made for steady state. We reuse the pipeline from Figure 6.2. This time, let the control input be only 1 bit wide. Whether we define all buffers filled, all buffers empty, or half the buffers full as the steady state is arbitrary. [Beh21] cites quantitative experiments for a an input of two tokens as their assumed steady state. If we define *all buffers filled* as the steady state of the pipeline in Figure 6.2, it is impossible to reach this state within 2 tokens.

We propose exposing the number of warmup tokens to the testing engineer, as this value is strongly dependent on the DUT.

6.2 Integration into action

Injection generation is loaded into `action` as a testcase generation engine during the prepare stage. The setup allows for very simple replacement of this component for future investigation as well as use of `action` for different purposes. As of now, the test injection engine performs very similar operations to [Beh21]. A list of signals is handed to the generation engine and injections are randomly generated both in value as well as in time. We currently use a simple factor as insertion multiplier per signal instead of [Beh21]'s average insertion density, as some oddities were encountered during the initial evaluation phase. We plan to replace this with a more robust failure space estimation model based on the coupon collector's problem in the future.

The tool then emits these generated testcases as a test configuration artifact. A full demo configuration for a run can be seen in Listing 6.1.

```
input_artifacts:
  - name: design_file
    type: act
    source: file
    path: test.act
  - name: victim_list
    type: siglist
    source: file
    path: victims.txt

prepare:
  - module: testcase_generation
    inputs:
        - victims: victim_list
```

```
    outputs:
      tests: injection_tests
    generator: naive-set-injection
    victim-iterations: 10
    victim-mode: random
    victim-coverage: 50
    injection-windows:
        - begin: 10000
          end: 20000
    injection-duration: 10
    inject-undefined: true
    random-seed: 1234567

deploy:
  - module: actsim
    inputs:
      design_file: design_file
      sim_configs: injection_tests
    outputs:
      sim_outputs: sim_results
    top: test
```

Listing 6.1: SET injection run configuration

CHAPTER 7

Simulation

We originally intended to write a more robust framework based on `prsim`. We however instead shifted our development to `actsim`, which is a much newer simulator and also part of the `act` toolchain. `actsim` can simulate both PRS as well as CHP designs and can even handle mixed designs. This enables us to simulate only the DUT at gate level and avoid simulation overhead for all other components. Additionally, `actsim` is able to perform simulation of a hierarchical designs and does not require the logic to be flattened first.

`actsim` however lacked support for the fault injection capability required for our project. In addition, there was little support logic provided with the simulator to enable easy assembly of a simulation harness.

7.1 Additions to the Simulator

7.1.1 Single Event Transient

We added the capability to trigger a Single Event Transient (SET) on a PRS node within the loaded design. The command syntax is shown in Listing 7.1.

```
sevt <name> 0|1|X <start-delay> <dur>
```

Listing 7.1: Command to schedule an SET event

It blocks further state changes of the given PRS node and sets the output to the forced value. Changes to inputs are still evaluated in the background but not observed while the transient event persists. When the SET ends, the currently hidden value of the node is restored to its output. If the node has been marked by a `watch` command, these changes are reported to the terminal accordingly.

7.1.2 Single Event Delay

We also chose to implement an additional command which manually overrides the delay for the next event on the targeted node. Single Event Delays (SEDs) are a subset of SETs. Two cases present themselves:

1. **The delay time injected by the SED is longer than the nominal delay.** In this case, an SED of length Δ is equivalent to an SET of length $\Delta + \varepsilon$, where the SET starts ε before the next signal transition and has the same logic value as the current state of the rail. In both cases, the signal transition on the output is delayed by Δ.
2. **The delay time injected by the SED is shorter than the nominal delay.** In this case, an SED of length $\nu - \Delta$, where ν is the nominal delay of the node, is equivalent to an SET of length $\Delta + \varepsilon$, where the SET starts at $\nu - \Delta$ after the signal transition and has the value of the node after the transition.

In consequence, delaying a signal transition enough might hide the transition altogether, as the evaluated value of the node changes back before the output can observe the suppressed transitions.

We have as of yet not employed this feature in our fault injection engine, but feel it will aid us in reducing the number of required injections. SEDs are a lot more targeted than injecting random value faults at arbitrary times. The syntax for injecting an SED can be seen in Listing 7.2.

```
sed <name> <start-delay> <dur>
```

Listing 7.2: Command to schedule an SED event

7.1.3 Local Delay Randomization

Finally, we added a new delay mode to the PRS simulation engine within `actsim`. Previously, `actsim` only supported the following modes:

1. **Delay randomization off.** The node delay is not randomized at all.
2. **Global delay randomization.** The node delay is randomized within the full range of possible values.
3. **Global delay range randomization.** The node delay is randomized based on a global upper and lower bound delays.

We expanded this list by an additional entry to support local delay randomization as presented in DiFit [Beh21]:

4. **Local delay range randomization.** The node delay is randomized based on a per node upper and lower bound delay.

While the simulator now supports this feature, the current PRS syntax in `act` does not yet contain notation of these bounds. We plan on adding this syntax in the near future.

7.2 Simulation library

As we chose to move several automated steps back to manual setup, a strong simulation library for easy testing harness development is required to justify these choices. We rewrote the `actsim` simulation library from scratch to support the necessary features. The goal is to write a harness once per design and reuse it for as many tasks and evaluation tests as possible.

7.2.1 Simulation Components

All components are if not otherwise denoted available within the `sim` namespace and can be used by calling `import sim;`.

Sources

The library provides multiple source processes for different data origins. Available are:

- `source_static`: Simple source which provides a static value as a token infinitely many times.
- `source_sequence`: Provides a sequence of items as tokens. Can be set to repeat the sequence.
- `source_file`: Reads a sequence of data values from a file. Can be set to repeat the sequence.

Additionally, the namespace `sim::random` provides additional random data sources:

- `source_simple`: Outputs a sequence of random values over the entire available integer range.
- `source_range`: Outputs a sequence of random values within an upper and lower bound.

All sources are available with a single output as well as multiple outputs, replicating their tokens over all of them. Additionally, each single- and multi-ended source is shipped with an `_en` version, exposing an enable signal.

Sinks

The library provides multiple sink processes for different data destinations. Available are:

- `sink`: Simple token sink which can print the received values to the terminal.
- `sink_file`: Saves the received values to an output file.

All sinks are available with an `_en` version, exposing an enable signal.

Scoreboards

Scoreboards can be found within the `sim::scoreboard` namespace. Available are:

- `lockstep`: Assumes number of tokens consumed per input channel as well as generated per output channel to be identical and in the same order for DUT and model.
- `deterministic`: Assumes number of tokens generated per output channel to be identical and in the same oder for DUT and model.
- `generic`: Makes no assumptions, receives information about test success or fail from external channel.
- `input_logger`: Used to log inputs in identical output format if not using lockstep.

Utility

Collection of miscellaneous processes used for validation. Available are:

- `logger`: Log passing tokens to terminal without introducing slack on the pipeline.
- `logger_file`: Log passing tokens to file without introducing slack on the pipeline.
- `buffer`: Infinite capacity buffer. Useful to decouple DUT from potential timing influence of the simulation harness.
- `splitter`: Clone incoming tokens onto multiple output channels without introducing slack into the pipeline.

7.2.2 Example Testing Harness

A very simple sample simulation harness for a theoretical serial adder can be seen in Listing 7.3. The basic approach is similar to Universal Verfication Methodology (UVM) in SystemVerilog.

```
import module1;
import sim;

defproc test ()
{
    // configuration parameters
    pint D_WIDTH = 8;
    pint NUM_D = 4;
    pint data[NUM_D];
    data = {0, 1, 2, 3};
    pint res[4];
    res = {0, 2, 4, 6};

    // instantiations
    module1::adders::serial_adder<D_WIDTH> add;

    // this example only uses sources and sinks with locally
       ↪ defined sequences
    sim::source_sequence_multi<D_WIDTH, 2, NUM_D, data, false,
       ↪ 1, false> src_i1;
    sim::source_sequence_multi<D_WIDTH, 2, NUM_D, data, false,
       ↪ 2, false> src_i2;
    sim::source_sequence<D_WIDTH, NUM_D, res, false, 0, false>
       ↪ oracle;

    sim::scoreboard::lockstep<D_WIDTH, 2, 1, 0, true> sb;

    // connections

    add.IN1 = src_i1.O[0];
    add.IN2 = src_i2.O[0];

    sb.IN[0] = src_i1.O[1];
    sb.IN[1] = src_i2.O[1];
    sb.OUT_D[0] = add.OUT;
    sb.OUT_M[0] = oracle.O;
```

```
}
```

Listing 7.3: Simple testing harness for adder using the new simulation library

CHAPTER 8

Future Work

While a lot of the targets for this project were hit, a lot of work still has to be done. `action` in its current form is a first sketch of its targeted functionality. As already mention, this limit in scope was intentionally made in order to somehow fit its development into the scope of this project. It also opens up opportunity for a lot of future work.

8.1 Refinement of Number of Insertions

We have as of yet not been able to refine estimating the number of insertions required to find all possible failure states. While we have made first strides to enable fast iteration of injection engines to create the tools to answer this question, we have as of yet not been able to find an improved answer over previous attempts.

For future work, we have identified several possible leverage points:

When looking at a handshaking protocol, there is only a finite amount of time windows in which a circuit is susceptible to soft error due to a propagated SET [SFS23]. In addition, the number of possible failure modes is directly dependent on how many nodes are influenced by the node experiencing the transient. For this reason, we believe a useful heuristic of failure mode estimation to be dependent on fanout. Using this heuristic, we believe it possible to create a more efficient injection plan.

8.2 Engine and Data Quality

Further improvements can be made to the simulator. Currently, only static value injections are possible, where the clamped value might be identical to the steady state of the signal - resulting in no signal level change happening in the circuit. An interesting subset of faults might be glitch injection, where current value of the node is flipped

instead. This subclass of faults can only inject transitions and not suppress them. This might be of specific interest in conjunction with the new `sed` command of `actsim`.

We are also planning on adding Python bindings to simplify implementation of future test-generators.

8.3 Tool Setup

`action` can still be massively improved. In our future implementation efforts we plan to introduce a controller server and change the worker nodes from a pull to a push architecture using a protocol like gRPC, where the controller deploys tasks to the nodes instead of the workers fetching them from a database. This should significantly improve the robustness of the setup against technical failure and thus make it ready for actual production deployment.

In addition, we plan on moving the cluster over to a commercial clustering solution like Kubernetes, to support dynamic up- and down-scaling, as well as ease the implementation of new worker units.

More data sources are planned for artifact loading, such as directly pulling from a network hosted file.

We want to enable snapshotting, where interim stages of the local and remote pipeline progress can be exported into a coherent data structure.

There are also efforts to expand the simulation library to include constrained random testing. We want to enable action to support this feature and have multiple nodes work toward a shared coverage model.

Finally, we want to support better tool integration. `action` in its final form is intended to be a comprehensive build system for the `act` toolchain. Thus integration of all relevant tools as well as easy adaptation for external tools is vital for the usefulness of `action`.

CHAPTER 9

Conclusion

We have presented `action` and its corresponding fault injection framework. Their concepts are loosely based on previous work by Behal [Beh21] and integrate into the `act` toolchain [Man19]. While currently using a similar architecture as DiFit, `action` is the first step at a local/remote build system, solving a broader issue than just distributed fault injection. We have laid out our reasoning for moving certain previously automated features into configuration to enable a broader spectrum of circuits to be tested and shown how to extend the currently very limited `action` to enable more operations to be performed. As of now, we don't feel the project has reached its final form. The net-code is in need of a more robust implementation and we intend to expand `action` to cover most if not all of the `act` toolchain. Further investigation is needed to increase the efficiency of our fault injection engine compared to the attempt made in DiFit. Finally, we have presented our modifications to the `actsim` simulator, as well as the our new simulation library.

List of Figures

Listings

List of Abbreviations

ASIC application specific integrated circuit

CHP Communicating Hardware Processes

CMOS Complementary MOS-FET

DUT design under test

FPGA field-programmable gate array

gRPC gRPC Remote Procedure Calls

OSVVM Open Source VHDL Verification Methodology

PLF Pipeline Load Factor

PRG pseudo-random generator

PRS Production Rule Set

QDI quasi delay-insensitive

SED Single Event Delay

SEU Single Event Upset

SET Single Event Transient

UVM Universal Verfication Methodology

VHDL VHSIC Hardware Definition Language

VLSI Very Large Scale Integration

Bibliography

[Alt13] Altera. „Introduction to Single-Event Upsets“. In: *WP-01206-1.0* (Sept. 2013).

[Beh21] Patrick Behal. „Quantitative Comparison of the Sensitivity of Delay-Insensitive Design Templates to Transient Faults“. Thesis. Technische Universität Wien, 2021. DOI: 10.34726/hss.2021.81601. (Visited on 05/09/2024).

[Dav+14] Mike Davies et al. „A 72-Port 10G Ethernet Switch/Router Using Quasi-Delay-Insensitive Asynchronous Design“. In: *2014 20th IEEE International Symposium on Asynchronous Circuits and Systems*. May 2014, pp. 103–104. DOI: 10.1109/ASYNC.2014.22.

[DM22] Ruslan Dashkin and Rajit Manohar. „General Approach to Asynchronous Circuits Simulation Using Synchronous FPGAs“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.10 (Oct. 2022), pp. 3452–3465. ISSN: 1937-4151. DOI: 10.1109/TCAD.2021.3131546. (Visited on 05/11/2024).

[Hue22] Florian Ferdinand Huemer. „Contributions to Efficiency and Robustness of Quasi Delay-Insensitive Circuits“. Thesis. Technische Universität Wien, 2022. DOI: 10.34726/hss.2022.107641. (Visited on 05/18/2023).

[Man19] Rajit Manohar. *An Open Source Design Flow for Asynchronous Circuits*. Tech. rep. 2019. Chap. Technical Reports. (Visited on 03/22/2023).

[Mar89] Alain J. Martin. *Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits*. DTIC, 1989. (Visited on 05/12/2024).

[Mar91] Alain J. Martin. *Synthesis of Asynchronous VLSI Circuits*. DTIC, 1991. (Visited on 05/12/2024).

[Mer+14] Paul A. Merolla et al. „A Million Spiking-Neuron Integrated Circuit with a Scalable Communication Network and Interface“. In: *Science* 345.6197 (Aug. 2014), pp. 668–673. DOI: 10.1126/science.1254642. (Visited on 03/22/2023).

[Moa17] Sajjad Moazeni. „High-Frequency Clock Distribution Methods In“. In: (2017).

[SBS10] Mingoo Seok, David Blaauw, and Dennis Sylvester. „Clock Network Design for Ultra-Low Power Applications“. In: *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*. Austin Texas USA: ACM, Aug. 2010, pp. 271–276. ISBN: 978-1-4503-0146-6. DOI: 10.1145/1840845.1840901. (Visited on 05/12/2024).

[Sch22] Martin Schwendinger. „Evaluation of Different Tools for Design and Fault-Injection of Asynchronous Circuits“. Thesis. Wien, 2022. DOI: 10.34726/hss.2022.98624. (Visited on 03/22/2023).

[SFS23] Raghda El Shehaby, Matthias Függer, and Andreas Steininger. „On the Susceptibility of QDI Circuits to Transient Faults“. In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Laure Petrucci and Jeremy Sproston. Cham: Springer Nature Switzerland, 2023, pp. 69–85. ISBN: 978-3-031-42626-1. DOI: 10.1007/978-3-031-42626-1_5.

[Von39] Richard Von Mises. *Über Aufteilungs-Und Besetzungswahrscheinlichkeiten*. na, 1939.