TU WIEN Informatics

Testen von OpenVADL Instruktions Set Simulatoren mittels Co-Simulation

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Informatik

eingereicht von

Benjamin Komar
Matrikelnummer 12223386

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.i.R. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 28. Juni 2026

Benjamin Komar	Andreas Krall

TU WIEN Informatics

Testing OpenVADL Instruction Set Simulators using Co-Simulation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Informatics

by

Benjamin Komar
Registration Number 12223386

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof.i.R. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, June 28, 2026

Benjamin Komar	Andreas Krall

Erklärung zur Verfassung der Arbeit

Benjamin Komar

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

Wien, 28. Juni 2026

Benjamin Komar

Acknowledgements

I would like to thank my supervisor, Prof. Andreas Krall, for allowing me to be part of this interesting project, as well as his advice, feedback, and guidance during the implementation and writing of this thesis.

I would also like to thank the OpenVADL team, especially Johannes Zottele, who continuously reviewed my implementation of the Co-Simulator and gave feedback on it, and Kevin Schlosser, who tried out and integrated my implementation into the testing-framework during his own thesis and also provided valuable feedback.

Kurzfassung

Es ist möglich mittels des OpenVADL Projekts eine Vienna Architecture Description Language (VADL) Spezifikation zu schreiben und daraus einen Instruction Set Simulator (ISS) zu generieren. Zum Prüfen der Korrektheit dieser Simulatoren werden automatisch Tests generiert und das Ergebnis wird schlussendlich mit einem Referenz-ISS verglichen. Die jetzige Implementierung bezüglich des Vergleichs der Simulatoren ist allerdings inakkurat, da nur der Endzustand und auch nur die Registerwerte verglichen werden.

Demnach wurde als Hauptziel dieser Arbeit die Implementierung eines effektiven und effizienten Co-Simulators festgelegt. Insbesondere wird, im Vergleich zum vorherigen Programm, direkt mit der Quick Emulator (QEMU)-API gearbeitet, was Vergleiche auf Instruktions-, Translation Block (TB)- und Speicherzugriffs-Ebene ermöglicht.

Abstract

The OpenVADL project allows the generation of an Instruction Set Simulator (ISS). Automatically generated tests are used to verify its correctness, where the result of the generated ISS is compared with a reference-ISS. However, the current implementation, regarding the comparison of the simulators, is inaccurate, given that only the final state and also only the register-values are compared.

Therefore, the main goal of this thesis is to implement an effective and efficient Co-Simulator. In comparison with the previous implementation, the Co-Simulator makes use of the Quick Emulator (QEMU)-API, which allows comparisons on an instruction-, Translation Block (TB)- and memory-access-level.

Contents

CHAPTER 1

Introduction

The implementation of a CPU is based on an Instruction Set Architecture (ISA) - a specification which defines the instructions that are valid for a CPU that implements it. Due to the complexity of the specification, testing such an implementation thoroughly is important to show its correctness. In terms of testing-strategies, one such option would be to manually write a suite of tests for each instruction and assert the expected result for each test-case. However, this is hardly a viable solution if the goal is to get a meaningful coverage of the ISA in reasonable development time. A better option is the generation of test-cases - in this case instructions - which allow coverage of many different inputs for each instruction using a comparatively short implementation.

The OpenVADL[1] project allows the generation of an Instruction Set Simulator (ISS) - a software-based implementation of an ISA using Vienna Architecture Description Language (VADL) as the Processor Description Language (PDL). In order to test the capabilities and correctness of the description language and the compiler, multiple implementations of real-world ISAs, such as RISC-V, AArch and PPC are currently developed by the OpenVADL team. Testing the generated ISS is done by using test-generation and comparing the behavior to an existing implementation which is assumed to be correct.

In the context of this thesis, a Co-Simulator is implemented. A program that allows simultaneous execution of both ISSs and comparing their states during each execution-step - failing the test if any divergence in behavior is detected between the two implementations.

In Chapter 2 Background an overview of the OpenVADL project, Co-Simulation, ISA and Quick Emulator (QEMU) is given. Chapter 3 Implementation contains the implementation of the Co-Simulator which is then evaluated in terms of accuracy and performance in Chapter 4 Evaluation. Finally, Chapters 5 Related Work and 6 Conclusion list related work and implementations in other systems as well as planned future improvements to the current implementation.

[1] https://github.com/OpenVADL/openvadl

CHAPTER 2

Background

2.1 Instruction Set Architecture

When a program is compiled into machine-code, the compiler makes assumptions about which registers, instructions and general behavior are defined on any machine the program is targeted for. Processors which comply with these assumptions are able to correctly parse and execute these programs. These general assumptions are called the Instruction Set Architecture (ISA), an interface which defines the general behavior of a processor, such as its registers, memory model, instructions or interrupt handling [Deg12]. However, it is irrelevant, for example, how an instruction is implemented inside a processor, as long as it matches the given specification. Therefore, multiple correct implementations of the same ISA are possible. However, these differences in the so-called micro-architecture, or computer organization, of a processor may result in better or worse performance or power usage depending on the programs being run.

Some examples of real-world ISAs are `AArch64`, `RISC-V` or `x86`. Generally, it is also possible to further group different ISAs based on their design philosophy, usually Complex Instruction Set Computer (CISC) (`x86`) or Reduced Instruction Set Computer (RISC) (`AArch64`, `RISC-V`) [Ale92].

2.2 QEMU

"QEMU is a generic open source machine emulator and virtualizer." [abo25] In short, QEMU supports two different kinds of emulation.

System Emulation is able to emulate a full virtual machine including its CPU, memory and other devices such as network cards and USB devices [qem25c, qem25a].

User Mode Emulation only emulates the CPU and expects the host and guest operating systems to be the same. Therefore, this mode allows executing programs compiled for a

different CPU architecture to be run on the host while system call translation, POSIX signal handling and threading are done by QEMU [qem25d].

In order to emulate the instructions of the CPU, one of multiple hardware-based hypervisors (also known as accelerators in QEMU) such as KVM, Xen or Windows Hypervisor Platform can be chosen [qem25c]. Notably, QEMU also offers a software-based accelerator known as the Tiny Code Generator (TCG). Besides supporting a larger set of host operating systems and architectures [qem25c], QEMU also provides a plugin interface for the TCG backend, making it possible to dynamically link code to a running QEMU process and subscribe to different internal events such as the translation and execution of individual instructions [qem26a]. To improve performance, the TCG also introduces so-called Translation Block (TB)s, which define groups of translated instructions that assume a given CPU state [qem26b]. Using TBs together with **Direct block chaining** allows the execution of many translated instructions with less overhead per executed instruction.

Finally, QEMU also provides a suite of tools[1] as well as remote debugging from the host system via gdb using the gdbstub [qem25b].

2.3 OpenVADL

OpenVADL offers a new, open-source implementation of VADL [FHH+25]. By specifying the ISA with VADL, it is possible to automatically generate a QEMU based ISS. Defining the ISA of a processor is done inside the `instruction set architecture` definition of the specification. Listing 2.1 gives an example specification which simply defines a program counter and a register file with 32 registers each 64-bit large.

```
1 instruction set architecture ISA = {
2     program counter PC: Bits<64>
3     register S: Bits<5> -> Bits<64>
4 }
```

Listing 2.1: Example VADL ISA

If given a proper specification, OpenVADL then generates a custom QEMU target which serves as the ISS.

2.4 Software Testing

In software development, automated testing is used to improve the quality of the software either by assisting in finding bugs or proving their absence. When writing tests, many

[1] https://qemu-stsquad.readthedocs.io/en/latest/tools/index.html

different approaches can be taken, each with different benefits and tradeoffs. Usually, tests can be categorized by different factors.

The scope or level of the test, such as Unit-Tests, Integration Test or System Tests, defines whether a test covers only a small module, multiple modules or even the whole application [SM17, Uma23].

The testing methodology groups tests based on how much the test-code knows about and interacts with the application code. For example, as described in [SM17], Black Box Testing implies that no application code internals are known during the execution of the test and a test result only depends on the output of the executed code. On the other hand, a White Box test has full knowledge over the application code and allows for tests such as Decision, Branch or Path coverage tests.

Additionally, tests may also be generated. Usually, a handwritten test with predefined inputs can be validated by comparing the output of the tested code with some expectation. However, a problem arises when the test, i.e., its inputs, are randomly generated: It is now more difficult to provide a correct expectation since the input is no longer fixed. This problem of defining a suitable system that correctly verifies the output of a program is commonly known as the "Test Oracle Problem", where the "Test Oracle" is this aforementioned system [Che]. Property-Based testing may be used in case a Test Oracle can be fully derived from the inputs given that the input space is known [tes26c]. For example, a test using property-based testing might check whether the addition of any number with zero always results in the same number. However, defining an exhaustive set of properties for a larger application might be difficult. Another option might be to compare the output of the program to other existing implementations. This method is usually called Differential Testing where a difference in outputs is usually interpreted as a bug which causes a test failure [ES07]. Metamorphic testing can be used in case the generation of a Test Oracle is not feasible, meaning that the correctness of the output of the program cannot be verified and uses properties based on how the output should change given a change of the input [tes26b]. For example, given a function that computes the length of a string and some generated input: If any character is appended to the input, then the length of the string should increase by one.

Finally, tests can also be categorized based on their soundness (i.e., always reporting a test failure if a bug exists) and completeness (i.e., not reporting a test failure in case the program is correct) [Bug22]. Sound testing systems, such as Symbolic Execution can be used to prove the absence of bugs, whereas unsound tests only assist in finding bugs but never prove complete correctness. Systems like Symbolic Execution, as described in [tes26a], try to explore every possible path of a given program and use SMT solvers to verify whether a counter-example for an assertion is available. However, an exhaustive exploration of all possible paths is often difficult or even impossible - for example, in the case of loops.

2.5 Co-Simulation

Co-Simulation is a process where two or more different systems are executed at the same time, and data may be transferred between systems using Interprocess Communication (IPC) at predefined intervals or events, also often described as a simulation or time step [GHK17, Fri11]. A coordinator can also be introduced for these Co-Simulations which synchronizes the simulated systems for each time step and exchanges any necessary data between them, ensuring that all systems run in so-called lockstep [BCWS11].

More formally, for Co-Simulation a so-called Simulation Unit is defined and represented by the tuple

$$\langle S_c, U_c, Y_c, \text{set}_c, \text{get}_c, \text{step}_c \rangle \tag{2.1}$$

which gives the state space, inputs, outputs, information transfer and simulation step of each simulated system c [HGP+21].

With regard to this thesis, where Co-Simulation is used for testing, it can be seen as a form of Differential Testing as described in the Section 2.4 Software Testing. However, Differential Testing usually only compares the end-result of a computation and does not require that both systems run simultaneously. In this case, executing both simulations in parallel is required or at least preferred, since separate execution would require storing every state snapshot during the complete execution of the simulation, which is likely not feasible due to space constraints.

2.6 Interprocess Communication

IPC is required whenever two or more different, isolated processes need to communicate with each other. Here, operating systems usually provide mechanisms and system calls to instantiate and use structures for IPC, in particular, this section will go over POSIX.1-2008 implementations for shared memory and pthreads.

2.6.1 Shared Memory

Since the address space of a process is isolated, it is necessary to define and allocate a separate memory region where access from multiple processes is permitted. POSIX-compliant systems can use `shm_open` and `shm_unlink` to create, open or delete a shared memory object [pos26b].

Other forms of data-transfer between processes are available, such as pipes or TCP sockets. However, using shared memory seems to be best in terms of read/write performance [VJ15].

2.6.2 pthreads

pthreads offers a multitude of facilities to communicate between different threads and processes. In particular, synchronization primitives such as mutexes and condition variables are provided to suspend and notify threads or processes that are waiting for a condition to be met [pos26a].

CHAPTER 3

Implementation

This chapter describes the general implementation and behavior of the Co-Simulator, as well as explain in which cases a more relaxed comparison between the Clients is done to provide more useful test-results. The first section gives a broad overview of the architecture of the Co-Simulator and the environment in which it is used. The subsequent chapters give more detailed explanations of each of those components.

3.1 Architecture

Figure 3.1 shows an overview of the Co-Simulation architecture and how it is integrated into the OpenVADL tests.

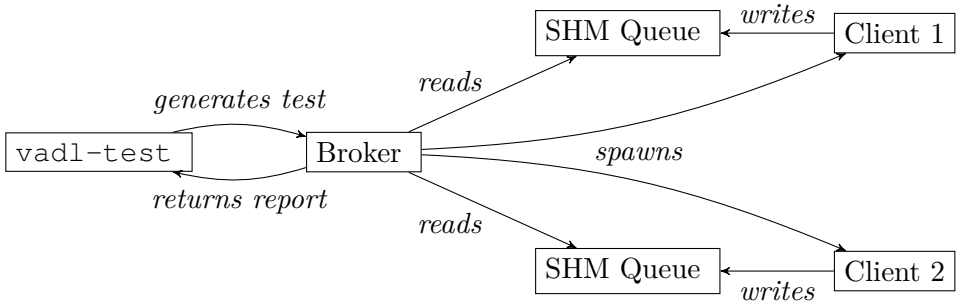

Figure 3.1: Co-Simulation Architecture

The Co-Simulator itself consists of two components: The Co-Simulation `Broker`, and the Co-Simulation `Client`. The Broker is essentially the entrypoint of the Co-Simulation, starting it will read from a provided configuration file and prepare an appropriate environment for the Clients. The `SHM Queue` partially refers to this environment, where the Broker initializes a shared-memory object for each Client to send their state. The

Broker then starts the Clients using a Co-Simulation plugin and periodically collects and compares their states. It then finally generates a test-report to the configured location. In OpenVADL, the Broker is integrated into the existing Java testing-architecture which automatically generates test-cases that are then used as input for the Co-Simulation.

3.2 Co-Simulation Client

The Clients need to provide the Broker with their current state at every execution-step such that the Broker can compare those states to decide whether the Clients behave equivalently or not. In order to retrieve the state, which in this case are the values of all registers as well as the bytes written to and read from memory, the QEMU Plugin API is used. While other options, such as the QEMU Monitor Protocol (QMP), are available, using a compiled plugin is preferred to avoid serialization and deserialization of the data, as well as the possibility for more fine-grained locking to increase performance. Using the plugin is done by simply passing the compiled shared object to the QEMU Client as an argument, causing it to be dynamically linked at startup. Once linked, the `qemu_plugin_install` function is called as the entrypoint of the plugin-code.

3.2.1 Register callbacks

The API provides multiple register-callback functions which accept a function-pointer that is called at different events. `qemu_plugin_register_vcpu_tb_trans_cb`, for example, calls the provided function whenever a new TB is translated. When called, the callback receives an opaque handle to the TB, which allows querying for general information of the TB (e.g., the number of instructions) as well as iterating over each instruction of the TB. It is now possible to define further callbacks inside the translation-callback. Depending on the configuration, either a single callback is registered when the TB is executed or two callbacks are added for each instruction inside the TB. The first calls a function on execution of the instruction, while the second is only called if the instruction accesses memory. While registering two callbacks for each instruction is obviously much slower, it also allows precise error reporting since a divergence will always be detected at the instruction it occurred.

3.2.2 Tracking register-state

In order to request the state of each register inside the plugin it is first necessary to prepare an array of handles for each register that should be tracked. This is done by registering another callback at the entrypoint of the plugin using `qemu_plugin_register_vcpu_init_cb` which is called at initialization of any given vCPU. Inside the callback it is safe to use the `qemu_plugin_get_registers` function to iterate over each register. Listing 3.1 shows the relevant parts of the implementation.

```c
QEMU_PLUGIN_EXPORT int qemu_plugin_install(/*...*/) {
    // Register the `vcpu_init` function at startup
    qemu_plugin_register_vcpu_init_cb(id, vcpu_init);
    return 0;
}

static void vcpu_init(qemu_plugin_id_t id, unsigned int vcpu_index) {
  // Store the register handles for the vCPU which has just been
      initialized in a global array of vCPUs
  CPU *c = get_cpu(vcpu_index);
  c->registers = registers_init(vcpu_index);
}


static GPtrArray *registers_init(int vcpu_index) {
  GPtrArray *registers = g_ptr_array_new();

  // Query all registers of the current vCPU using the QEMU Plugin API
  g_autoptr(GArray) reg_list = qemu_plugin_get_registers();

  for (int r = 0; r < reg_list->len; r++) {
    // Store the handle of the register to query its state later
  }

  return registers;
}
```

Listing 3.1: Collect register handles

Given this setup, it is now possible to request the contents of any register using the `qemu_plugin_read_register` function. Notably, this function should only be called with a given register handle, if read-access to the register was previously requested. Setting this request is done when registering a callback for execution of either a TB, an instruction or a memory access by providing the `QEMU_PLUGIN_CB_R_REGS` flag. Listing 3.2 again shows the relevant parts of the implementation.

```
// `vcpu_tb_trans` is called whenever a TB is translated
static void vcpu_tb_trans(/*...*/, struct qemu_plugin_tb *tb) {
    size_t insns = qemu_plugin_tb_n_insns(tb);
    for (int i = 0; i < insns; i++) {
      struct qemu_plugin_insn *insn = qemu_plugin_tb_get_insn(tb, i);
      // Register a callback to `vcpu_insn_exec` for each instruction
          and set the flag to allow register-reads
      qemu_plugin_register_vcpu_insn_exec_cb(
          insn, vcpu_insn_exec, QEMU_PLUGIN_CB_R_REGS, tbinsn_info);
    }
}

// Called inside of `vcpu_insn_exec`
static SHMCPU get_cpu_state(unsigned int cpu_index) {
  CPU *c = get_cpu(cpu_index);
  for (int reg_idx = 0; reg_idx < c->registers->len; reg_idx++) {
    SHMRegister shm_reg = {};
    // Access the prepared register handles
    Register *reg = c->registers->pdata[reg_idx];

    // Read the bytes of the register into an array-buffer, returning
        the size of the register
    shm_reg.size = qemu_plugin_read_register(reg->handle, buf);

    // Save the current state of the register
    shm_cpu.registers[reg_idx] = shm_reg;
  }
  return shm_cpu;
}
```

Listing 3.2: Read registers

3.2.3 Tracking memory accesses

Tracking reads and writes on memory is very simple using the API since no setup code to manage handles needs to be written. The `qemu_plugin_register_vcpu_mem_cb` function is simply added next to the `qemu_plugin_register_vcpu_insn_exec_cb` function, using the `QEMU_PLUGIN_MEM_RW` flag to indicate that both reads and writes should be monitored. `qemu_plugin_mem_get_value` is then used inside the callback to retrieve a struct containing the size, type and bytes of the memory access.

However, one exception to this simple procedure has to be made since two QEMU Clients might generate different memory calls on the same instruction. For example, the upstream Client might generate a single call to memory to write all 128 bits while the generated VADL-ISS Client might generate two calls for the higher and lower 64 bits of the store instruction respectively. A concrete example for this case is the `stp` instruction of the AArch64 architecture which stores the values of two 64-bit registers in a consecutive 128-bit block of memory. Therefore, in order to handle this potential scenario, the Client

does not immediately write the memory access to the Broker but stores it and checks whether the next call is another memory access which is directly adjacent in memory to the previous one. If both memory accesses are adjacent, the Client reports only one memory access to the Broker containing a merged version of the original two accesses.

3.2.4 Synchronizing Clients on Translation Blocks

As mentioned in 3.2.1 Register callbacks, it is possible to only query the state of the Client once for every executed TB. However, two ISA-compliant Clients might still create different sets of TBs given the same program. Generally, a test should still pass even if the TBs are generated differently, given that the behavior of both Clients is otherwise the same. This means that a common synchronization point has to be chosen. Since QEMU forces a TB to end on any branch instruction, it is possible to assume that two Clients are synchronized whenever a jump is detected. This, of course, further increases the number of instructions that may be executed even after a divergence occurred. The implementation for detecting a jump can be seen in Listing 3.3.

```
1 static int64_t insns_sum_collect = 0;
2
3 // if the start-pc + the offset of the executed instructions does not
    equal
4 // the new pc, then a jump has occurred
5 inline static bool is_jump(TBInfo *tb_info) {
6   return tb_info_collect.pc + insns_sum_collect != tb_info->pc;
7 }
```

Listing 3.3: Detecting a jump

As seen in Listing 3.3, in order to properly synchronize and compare both Clients, it is necessary to collect all instructions across multiple TBs until a jump is found. Only then the data is sent to the Co-Simulation Broker and the buffer is cleared.

3.3 Co-Simulation Broker

3.3.1 Interprocess-Communication

As shown in 3.1 Architecture, the Broker spawns and manages the Co-Simulation Client processes and reports whether a given test has passed or failed. In order to properly run and retrieve any messages from the Clients, the Broker has to set up a shared-memory location for each of the Clients to write to. More specifically, the implementation uses a fixed-size ring-buffer that is fully allocated at the start of the test. This allows a Client to write new messages into the buffer while the Broker simultaneously reads from it without allocating further memory. Also, the current number of unread elements inside the buffer is stored in a shared integer `count` and the `pthread` API is used to signal to

the Broker and Client when a new read or write is available. Parts of the implementation for reading from the buffer is shown in Listing 3.4.

```
pub fn start_read(&mut self) -> Result<Option<&BrokerSHMData>> {
    let waitres = self
        .rb_mutex
        .timedwait(
            /* timeout */,
            || { /* buffer available condition */ }
        );

    match waitres {
        Ok(res) => match res {
            crate::ipc::sem::TimedWaitState::Timeout => /* ... */,
            crate::ipc::sem::TimedWaitState::Success => /* ... */,
        },
        Err(err) => /* ... */,
    }

    let idx = self.ring_idx(self.read_idx);
    let elem = &self.data[idx];

    Ok(Some(elem))
}
```

Listing 3.4: Reading from the ring-buffer

Table 3.1 lists the different kinds of results which are returned by `start_read` and how they are interpreted by the Broker.

Result	Description
`Err(...)`	Waiting for an available read resulted in an error. This usually indicates a crash of the Client, which always results in a test-failure.
`Ok(None)`	Waiting for an available read succeeded, but it did not return any data. This indicates that the Client finished executing. The test will pass if and only if the other Client also returns `Ok(None)` in the same iteration.
`Ok(Some(...))`	The Broker successfully read from the Client and will compare the returned state with the other Client.

Table 3.1: `start_read` return values

3.3.2 Comparing Registers

This section shows how the Broker compares the state over all relevant registers during one iteration of the testing-loop. Generally, the Broker cannot assume that the same set of registers is available in both ISS, especially because one of the two is assumed

to be currently in development. This further implies that the registers are not ordered equivalently in their respective shared-memory location. Next, reporting a test-failure when a register is found that the other Client does not have is not useful since it does not allow testing the currently implemented state of the in-development Client. Therefore, only a subset of all registers is taken into consideration, namely the ones which exist in both Clients. Further exceptions and special cases are configurable, which is discussed more thoroughly in 3.4 Co-Simulation configuration. Assuming that one Client (most likely the in-development Client) always contains a subset and therefore fewer registers than the upstream Client, the Broker iterates over all registers of the in-development Client and searches for the respective upstream register by its name. Once a corresponding register of the other Client has been found, it is possible to store the index to the register to improve the linear search into a constant lookup. Ideally, once a register-pair is selected, comparing them should only involve a byte-wise comparison of their values. However, this does not work when a specific representation of a register is currently not possible. Currently, VADL generated ISS Clients always have a 64-bit representation internally, meaning the test would fail for any comparison with a register that is not 64-bit. Additionally, it might be useful to compare two ISSs with different endianness. Therefore, registers might need to be compared differently if their endianness does not match. Handling this case allows, for example, testing the generated PowerPC Client, which is currently little-endian, with the upstream big-endian version. Given these two constraints, the byte-array representing the value of a register is first converted into a 64-bit integer. Listing 3.5 shows the implementation of this conversion, and Figure 3.2 illustrates the alignment of the bytes.

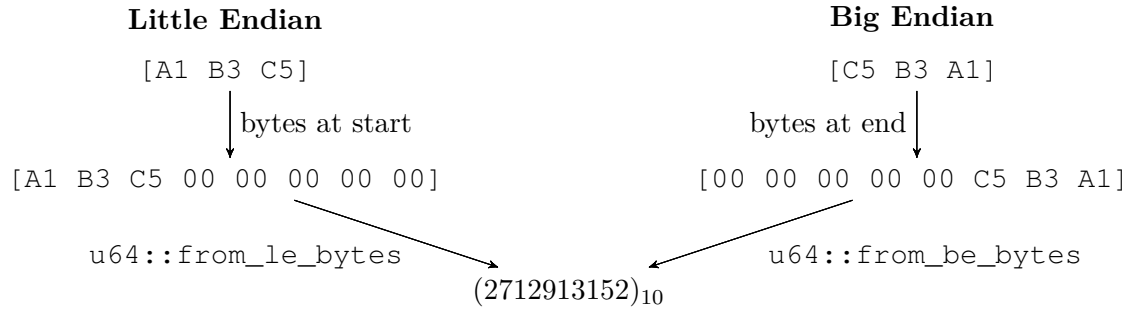

Figure 3.2: Byte-Alignment into a 64-bit integer

```rust
pub fn to_u64(&self, endian: &Endian) -> u64 {
    const BUF_LEN: usize = 8;
    let mut buf: [u8; BUF_LEN] = [0; BUF_LEN];
    match endian {
        Endian::Little => {
            buf[..self.size as usize].copy_from_slice(self.data_slice
                ());
            u64::from_le_bytes(buf)
        },
        Endian::Big => {
            buf[BUF_LEN - self.size as usize..].copy_from_slice(self.
                data_slice());
            u64::from_be_bytes(buf)
        },
    }
}
```

Listing 3.5: Converting a byte-array into a 64-bit integer

Finally, it might also be possible that a single register in one Client is split up into multiple registers in another Client. One such example is in the implementation of the AArch64 ISA. The AArch64 ISA defines the Current Program Status Register (CPSR), containing multiple bit-flags including the `N,Z,C,V` condition flags. However, the VADL implementation of the AArch64 Client has four separate single-bit flag registers and creates a bigger CPSR register only when it is moved into or from a general purpose register. While access to the condition flags is handled correctly, each of those flags is stored inside its own 64-bit register. In order to compare these four separate status registers with the four bits of the CPSR register, a custom mapping has to be defined. The mapping needs to define which bits need to be compared between both registers. For example, it is necessary to compare the first bit of the `NZCV_Z` register with the 31st bit of the `CPSR` register. To properly compare these bits only two instructions are necessary: Masking the relevant bits using a bitmask and then shifting the bits such that the first bit equals the first used bit of the mask. Applying those two operations ensures that the bits are aligned and only the relevant bits are compared. The configuration of this mapping is further explained in 3.4 Co-Simulation configuration.

3.3.3 Test-Report

The Broker does not need to provide any additional information if a test succeeds. On the other hand, a report for a test failure should include enough information such that it is clear where and why a test failed. This means that a test report should include the instruction (or all instructions of a TB) which caused the test-failure, as well as the state before the instruction was executed and the state after. This information allows one to review how the faulty instruction behaved as well as reproduce the test by using the same instruction on the given before-state. Additionally, a descriptive message is added

to any error which should explain where exactly a divergence was found. Listing 3.6 shows one such example given two registers with different values.

```
if reg1val != reg2val {
    let r1name = reg1.mapped_name(config);
    diffs.push(DiffEntry::new(
        format!("cpu[{cpu_index}].registers[{reg_index}].data"),
        vec![reg1.data_slice_fmt(), reg2.data_slice_fmt()],
        format!("different register data for {r1name}"),
    ));
}
```

Listing 3.6: Adding a DiffEntry to the test-report

The integration of the test-report into the testing infrastructure is further explained in 3.5 Integration into OpenVADL tests.

3.4 Co-Simulation configuration

This section gives a brief explanation of the different parts of the configuration that are necessary to properly run the Co-Simulator. The configuration file, written in the Tom's Obvious Minimal Language (TOML) format, can be split up into three parts: The configuration of the QEMU-Clients, the configuration of the testing protocol and the report, and various configurations for logging and tracing of the Co-Simulation.

3.4.1 QEMU configuration

Most importantly, when configuring the QEMU-Clients, a path to the QEMU Co-Simulation plugin, as well as a configuration of how a Client should be started, needs to be given. See Listings 3.7 and 3.8.

```
# General settings for the QEMU plugin system which set up the co-
    simulation
[qemu]
# The path to the compiled Co-Simulation qemu-plugin
plugin="./qemu-setup/build/contrib/plugins/libcosimulation.so"
```

Listing 3.7: QEMU Co-Simulation plugin path

```
# Defines a pair of Clients to test against
[[qemu.clients]]
# Optional: A custom name for a Client
# Will default to the index of the Client in this list
name = "VADL"

# The executable of the ISS
exec = "qemu-system-a64"

# The path to the compiled file to use for testing
# This file will be passed to all Clients
test_exec="./test-execs/embench/edn"

# "bios" | "kernel": Defines where the test-executable is passed to
    when starting the QEMU-Client
pass_test_exec_to = "bios"

# Applies additional arguments to the ISS-executable, e.g. `qemu-
    system-a64 -nographic -d plugin`
additional_args = [ "-nographic", "-d", "plugin" ]

# Skips the first n instructions (i.e., omitted from comparison)
skip_n_instructions = 2

# Whether the Client is big-endian or little-endian
endian = "little"
```

Listing 3.8: QEMU-client configuration

Furthermore, as already mentioned in 3.3.2 Comparing Registers, it must be possible to define custom mappings to tell the Broker which registers should be compared. Since register names may differ between Clients, it is necessary to rename each register of one Client such that its name matches with the correct register of the other Client. In Listing 3.9 an example of such a mapping is given.

```
[qemu.gdb_reg_map]
s0 = "x0"
s1 = "x1"
s2 = "x2"
# ...
# Even though the names of this register already match, an entry can
    be added anyway to include the register for comparison
pc = "pc"
```

Listing 3.9: QEMU register-map

This mapping also instructs the Broker to compare these two registers. For bitwise mappings as discussed at the end of Section 3.3.2 Comparing Registers, another map

needs to be defined.

```
[[qemu.sliced_reg_map]]
client1 = { name = "nzcv_n", slice = [ 0 ] }
client2 = { name = "cpsr", slice = [ 31 ] }

[[qemu.sliced_reg_map]]
client1 = { name = "nzcv_z", slice = [ 0 ] }
client2 = { name = "cpsr", slice = [ 30 ] }

[[qemu.sliced_reg_map]]
client1 = { name = "nzcv_c", slice = [ 0 ] }
client2 = { name = "cpsr", slice = [ 29 ] }

[[qemu.sliced_reg_map]]
client1 = { name = "nzcv_v", slice = [ 0 ] }
client2 = { name = "cpsr", slice = [ 28 ] }

# Note: A range can also be given inside the slice using: `slice = [
    2, 7, [9, 13] ]`, which includes the bits 2, 7, 9, 10, 11, 12, 13
```

Listing 3.10: QEMU sliced-register-map

Listing 3.10 shows the configuration for the `NZCV` bits of the `CPSR` register. Here it is necessary to define multiple mappings to the same `CPSR` register, since each of these bits are stored in a separate register in the VADL-Client. The important part of this mapping is the definition of the `slice` array, indicating which bits should be used for comparison. When reading this configuration, the slice array is converted into a bitmask and an amount of bits to shift right. By default, the number of bits to shift by is equal to the position first used bit. This ensures that, like in the example in Listing 3.10, that those bits are correctly aligned before comparison in case their position differed.

3.4.2 Testing-Protocol configuration

While the previous section defined which registers should be compared, it did not define when or how often they should be compared. As already mentioned in 3.2.1 Register callbacks, it is possible to define the `layer` at which the Co-Simulator operates, i.e., either at an instruction level or at a TB level. Additionally, setting the `layer` to `tb-strict` disables any of the aforementioned synchronization of TBs and therefore enforces that the TBs of both Clients are equivalent.

Given that an ISS might run indefinitely, either expectedly given the program, or due to a bug, multiple configuration options are given to indicate when the Co-Simulator stops. When configuring any of the following exit conditions, a test always passes if no divergence occurred before reaching one of these conditions. The simplest condition is configured by `stop_after_n_instructions`. The Broker will stop once the configured number of instructions have been executed and compared. However, it might be useful to define

an exit-condition based on certain events, such as reaching a certain PC or writing to a memory address. Listing 3.11 shows the configuration for these events. Since the test-binary is always available, it is also possible to reference the PC or memory location via labels instead of raw integer values. The location of these labels is then resolved before the start of the simulation.

```
[testing.exit_condition]
on_address = "0x12345678"
on_label = "cosim_exit"

[testing.exit_condition.mem_write_exit_condition]
on_address = "0x87654321"
on_label = "write_exit",
with_constant_value = "0x123",
```

Listing 3.11: Custom exit-condition flags

Finally, when generating the report, it is also possible to define the output verbosity. Where `verbosity = "short"` returns a custom human-readable format to briefly describe the error, and a value of `"full"` returns a YAML-formatted string containing the complete before- and after-state of both Clients, as well as the instruction that caused the error.

3.5 Integration into OpenVADL tests

The test-system of OpenVADL makes use of random test-generation in order to exhaustively test the instructions of a generated ISS. With regard to ISS tests, the OpenVADL project uses containerized environments for testing using Docker. For each test-run, all dependencies for compiling the QEMU project, as well as compiling the generated assembly files are pulled from a pre-built docker image and the ISS is built once and cached for consecutive access.

Given that the Co-Simulator is written in Rust, it is necessary to include the Rust compiler in the docker image. Also, to improve compilation times, it is possible to run the `cargo fetch` command to download the dependencies when building the image such that they are available locally once the Co-Simulator is being compiled. Listing 3.12 shows the necessary additions to the Dockerfile used in the ISS tests.

```
# download rust for cosim
RUN curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s
    -- -y
ENV PATH=/root/.cargo/bin:$PATH

# download cosim dependencies
COPY vadl-cosim /work/vadl-cosim
WORKDIR /work/vadl-cosim
RUN cargo fetch
```

Listing 3.12: Co-Simulator Docker setup

Finally, once compiled, the Co-Simulator is then executed for each test-case which then writes the result of the test in a separate file. The JUnit tests then read and parse the results, calling `Assertions.fail` in case of a test-failure.

CHAPTER 4

Evaluation

When evaluating the Co-Simulator, two aspects need to be taken into consideration: The accuracy, meaning how many types of bugs can be found using the Co-Simulator, and also its performance. In both cases, the Co-Simulator is compared to a certain baseline.

4.1 Testing Accuracy

To illustrate the accuracy of the Co-Simulator it will be compared to the previously implemented system, which is more akin to Differential Testing as described in 2.4 Software Testing. In short, the previous system executes both ISS in parallel, using a QEMU-plugin that prints the value of each register when an exit-signal is received. Notably, it only returns the state **once** (after executing the full program) and does not return any memory read/write information.

Listing 4.1 gives a very simple example-progam which illustrates how the Co-Simulator is able to detect a difference in behavior while a simple comparison of the end-state is insufficient in doing so:

```
1 add x3, x3, #1
2 sub x3, x3, #1
```

Listing 4.1: Comparing Test Accuracy

Assuming that there is a bug in the generated ISS which does not change the register value when executing the `add` and `sub` instructions, using simple differential testing would not find any divergence since adding and then subtracting 1 from a register "restores" its initial value.

Running this test (using a modified AArch64 VADL specification, which encodes this faulty behavior) on the old test system results, as expected, in a test pass, as can be seen in Listing 4.2. On the other hand, the Co-Simulator detects the bug (see Listing 4.3).

```
[PASS] Finish test case ADD-1-then-SUB-1 in 65.43ms
Total time: 0.087s
```

Listing 4.2: Inaccurate Test pass

```
Cosimulation failed!
Failure at pc = 0x40000004 (1073741828)

The following divergences were found:
- "different register data for x3":
    - In "VADL" the value is "0x0000000000000000"
    - In "UPSTREAM" the value is "0x0100000000000000"

The divergence occurred after the following instructions were executed
    :
- (pc=1073741824): add x3, x3, #1 (0x63040091)
```

Listing 4.3: Accurate Test failure

However, even if the bug is detected, it might still be very difficult to find the faulty instruction in a sufficiently large program. During the implementation of the Co-Simulator, a real bug was introduced in the AArch64 specification which surfaced in only one of the *embench* programs, that are used to measure the performance of the generated ISS. Since these programs are much larger than a simple test-case that is tailored to a specific instruction, it was not feasible to efficiently find the bug only using the difference of the end-state of both ISSs.

Specifically, the AArch64 VADL specification used a macro `MulAddSubLongInstr` to define both signed and unsigned instructions. However, since the type (unsigned or signed) was not used inside the macro, the generated instruction was always using an unsigned implementation.

Listing 4.4 shows that the Co-Simulator not only finds this bug, it also reports exactly which instruction was at fault. This also makes it simple to reproduce this behavior in a much smaller testcase by simply writing a new program that initializes the state accordingly and then executes the instruction that surfaced the bug.

```
Cosimulation failed!
Failure at pc = 0x400024DC (1073751260)

The following divergences were found:
- "different register data for x3":
    - In "VADL" the value is "0x0338B60200ECFFFF"
    - In "UPSTREAM" the value is "0x0338B60200000000"

The divergence occurred after the following instructions were executed:
- (pc=1073751256): smaddl x3, w1, w1, x3 (0x230C219B)
```

Listing 4.4: Embench Co-Simulation failure

4.2 Performance

4.2.1 Setup

The performance of the Co-Simulator (both on the instruction and TB layer) is compared against a "baseline" where the test-program is executed on both ISSs concurrently but without any Co-Simulation. This is done using a simple Bash-Script as shown in Listing 4.5.

The execution time (wall clock time) will be measured using the `perf`[1] tool, executing each instance 50 times and taking the measured average.

The Co-Simulator is compiled in release-mode using the `cargo build --release` command.

In total, three different types of test-programs will be used to evaluate the performance, namely:

1. An empty program (only containing the instructions necessary to exit QEMU), to measure the overhead of simply starting the Co-Simulator
2. A simple loop, that increments a counter and exits at some defined value to measure the actual runtime of the Co-Simulator
3. The embench programs, to test Co-Simulation performance on more real-world programs

[1] https://man7.org/linux/man-pages/man1/perf.1.html

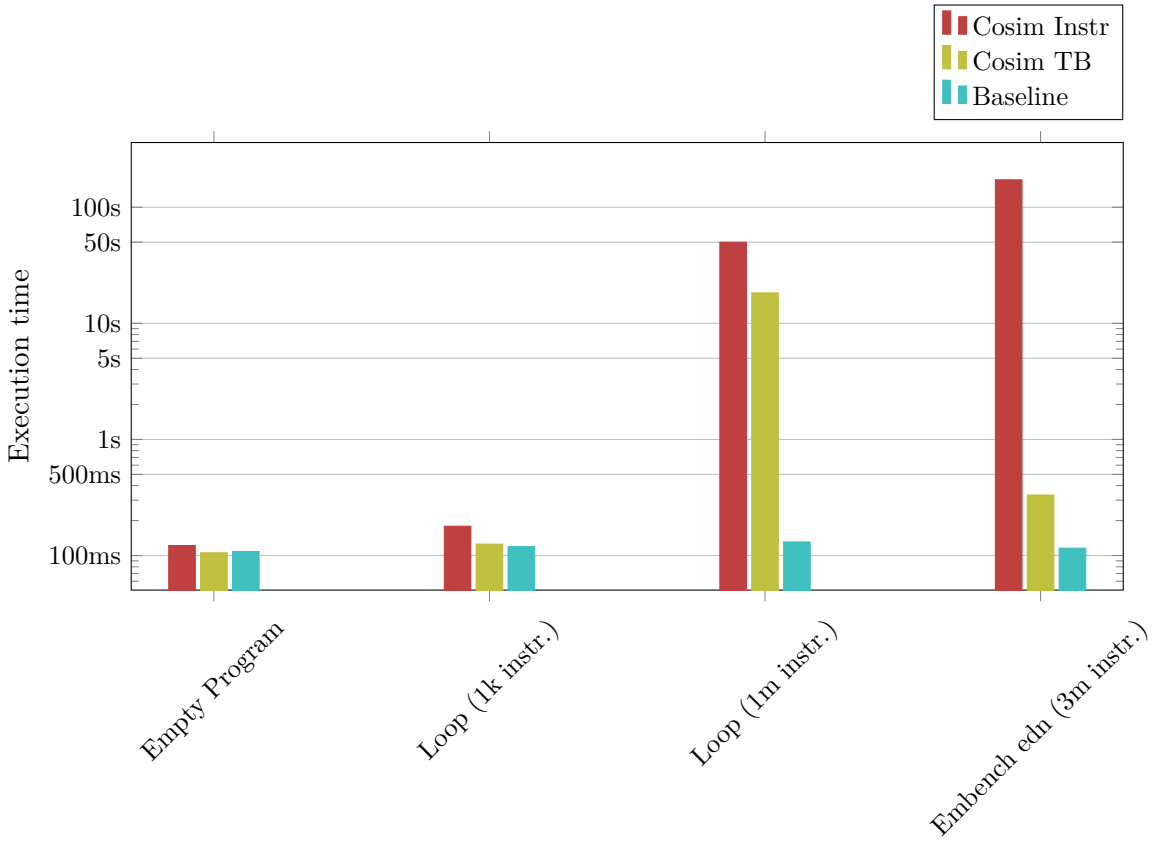

Figure 4.1: Co-Simulator execution time on different test-programs

```
#!/usr/bin/env bash

testprogram="${1}"

qemu-system-a64 -nographic -bios "${testprogram}" &
qemu-system-aarch64 -nographic -M virt -cpu max,sve=off -semihosting -
    kernel "${testprogram}" &

# wait for both iss to finish executing
wait
```

Listing 4.5: Baseline Bash-Script

4.2.2 Results

Notably, the Baseline does not measurably increase with the increase of executed instructions. In fact, a difference in runtime is only measurable after the 100m instructions

mark, reaching an execution performance of roughly 1 billion instructions per second. As illustrated in Figure 4.1, the Co-Simulator is orders of magnitude slower. Especially when comparing the full state after every instruction, the Co-Simulator is slower by a factor of roughly 50000. With regard to the TB comparisons, the potential speedup heavily depends on the input test-program. The graph illustrates this dependency: In the 1m instructions loop, using TB comparisons only nets a performance increase by a factor of 3, given that the loop body (see Listing 4.6) consists of only 3 instructions which are grouped into a TB. However, for programs that contain much larger loop bodies, such as the *edn* program of the Embench suite, the performance increase is also much larger. Here, a speedup by a factor of over 500 was measured, even resulting in a much faster execution time than the simple loop program. Comparing this execution speed with the baseline now indicates a slowdown only by a factor of roughly 110.

The slowdown, especially with regard to comparing individual instructions, is easily explained, considering that after each instruction the following - additional - steps also need to be executed:

1. The Client iterates over each register using the provided QEMU-API, writing the relevant data into memory.
2. The Broker waits until a signal is received by the operating system, indicating that new data can be read.
3. Again, each register is iterated, testing whether it should be compared.
4. If the register should be compared, its bytes need to be copied based on the endianness before comparison.
5. The Broker again sends a signal to wake the Client waiting for the next possible write.

```
1 loop:
2   add x3, x3, #1
3   cmp x3, x4          // x4 is set to 1m
4   b.lt loop
```

Listing 4.6: Loop Body

CHAPTER 5

Related Work

Co-Simulation has also already been successfully implemented in other systems to verify the correctness of ISSs. Goel *et al.* in [GHK17] implemented a x86 ISS written in Lisp, and ACL2 for theorem proving. Using ACL2 in the implementation of the simulator should allow for formal reasoning of x86 programs aided by the ISS. However, the correctness of the x86 simulator has to be verified in order to be able to trust any proofs derived from it. Hence, regular Co-Simulation between the ISS and a real x86 processor was used. Modeling a suitable simulator that is both efficient for Co-Simulation (i.e., executing a x86 program and then comparing its state with a real x86 processor) and provides simple, easy to reason about constructs, posed difficulties during implementation. Techniques such as ACL2's `mbe` function were used to provide two implementations of the same function, one of which was used in the proof system while the other offered an efficient implementation for execution. Co-Simulation with a real processor was done using GDB and Intel's Pin tool for instrumentation.

Co-Simulation is also often used in combination with other techniques that assist in exploring different execution paths of a processor. In [BHD23], Bruns *et al.* used symbolic execution on the instruction and data memory as well as the registers to generate new instructions that are then executed on a RISC-V ISS and Register-Transfer Level (RTL). A list of intentionally inserted bugs was used to validate the effectiveness of this setup; according to the authors, all bugs were found. Similarly, in [KTS+21], Kabylkas *et al.* use a Logic Fuzzer to change the state of a RISC-V RTL during execution, such as changing branch predictor tables or random invalidation of TLB entries. Changing these values that should not affect the result of an execution aided in finding bugs by exploring multiple execution paths using the same programs. The authors reported a list of bugs found in three different RISC-V cores, also showing the effectiveness of their implementation.

As mentioned in Section 2.5 Co-Simulation, the formal definition of simulation units in Co-Simulation define inputs and outputs, meaning that the behavior of one system affects the behavior of another. While this was not needed in this thesis or in the other use-cases described in this section, it is often relevant for co-simulating physical or mechatronic systems such as those described in [Fri11]. Other applications also include Co-Simulation of Smart Grids [LAH11, SSAP14] or cyber-physical systems [AH12, WB13].

CHAPTER 6

Conclusion

The VADL-PPC64 ISS is already solely tested using the new Co-Simulator and is fully integrated into OpenVADL's testing pipeline.

As seen in 4 Evaluation, the new Co-Simulator improves on the old implementation especially with regard to the accuracy of the test-results. Allowing comparisons on a much more granular level, as well as comparing memory access operations. However, it still suffers from a major slowdown on large executables (compared to running the ISS in a standalone setting), especially when comparing at the instruction-level.

6.1 Future Work

To vastly improve the performance of the current Co-Simulator, it is necessary to omit any comparisons of values that cannot change during the execution of an instruction. Since most instructions only write to one or a few registers at a time, implementing this change may speed-up the register-compare algorithm by orders of magnitude. However, this would first require work to encode which instruction affects the values of which registers.

The Co-Simulator currently only compares the generated ISS with another ISS chosen as a reference-model. However, in the future it may be beneficial to be able to run the Co-Simulation against real hardware.

Finally, it may also be useful to extend the usage of the Co-Simulator to RTLs using the Verilator[1] tool.

[1] https://www.veripool.org/verilator/

APPENDIX A

A notion on the correctness of the Co-Simulator

This chapter is not giving a rigorous correctness-proof for the whole Co-Simulator as this would require its own dedicated thesis due to the complexity and size of the program code. The idea of this chapter is to semi-formally show correctness for a specific part of the Co-Simulation algorithm while choosing reasonable assumptions to simplify reasoning over the program code.

Particularly, the goal is to give a "proof" for the following two claims:

1. Given a correct ISS, the Co-Simulator and QEMU-Client **correctly lock and unlock the shared semaphore**.
2. Given a correct ISS, the Co-Simulator and QEMU-Client **never perform an illegal read or write operation on the shared buffer**.

Proving both claims is important to improve confidence in the Co-Simulator. A bug in the Co-Simulator, resulting in an incorrect test-outcome, reduces its effectiveness during testing. Of course, only showing correctness for those two claims does not fully imply the correctness of the whole Co-Simulator. However, both claims are regarding the multi-threading and synchronization part of the algorithm, which are categorically more difficult to test and to debug compared to, for example, the single-threaded register-compare algorithm.

Both "proofs" are making use of the *Kronecker Algebra*, an introduction and example uses can be found in [kro26] and [MB11]. The outline of each proof is:

1. Represent the programs and the resources as a state-machine / Control-Flow Graph (CFG).
2. Convert each CFG into an adjacency matrix.
3. Apply the Kronecker-Product and Kronecker-Sum over the defined matrices.
4. Find the subgraph that is reachable from the new start-node.
5. Analyze the subgraph to show its implied guarantees.

A.1 Representing the programs and resources as CFGs

In total, the following components need to be converted into CFGs:

1. The Co-Simulation Broker (written as R for *Reader*) and a Co-Simulation Client (written as W for *Writer*), representing the programs.
   A valid, as well as an invalid "implementation" will be given later (see A.5.1, A.5.2) for the reader to better demonstrate the case when a resource-misuse occurs.
2. A semaphore (S) and a shared buffer (B), representing the resources

For the CFGs for all the aforementioned components see A.5, A.8, A.9, and A.10. In order to understand what each CFG represents, it is important to understand each state-transition. For a descriptive table for each state-transition see A.1

State-Transition	Description
`p`	Aquire the semaphore-lock
`v`	Release the semaphore-lock
`rp, rv`	semaphore operations initiated by the **r**eader
`wp, wv`	semaphore operations initiated by the **w**riter
`ra`	**r**ead **a**vailable on the shared buffer
`wa`	**w**rite **a**vailable on the shared buffer
`r`	**r**ead from the shared buffer
`w`	**w**rite to the shared buffer

Table A.1: Description of each state-transition used across all CFGs

The simplest CFG is the semaphore, which only has two states (where 0 is the start node) with one state-transition each. The graph can be read as: *Starting from an unlocked semaphore, it can be locked using p and then again unlocked using v.* Notably, it is

impossible (given this representation) to lock the semaphore twice, since there is no state-transition p in state 1. The same holds analogously for unlocking the semaphore twice using v.

The size-1 buffer is similar to the semaphore, however it also includes two additional state-transitions that can be used to check whether a read or write operation is available on the buffer. This is akin to checking if `buffer.count > 0` or `buffer.count < RING_BUFFER_SIZE` inside the real Co-Simulation code.

Finally, both the reader A.5 and writer A.8 are represented. Both make use of the read/write available checks and a "busy-wait" (notice the $0 \to 1 \to 6$ cycle in the CFG) in case their buffer operation is not available. Note that the busy-wait does not fully represent the actual implementation, for an explanation on this, see A.6 Discussing used proof simplifications.

A.2 Convert each CFG into an adjacency matrix

The CFGs need to be converted into adjacency matrices in order to apply the Kronecker operations on them. This is done by representing a CFG using a square $n \times n$ matrix, where n is the number of states. For the converted matrices see A.16, A.19, A.20 and A.21.

Again, the simplest matrix is the semaphore matrix S. The adjacency matrix should be read as:

- If the value $v \neq 0$ is inserted into the cell at row r and column c
- Then v represents a state-transition from state r to state c using v.

In other words, each row in the matrix represents a state and the columns describe to which *next* state to jump to. A 0 in a cell means that no state-transition from the row to the row with the column-index exists.

A.3 Apply the Kronecker-Product and Kronecker-Sum over the defined matrices

Since all relevant parts are now converted into square adjacency matrices, it is now possible to apply the **Kronecker Product** and **Kronecker Sum**. Both result in new adjacency matrices with certain properties.

The **Kronecker Product** is defined [kro26] as

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}$$

Figure A.1: Kronecker Product $\otimes$

where the resulting matrix $C = A \otimes B$ represents all operations (state-transitions) executed **in lockstep** over A and B.

The **Kronecker Sum** is defined [kro26] as

$$A \oplus B = A \oplus I_n + I_m \oplus B$$

Figure A.2: Kronecker Sum $\oplus$
where I_n, I_m are the identity matrices of order n, m respectively
and A is a $m \times m$ and B is a $n \times n$ matrix

where the resulting matrix $C = A \oplus B$ represents all possible **interleavings** over the operations of A and B. An example interleavings graph can be seen in A.12.

However, before calculating the matrix, it is necessary to adapt the definition of the **Kronecker Product** to only include *valid* lockstep state-transitions [MB11]. Informally, this implies that the state of a resource can only change (in lockstep) if a program (here, either the reader or the writer) actually executes the respective operation on the resource. The new (**selective**) Kronecker Product is now adapted by changing the multiplication operator to:

$$c_{i,p,j,q} = \begin{cases} l & \text{if } a_{i,j} = b_{p,q} = l,\ l \in L, \\ 0 & \text{otherwise} \end{cases}$$

Figure A.3: Selective Kronecker Product, L is the set of state-transitions

Given these two operations, it is now possible to calculate the following:

$$F = (R \oplus W) \otimes (S \oplus B_1)$$

Figure A.4: Computing the final adjacency matrix given the programs and their resources

Which effectively computes an adjacency that represents *all possible interleavings of the reader and writer, using all possible interleavings of the semaphore and buffer in lockstep.*

A.4 Find the subgraph that is reachable from the new start-node

The start-node of the CFG F is represented by the first row of the matrix. This is analogous to constructing an adjacency matrix from a CFG. An important observation is that not every state and state-transition that is calculated using the Kronecker operations is actually reachable. Any subgraph that is not connected with the actual start-node must therefore be ignored since this represents an impossible program state. By computing the subgraph that is reachable from node 0 (the start-node), we finally get the CFG in A.13.

A.5 Analyze the subgraph to show its implied guarantees

It is possible to conclude whether the programs correctly interact with the provided resources by searching for *a node with no outgoing edges that is **not** an end-node.* In this case, the programs are endlessly looping, therefore no end-node exists - which means that any such node with no outgoing edges implies a bug.

First, looking at A.13 it can be seen that no such faulty node exists - the described program **is correct**. However, to better illustrate what happens in case a faulty implementation is provided, two separate changes will be made to the reader R. R_b describes an incorrect use of the buffer, where no read-available check is made beforehand. R_s describes an incorrect use of the semaphore, where two consecutive unlocks are performed.

A.5.1 Incorrect buffer usage

For details, see the CFGs at A.6 and the matrix at A.17.

The new graph F can be seen at A.14. Here, the nodes 30 and 42 have no outgoing edges, which implies a faulty resource usage. Note that this does not show where the actual faulty implementation is, it only shows the existence of some bug.

A.5.2 Incorrect semaphore usage

Analogously, for details, see the CFGs at A.7 and the matrix at A.18.

The relevant parts of the new graph F can be seen at A.15. The full graph is omitted since it spans over roughly 100 nodes. Again, here, the nodes $188, 192, 176, 189$ and 193 have no outgoing edges, which again implies a faulty resource usage.

A.6 Discussing used proof simplifications

During this informal proof, two simplifications were used. First, the buffer was limited to a **size-1** buffer. This was solely done to reduce the size of the resulting matrices and CFGs to more easily present it in the thesis. It is easy to extend the buffer to any size, an example for a **size-3** buffer can be seen at A.11.

The second simplification is with regard to the busy-wait. While using a busy-wait is a correct implementation for this program (as demonstrated by this proof), the Broker and Client do not use this as it would unnecessarily put more load on the CPU. Rather, a conditional wait is implemented using `pthread_cond_wait` and `pthread_cond_timedwait`. The proof made use of a much simpler busy-wait due to the complexity of the implementation of such a signaling-based conditional wait. However, it is worth noting that only one ("additional") bug could occur using a conditional wait instead of a busy-wait. The bug is known as the **Lost Wake-Up Problem** which is described in [los26].

Finally, it is worth mentioning that the calculations and visualizations were not done by hand, given that the matrices reached sizes up to 224×224. Both were done using a python script. The script can be seen at A.1.

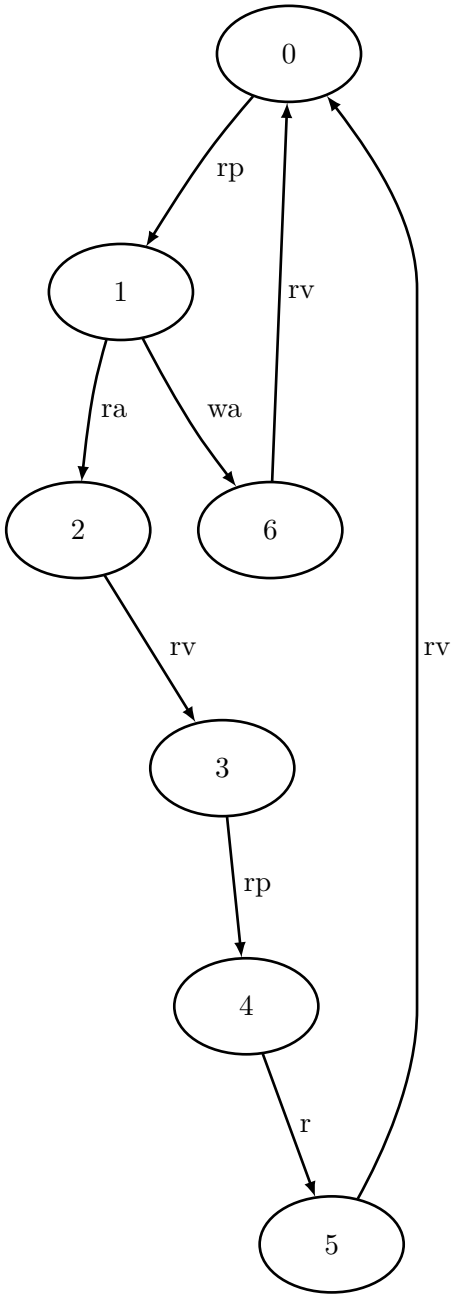

Figure A.5: CFG of the Co-Simulation Broker (R) with a correct implementation

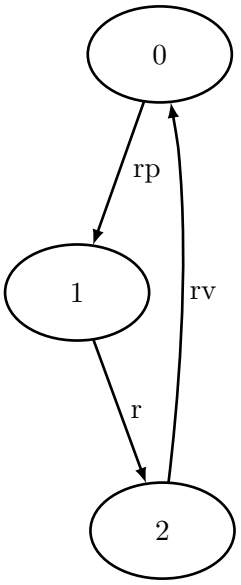

Figure A.6: CFG of the Co-Simulation Broker (R_b) with an incorrect buffer usage

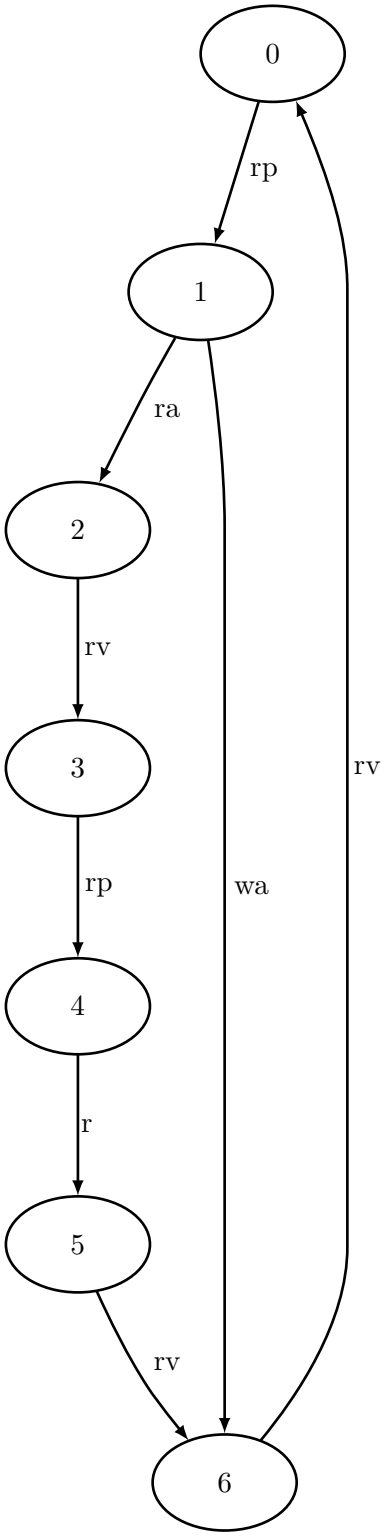

Figure A.7: CFG of the Co-Simulation Broker (R_s) with an incorrect semaphore usage

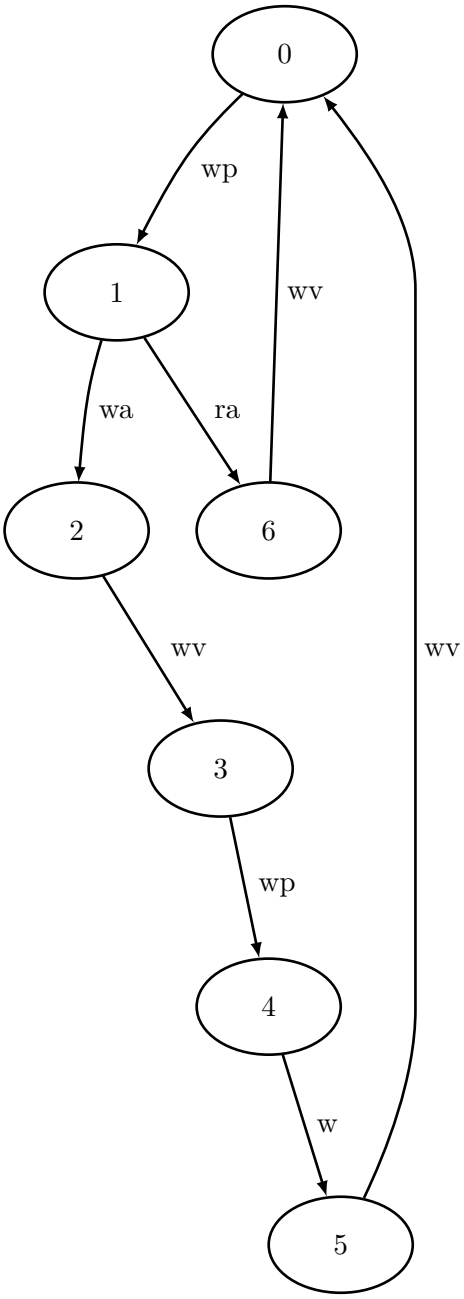

Figure A.8: CFG of the Co-Simulation Client (W) with a correct implementation

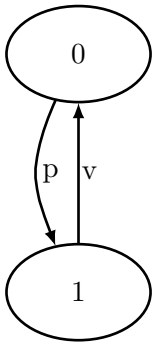

Figure A.9: CFG of a semaphore

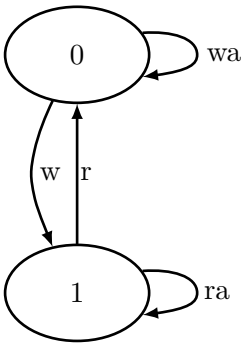

Figure A.10: CFG of a buffer with size 1

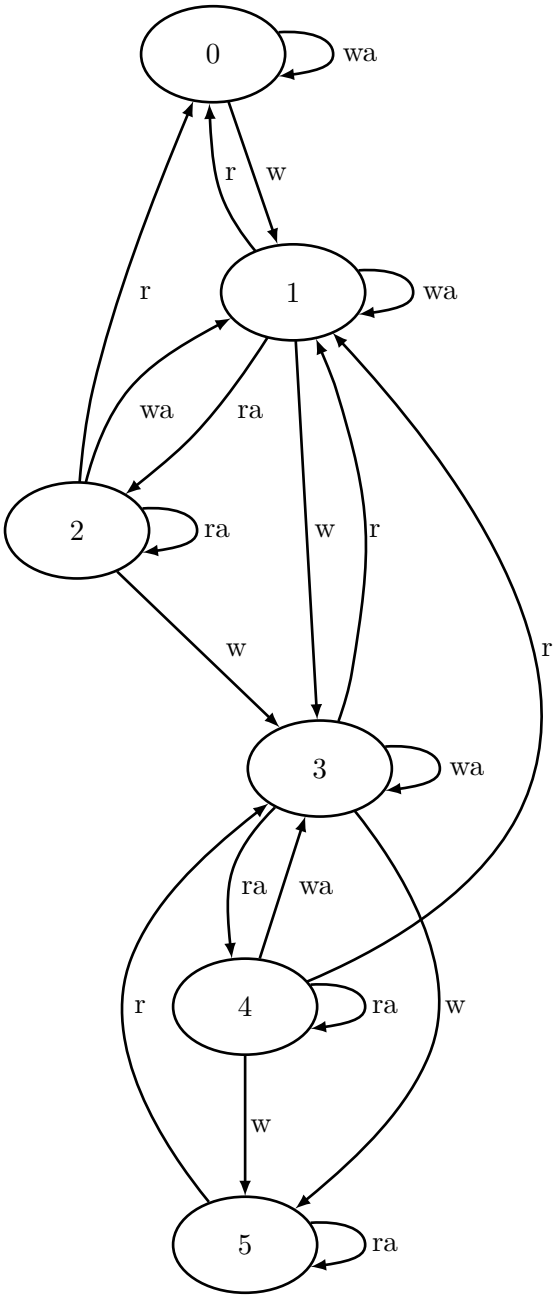

Figure A.11: CFG of a buffer with size 3

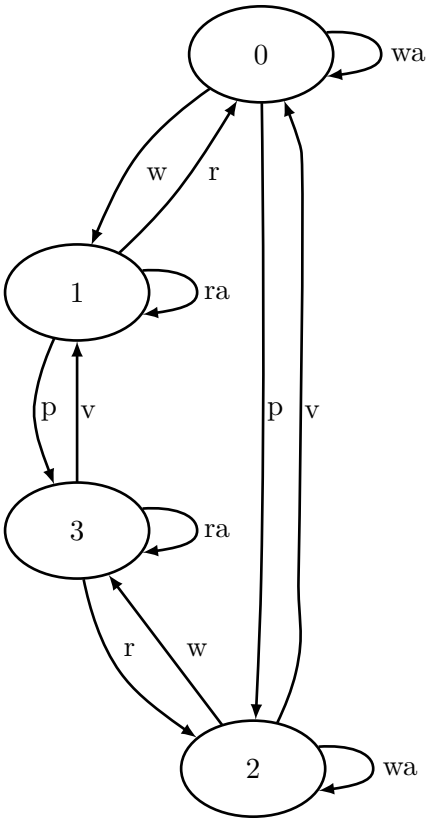

Figure A.12: Interleavings-CFG between the semaphore and the buffer

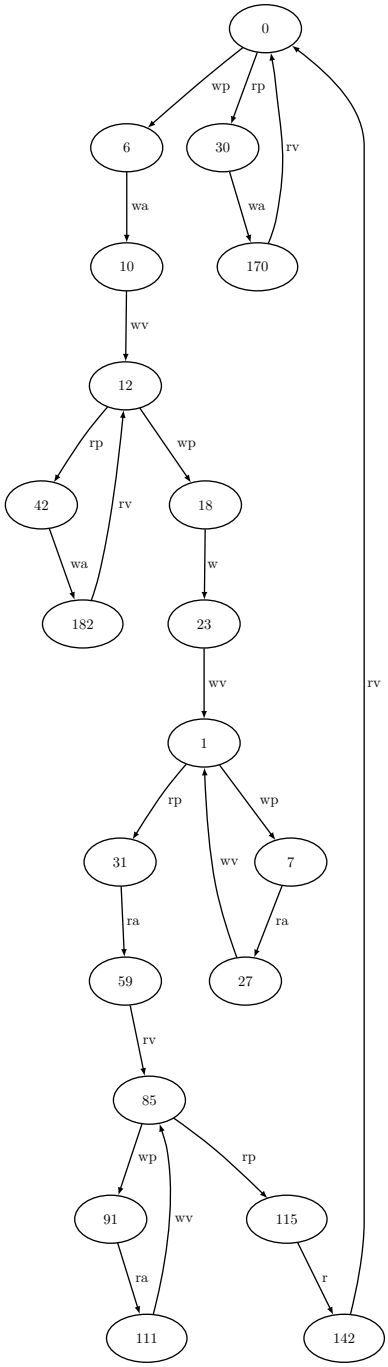

Figure A.13: Final CFG (correct)

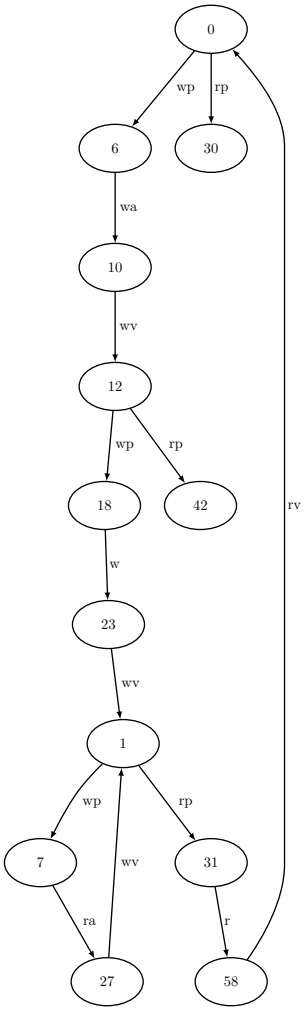

Figure A.14: Final CFG (incorrect buffer usage)

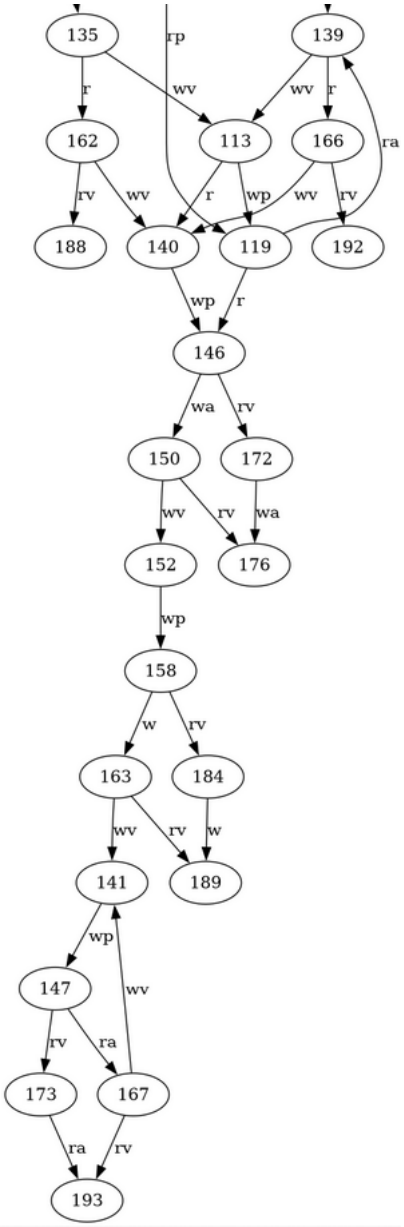

Figure A.15: Final CFG (incorrect semaphore usage)

$$R = \begin{bmatrix} 0 & rp & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & ra & 0 & 0 & 0 & wa \\ 0 & 0 & 0 & rv & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & rp & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & r & 0 \\ rv & 0 & 0 & 0 & 0 & 0 & 0 \\ rv & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure A.16: R matrix (correct)

$$R_b = \begin{bmatrix} 0 & rp & 0 \\ 0 & 0 & r \\ rv & 0 & 0 \end{bmatrix}$$

Figure A.17: R matrix (incorrect buffer usage)

$$R_s = \begin{bmatrix} 0 & rp & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & ra & 0 & 0 & 0 & wa & 0 \\ 0 & 0 & 0 & rv & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & rp & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & r & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & rv & 0 \\ rv & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ rv & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure A.18: R matrix (incorrect semaphore usage)

$$W = \begin{bmatrix} 0 & wp & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & wa & 0 & 0 & 0 & ra \\ 0 & 0 & 0 & wv & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & wp & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w & 0 \\ wv & 0 & 0 & 0 & 0 & 0 & 0 \\ wv & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure A.19: W matrix (correct)

$$S = \begin{bmatrix} 0 & p \\ v & 0 \end{bmatrix}$$

Figure A.20: S matrix

$$B_1 = \begin{bmatrix} wa & w \\ r & ra \end{bmatrix}$$

Figure A.21: B_1 matrix (size-1 buffer)

$$B_3 = \begin{bmatrix} wa & w & 0 & 0 & 0 & 0 \\ r & wa & ra & w & 0 & 0 \\ r & wa & ra & w & 0 & 0 \\ 0 & r & 0 & wa & ra & w \\ 0 & r & 0 & wa & ra & w \\ 0 & 0 & 0 & r & 0 & ra \end{bmatrix}$$

Figure A.22: B_3 matrix (size-3 buffer)

$$SB_1 = \begin{bmatrix} wa & w & p & 0 \\ r & ra & 0 & p \\ v & 0 & wa & w \\ 0 & v & r & ra \end{bmatrix}$$

Figure A.23: SB_1 matrix (interleaving of S and B_1)

```
from sympy import *
import networkx as nx
from networkx.drawing.nx_agraph import to_agraph


def solve():
    def program_missing_check(
        sem_p, sem_v, buffer_check, buffer_action, buffer_else_check
    ):
        return Matrix(
            [
                [0, sem_p, 0],
                [0, 0, buffer_action],
                [sem_v, 0, 0],
            ]
        )

    def program_correct(sem_p, sem_v, buffer_check, buffer_action,
        buffer_else_check):
        return Matrix(
            [
                [0, sem_p, 0, 0, 0, 0, 0],
                [0, 0, buffer_check, 0, 0, 0, buffer_else_check],
                [0, 0, 0, sem_v, 0, 0, 0],
                [0, 0, 0, 0, sem_p, 0, 0],
                [0, 0, 0, 0, 0, buffer_action, 0],
                [sem_v, 0, 0, 0, 0, 0, 0],
                [sem_v, 0, 0, 0, 0, 0, 0],
            ]
        )

    def program_double_unlock(
        sem_p, sem_v, buffer_check, buffer_action, buffer_else_check
    ):
        return Matrix(
            [
                [0, sem_p, 0, 0, 0, 0, 0, 0],
                [0, 0, buffer_check, 0, 0, 0, buffer_else_check, 0],
                [0, 0, 0, sem_v, 0, 0, 0, 0],
                [0, 0, 0, 0, sem_p, 0, 0, 0],
                [0, 0, 0, 0, 0, buffer_action, 0, 0],
                [0, 0, 0, 0, 0, 0, sem_v, 0],
                [sem_v, 0, 0, 0, 0, 0, 0, 0],
                [sem_v, 0, 0, 0, 0, 0, 0, 0],
            ]
        )
```

```
52      p, v = symbols("p v")
53
54      SEMAPHORE = Matrix(
55          [
56              [0, p],
57              [v, 0],
58          ]
59      )
60      save_graph(draw_graph_from_mat(SEMAPHORE), "semaphore")
61      save_mat(SEMAPHORE, "semaphore")
62
63      r, w = symbols("r w")
64      ra, wa = symbols("ra wa")
65      BUFFER1 = Matrix(
66          [
67              [wa, w],
68              [r, ra],
69          ]
70      )
71      save_graph(draw_graph_from_mat(BUFFER1), "buffer1")
72      save_mat(BUFFER1, "buffer1")
73
74      BUFFER3 = Matrix(
75          [
76              [wa, w, 0, 0, 0, 0],
77              [r, wa, ra, w, 0, 0],
78              [r, wa, ra, w, 0, 0],
79              [0, r, 0, wa, ra, w],
80              [0, r, 0, wa, ra, w],
81              [0, 0, 0, r, 0, ra],
82          ]
83      )
84      save_graph(draw_graph_from_mat(BUFFER3), "buffer3")
85      save_mat(BUFFER3, "buffer3")
86
87      BUFFER = BUFFER1
88
89      rp, rv = symbols("rp rv")
90      READER_correct = program_correct(rp, rv, ra, r, wa)
91      save_graph(draw_graph_from_mat(READER_correct), "reader-correct")
92      save_mat(READER_correct, "reader-correct")
93
94      READER_buffer_wrong = program_missing_check(rp, rv, ra, r, wa)
95      save_graph(draw_graph_from_mat(READER_buffer_wrong), "reader-
            buffer-wrong")
96      save_mat(READER_buffer_wrong, "reader-buffer-wrong")
97
98      READER_semaphore_wrong = program_double_unlock(rp, rv, ra, r, wa)
99      save_graph(draw_graph_from_mat(READER_semaphore_wrong), "reader-
            semaphore-wrong")
100     save_mat(READER_semaphore_wrong, "reader-semaphore-wrong")
101
102
```

```
103     wp, wv = symbols("wp wv")
104     WRITER_correct = program_correct(wp, wv, wa, w, ra)
105     save_graph(draw_graph_from_mat(WRITER_correct), "writer-correct")
106     save_mat(WRITER_correct, "writer-correct")
107
108     symmap = {
109         (p, rp): rp,
110         (p, wp): wp,
111         (v, rv): rv,
112         (v, wv): wv,
113     }
114
115     SB = kronecker_sum(SEMAPHORE, BUFFER, symmap)
116
117     save_graph(draw_graph_from_mat(SB), "semaphore-buffer-sum")
118     save_mat(SB, "semaphore-buffer-sum")
119
120     RW_correct = kronecker_sum(READER_correct, WRITER_correct, symmap)
121     RW_buffer_wrong = kronecker_sum(READER_buffer_wrong, WRITER_
            correct, symmap)
122     RW_semaphore_wrong = kronecker_sum(READER_semaphore_wrong, WRITER_
            correct, symmap)
123
124     return (
125         prod(RW_correct, SB, symmap),
126         prod(RW_buffer_wrong, SB, symmap),
127         prod(RW_semaphore_wrong, SB, symmap),
128     )
129
130
131 def mul(a, b, symmap={}):
132     if a == 1:
133         return b
134     if b == 1:
135         return a
136
137     if (a, b) in symmap:
138         return symmap[(a, b)]
139
140     if (b, a) in symmap:
141         return symmap[(b, a)]
142
143     if a == b:
144         return a
145
146     return 0
147
148
149
150
151
152
153
```

```
154 def sync_kron(A, B, symmap={}):
155     rows = A.rows * B.rows
156     cols = A.cols * B.cols
157     result = Matrix.zeros(rows, cols)
158 
159     for i in range(A.rows):
160         for j in range(A.cols):
161             for k in range(B.rows):
162                 for l in range(B.cols):
163                     result[i * B.rows + k, j * B.cols + l] = mul(
164                         A[i, j], B[k, l], symmap
165                     )
166 
167     return result
168 
169 
170 def kronecker_sum(A, B, symmap={}):
171     AI = eye(A.shape[0])
172     BI = eye(B.shape[0])
173     ABI = sync_kron(A, BI, symmap)
174     AIB = sync_kron(AI, B, symmap)
175     return ABI + AIB
176 
177 
178 def draw_graph_from_mat(m, silent=False):
179     G = nx.DiGraph()
180 
181     n = m.shape[0]
182 
183     if not silent:
184         print(f"shape: {m.shape}, n: {n}")
185         print(f"{m.row(0)}")
186 
187     for i in range(n):
188         for j in range(n):
189             val = m[i, j]
190 
191             if not val.equals(0):
192                 G.add_edge(i, j, label=str(val))
193 
194     start = 0
195 
196     reachable = nx.descendants(G, start)
197     reachable.add(start)
198 
199     G = G.subgraph(reachable).copy()
200 
201     if not silent:
202         sinks = [n for n, deg in G.out_degree() if deg == 0]
203         for sink in sinks:
204             print(f"found deadlock in node: {sink}")
205 
206     return G
```

```
207 def save_mat(mat, name):
208     tex = latex(mat)
209     with open(f"{name}_mat.tex", "w") as f:
210         f.write(tex)
211
212
213 def save_graph(g, name):
214     A = to_agraph(g)
215     A.layout("dot")
216     A.draw(name + ".png")
217     A.draw(name + ".dot")
218
219
220 Solution_correct, Solution_buffer_wrong, Solution_semaphore_wrong =
        solve()
221 save_graph(draw_graph_from_mat(Solution_correct), "solution-correct")
222 save_mat(Solution_correct, "solution-correct")
223 save_graph(draw_graph_from_mat(Solution_buffer_wrong), "solution-
        buffer-wrong")
224 save_mat(Solution_buffer_wrong, "solution-buffer-wrong")
225 save_graph(draw_graph_from_mat(Solution_semaphore_wrong), "solution-
        semaphore-wrong")
226 save_mat(Solution_semaphore_wrong, "solution-semaphore-wrong")
```

Listing A.1: Python script used for proofing correctness of the Co-Simulator

Overview of Generative AI Tools Used

No generative AI tools were used during the creation of this work.

List of Figures

List of Tables

Acronyms

CFG Control-Flow Graph. xv, xvi, 34, 35, 37–48, 59, 61

CISC Complex Instruction Set Computer. 3

CPSR Current Program Status Register. 16

IPC Interprocess Communication. 6

ISA Instruction Set Architecture. 1, 3, 4, 13, 16

ISS Instruction Set Simulator. xi, xiii, 1, 4, 12, 14, 15, 19, 20, 23–25, 29, 31, 33

PDL Processor Description Language. 1

QEMU Quick Emulator. xi, xiii, 1, 3, 4, 17, 20, 23, 27, 33

QMP QEMU Monitor Protocol. 10

RISC Reduced Instruction Set Computer. 3

RTL Register-Transfer Level. 29, 31

TB Translation Block. xi, xiii, 4, 10, 11, 13, 16, 19, 27

TCG Tiny Code Generator. 4

TOML Tom's Obvious Minimal Language. 17

VADL Vienna Architecture Description Language. xi, 1, 4, 12, 15, 16, 19, 24, 31

Bibliography

[abo25] About QEMU — QEMU 10.0.94 documentation, August 2025. [Online; accessed 8. Mar. 2026]. URL: https://qemu-stsquad.readthedocs.io/en/latest/about/index.html.

[AH12] Ahmad T. Al-Hammouri. A comprehensive co-simulation platform for cyber-physical systems. *Computer Communications*, 36(1):8–19, 2012. URL: https://www.sciencedirect.com/science/article/pii/S0140366412000047, doi:10.1016/j.comcom.2012.01.003.

[Ale92] Samuel O. Aletan. An overview of RISC architecture. In *ACM Conferences*, pages 11–20. April 1992. doi:10.1145/143559.143570.

[BCWS11] Jens Bastian, Christoph Clauß, Susann Wolf, and Peter Schneider. Master for co-simulation using fmi. In *8th International Modelica Conference*, volume 2011. Linköping University Electronic Press, Linköpings universitet Dresden, Germany, 2011. URL: https://ep.liu.se/en/conference-article.aspx?series=ecp&issue=63&Article_No=14, doi:10.3384/ecp11063115.

[BHD23] Niklas Bruns, Vladimir Herdt, and Rolf Drechsler. Processor verification using symbolic execution: A risc-v case-study. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023. doi:10.23919/DATE56975.2023.10137202.

[Bug22] Alexandra Bugariu. Automatically Identifying Soundness and Completeness Errors in Program Analysis Tools. *ETH Zurich*, 2022. doi:10.3929/ethz-b-000548050.

[Che] Tsong Yueh Chen. Metamorphic Testing: A Simple Method for Alleviating the Test Oracle Problem. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, pages 23–24. IEEE. doi:10.1109/AST.2015.18.

[Deg12] Ulan Degenbaev. *Formal specification of the x86 instruction set architecture*. PhD thesis, 2012. URL: https://jahrbib.sulb.uni-saarland.de/handle/20.500.11880/26394.

[ES07] Robert B. Evans and Alberto Savoia. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ESEC-FSE companion '07, page 549–552, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1295014.1295038.

[FHH+25] Florian Freitag, Linus Halder, Simon Himmelbauer, Christoph Hochrainer, Benedikt Huber, Benjamin Kasper, Niklas Mischkulnig, Michael Nestler, Philipp Paulweber, Kevin Per, Matthias Raschhofer, Alexander Ripar, Tobias Schwarzinger, Johannes Zottele, and Andreas Krall. The vienna architecture description language, 2025. URL: https://arxiv.org/abs/2402.09087, arXiv:2402.09087.

[Fri11] Markus Friedrich. *Parallel Co-Simulation for Mechatronic Systems*. PhD thesis, Technische Universität München, 2011. URL: https://mediatum.ub.tum.de/1063436.

[GHK17] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. Engineering a Formal, Executable x86 ISA Simulator for Software Verification. In *Provably Correct Systems*, pages 173–209. Springer, Cham, Switzerland, March 2017. doi:10.1007/978-3-319-48628-4_8.

[HGP+21] Simon Thrane Hansen, Cláudio Gomes, Maurizio Palmieri, Casper Thule, Jaco van de Pol, and Jim Woodcock. Verification of Co-simulation Algorithms Subject to Algebraic Loops and Adaptive Steps. In *Formal Methods for Industrial Critical Systems*, pages 3–20. Springer, Cham, Switzerland, August 2021. doi:10.1007/978-3-030-85248-1_1.

[kro26] View of Kronecker Algebra-based Deadlock Analysis for Railway Systems, May 2026. [Online; accessed 26. May 2026]. URL: https://traffic2.fpz.hr/index.php/PROMTT/article/view/2356/1234.

[KTS+21] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. Effective processor verification with logic fuzzer enhanced co-simulation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 667–678, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3466752.3480092.

[LAH11] Vincenzo Liberatore and Ahmad Al-Hammouri. Smart grid communication and co-simulation. In *IEEE 2011 EnergyTech*, pages 1–5, 2011. doi:10.1109/EnergyTech.2011.5948542.

[los26] The Lost Wake-Up Problem (Multithreaded Programming Guide), May 2026. [Online; accessed 26. May 2026]. URL: https://docs.oracle.com/cd/E19683-01/806-6867/sync-30/index.html.

[MB11] Robert Mittermayr and Johann Blieberger. Shared memory concurrent system verification using kronecker algebra. *CoRR*, abs/1109.5522, 2011. URL: http://arxiv.org/abs/1109.5522, arXiv:1109.5522.

[pos26a] pthreads(7) - Linux manual page, March 2026. [Online; accessed 9. Mar. 2026]. URL: https://man7.org/linux/man-pages/man7/pthreads.7.html.

[pos26b] shm_open(3) - Linux manual page, March 2026. [Online; accessed 9. Mar. 2026]. URL: https://man7.org/linux/man-pages/man3/shm_open.3.html.

[qem25a] Device Emulation — QEMU 10.0.94 documentation, August 2025. [Online; accessed 8. Mar. 2026]. URL: https://qemu-stsquad.readthedocs.io/en/latest/system/device-emulation.html#device-emulation.

[qem25b] GDB usage — QEMU 10.0.94 documentation, August 2025. [Online; accessed 8. Mar. 2026]. URL: https://qemu-stsquad.readthedocs.io/en/latest/system/gdb.html.

[qem25c] Introduction — QEMU 10.0.94 documentation, August 2025. [Online; accessed 8. Mar. 2026]. URL: https://qemu-stsquad.readthedocs.io/en/latest/system/introduction.html#virtualisation-accelerators.

[qem25d] QEMU User space emulator — QEMU 10.0.94 documentation, August 2025. [Online; accessed 8. Mar. 2026]. URL: https://qemu-stsquad.readthedocs.io/en/latest/user/main.html.

[qem26a] QEMU TCG Plugins — QEMU documentation, March 2026. [Online; accessed 8. Mar. 2026]. URL: https://www.qemu.org/docs/master/devel/tcg-plugins.html.

[qem26b] Translator Internals — QEMU documentation, March 2026. [Online; accessed 8. Mar. 2026]. URL: https://www.qemu.org/docs/master/devel/tcg.html.

[SM17] Karuturi Sneha and Gowda M Malle. Research on software testing techniques and software automation testing tools. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pages 77–81, 2017. doi:10.1109/ICECDS.2017.8389562.

[SSAP14] Thomas Strasser, Matthias Stifter, Filip Andrén, and Peter Palensky. Co-simulation training platform for smart grids. *IEEE Transactions on Power Systems*, 29(4):1989–1997, 2014. doi:10.1109/TPWRS.2014.2305740.

[tes26a] A Survey of Symbolic Execution Techniques, March 2026. [Online; accessed 8. Mar. 2026]. doi:10.1145/3182657.

[tes26b] Metamorphic Testing, March 2026. [Online; accessed 8. Mar. 2026]. doi:10.1145/3143561.

[tes26c] Property-Based Testing in Practice, March 2026. [Online; accessed 8. Mar. 2026]. doi:10.1145/3597503.3639581.

[Uma23] Mubarak Albarka Umar. A Study of Software Testing: Categories, Levels, Techniques, and Types. *Authorea Preprints*, October 2023. doi:10.36227/techrxiv.12578714.v1.

[VJ15] Aditya Venkataraman and Kishore Kumar Jagadeesha. Evaluation of inter-process communication mechanisms. *Architecture*, 86(64):1–5, 2015. URL: https://api.semanticscholar.org/CorpusID:6899525.

[WB13] Baobing Wang and John S. Baras. Hybridsim: A modeling and co-simulation toolchain for cyber-physical systems. In *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, pages 33–40, 2013. doi:10.1109/DS-RT.2013.12.