THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

VeriTest: Automatic Verification of Compiler Optimizations

by

Achmad Afriza Wibawa
47287888
a.wibawa@uqconnect.edu.au

School of Electrical Engineering and Computer Science,
University of Queensland

Submitted for the degree of Master of Computer Science
in the division of Computer Science

3 June 2024

Achmad Afriza Wibawa
a.wibawa@uqconnect.edu.au

January 1, 1970

Professor Michael Bruenig
Head of School
School of Electical Engineering and Computer Science
The University of Queensland
St Lucia, Queensland 4072

Dear Professor Bruenig,

In accordance with the requirements of the degree of Master of Computer Science in the School of Electical Engineering and Computer Science, I present the following thesis entitled

'VeriTest: Automatic Verification of Compiler Optimizations'

This thesis was performed under the supervision of Associate Professor Mark Utting, Emeritus Professor Ian J. Hayes, and Brae J. Webb. I declare that the work submitted in the thesis is my own, except as acknowledged in the text and footnotes, and that it has not previously been submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Achmad Afriza Wibawa

Dedication

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Acknowledgements

Acknowledgements: recognise those who have been instrumental in the completion of the project. Acknowledgements should include any professional editorial advice received including the name of the editor and a brief description of the service rendered.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Abstract

Abstract: should outline the main approach and findings of the thesis.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer. Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio. Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus. **feynman**

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While compilers are generally believed to be correct, unwanted errors would sometimes happen. This is due to: the common pitfalls of the language it's written in are *just* accepted by the community; edge cases of the language that are not well considered; or simply making a mistake inside the implementation [1, Sec. 1.2]. Human mistakes are natural in human-made software. As such, it is critical to *try* to minimize the intrinsic risks of error in compilers.

Minimizing the risks of error is non-trivial. A suite of testing mechanisms is needed to ensure the reliability of software. There are several ways to do this. For software, regression testing in the form of Unit, Integration, and System level tests are the industry standard ways for mitigating risks [2]. Such testing suites are ideal for software with human-understandable behavior. However, the behaviors of compilers itself are not exactly human-readable. As such, manually defining the obscure behaviors of compilers is tedious and time-consuming [3].

Another way to verify the behavior of compilers is to **formally** specify them [3].

add in details intro to formal specification

1.1 Objectives & Scope

The primary purpose of this project is to provide a framework for answering the 2$^{\text{nd}}$ step of the compiler research thread proposed by Lopes and Regehr:

> "Implement solver-based tools for finding incorrect optimizations, not-weakest preconditions, groups of optimizations that either subsume each other or undo each other, etc." [1, p. 5].

Hence, this project intodúces VeriTest: a software interface for automatic verification of GraalVM's expression optimizations. VeriTest aims to provide fast analysis over proposed optimization rules, written in Veriopt's GraalVM IR Domain Specific Language (DSL) [4, Sec. 3], in terms of incorrect optimizations. GraalVM developers could integrate the analysis into their development workflow. This would make it easy for GraalVM's developers to use the tool *as you go*, without being a *"proof expert"* on Isabelle.

To achieve this, we intend to leverage automation tools built in interactive theorem provers such as Isabelle [5], inspired by the previous works such as CompCert [3] and Alive [6], [7]; all of which integrate the theoretical aspects of formal verification into the practicality of using it in the industry. This project builds upon the previous works done to introduce formal semantics for GraalVM's [8] Intermediate Representation (IR) [4], [9] implemented in Isabelle [5].

1. GraalVM developers write a proposed optimization proof in Veriopt's DSL.
2. The optimization rule would be passed into VeriTest by the developers with the following goals:
   (a) Determine if it is verifiable;
   (b) If VeriTest can't determine, optimization rules would be passed to *"proof experts"*.

Figure 1.1: Proposed development workflow for GraalVM developers

However, verification that DSL matches the implementation of GraalVM would be out of the scope of this project. It would represent a thorough 2nd step of the compiler verification research thread [1, p. 5] – similar to Alive2's [7] solution on LLVM's formal verification. Furthermore, extending Veriopt's DSL and Isabelle's automated tools is out of the scope of this project, considering the time constraints of this project and the significant amount of work needed to undertake the endeavour. All of which could represent a future direction in the research.

In relation to the goals, there would be several questions implied:

1. Is it possible to extend the current workings of Veriopt in Isabelle to a tool that could provide analysis for the developers?
2. Would the analysis be useful to GraalVM developers?
3. How fast could the analysis be provided to GraalVM developers?

mention that benchmarking isabelle is out of scope of this project, and they can refer to isabelle section

1.2 Thesis Structure

refer to sections

Chapter 2

Background

2.1 Software Testing

Testing is a critical part of the software development life cycle [2]. It aims to minimize the risk of errors inside the software itself. In testing, we attempt to determine whether the semantics of a software follows their specifications. However, we quote Dijkstra's famous line [10]:

"Testing can be used to show the presence of bugs, but never to show their absence".

2.1.1 Regression Testing

In software engineering, the most commonly used method of software testing is regression testing. Regression testing revolves around identifying parts of the program that are changed and verifying their behavior through a test suite; e.g. Unit, System, and Integration-level tests [2]. Thus, that should verify whether the program behaves *as intended or not.*

In practice, regression testing is limited to the capabilities of humans to define the behavior of the program. Therefore, there is an inherent risk of errors happening due to human mistakes; as it's possible that the defined behavior is not correct. Defining the complete set of the behavior of the intended program through testing requires developers to spend a considerable amount of time writing tests manually [11].

Thus, **random testing** is introduced to substitute humans with defined systematic computer behaviors.

2.1.2 Random Testing

Random testing is a suite of random tests generated by the computer to determine the correctness of the program based on a predetermined set of rules [11]. For example, [12, Sec. 2] generates test cases and executes them on programs that have a predictable result. For instance, if a program deviates from the expected results, then the program has undefined behaviors.

The difficulty of random testing lies in determining the set of rules to generate test cases. In [11], the test cases are generated by substituting the subset of the input to a random input. While this allows test cases to be generated quickly, programs would only need several *"interesting values"* or edge cases to consider in their behavior. As such, a stronger test suite such as an **inference-based test generation** will be preferable in this project.

2.1.3 Inference-based Exhaustive Testing

Exhaustive tests such as [13] take into account the possible variable bounds of a program and convert it to a set of inference rules. For a program to be correct, all of the premises *(P, Q)* in the inference rules *(P* $\rightarrow$ *Q)* must be correct. Hence, finding a counterexample is as simple as determining if the bounds of a variable result in a satisfiable $\neg$*(P* $\rightarrow$ *Q)*. This could be extended even further by checking the inference rules on an SAT solver to find premises where the inference rules are incorrect [14, Ch. 5]. Exhaustive searching allows developers to only focus on defining the behavior of the program, rather than defining tests to define the behavior.

2.1.4 Mutation Testing

Mutation testing is a form of testing where faults (or mutations) are inserted into a program [15]. These mutations are based on the grammar of the program, and would represent programs that are invalid. A key aspect of mutation testing is to determine suitable mutations to inject into the program called *mutation operators* [15, Sec. 2], as inserting random values into the program would be akin to random testing. Specifically, Offutt et al. [15, Sec. 6] elaborates how mutation testing can be used to generate valid or invalid input strings that will have erroneous responses.

1. **Nonterminal Replacement**

   Every nonterminal symbol in a production is replaced by other nonterminal symbols.

2. **Terminal Replacement**

   Every terminal symbol in a production is replaced by other terminal symbols.

3. **Terminal and Nonterminal Deletion**

   Every terminal and nonterminal symbol in a production is deleted.

4. **Terminal and Nonterminal Duplication**

   Every terminal and nonterminal symbol in a production is duplicated.

Figure 2.1: Mutation operators for input strings. Summarised from [15, Sec. 6.2]

2.2 Formal Verification of Compiler

If software code is the recipe for system behaviors, then a compiler would be the chef who puts it all together. Most people would assume that the behavior of compiled programs will match the input program exactly. However, this is usually not the case [3]. Chefs have their way of creating magical concoctions from a recipe, and so does a compiler. Not only does a compiler try to replicate system behaviors, but it will try to make them faster in their ways; i.e. adding optimizations or reducing unneeded behaviors. However, the original behavior of the program must be preserved in order to consider a compiler to be correct [3], [4], [6], [7].

Verifying compiler correctness is not easy. While the correctness of software *could* be verified by defining behaviors through regression testing, compiler bugs are infamous for being hard to spot [2], [3]. Hence, it will be time-consuming to do and not exactly productive. There are a multitude of ways that a compiler can go wrong [1, Sec. 1.2]; all of which have their specific way of verification.

For example, CompCert [3] tries to tackle all of the implementation and semantics errors inside a compiler (See Sec. 2.2.1) – creating a completely verified compiler for the C language platform. Formally verifying compilers on the scale of CompCert will require vast amounts of time and resources, which projects often don't have.

As such, there are smaller-scale projects such as Alive [6] & Alive2 [7] that focus on behavior translation errors in LLVM's peephole optimizer (See Sec. 2.2.2). Veriopt attempts to work on the same steps as CompCert – by defining the IR of GraalVM and proving much of the side-effect-free data-flow behavior optimizations that occur in GraalVM [4], [9] (See Sec. 2.4).

2.2.1 CompCert

CompCert verifies that Clight, a subset of the C programming language [3], is correct through several steps:

1. With deterministic programs, a compiler compiles a source program to the produced program – in which both of the programs must have the same behavior.

   This step is done by augmenting the compiler code with a *certificate* – code that carries proof that the behavior is exactly as intended [3, Sec. 2.2].

2. Compiler optimization rules must be accompanied by the formal definition of their Intermediate Representation (IR) semantics [3], [9].

3. Lastly, to formally verify each of the optimization rules, a compiler must either:

   (a) Prove that the code implementing the optimization is correct [3, Sec. 2.4].
   Veriopt uses this approach in verifying data-flow optimizations (See Sec. 2.4).

   (b) Prove that the unverified code produces the correct behavior in their translation [3, Sec. 2.4].
   Alive uses this approach in verifying LLVM (See Sec. 2.2.2).

CompCert formally verifies each step in the source, intermediate, and target languages that go through the compiler [3, Sec. 3.3]. Furthermore, each code translation and optimization are accompanied by *certificates* that prove the correctness of the semantics. This is done through Coq, a proof assistant similar to Isabelle. The workflow of Coq also remains closely related to Isabelle (See Sec. 2.3) [3, Sec. 3.3].

CompCert utilizes Coq not only to formally verify the semantics of Clight but also to generate the verified parts of the compiler code [3, Sec. 3.4]. The clever part of CompCert is that it utilizes the functional programming capabilities of Coq to automatically generate code. As such, it is able to write a compiler-compiler – which is the 3$^{\text{rd}}$ step of compiler verification research [1].

The formal verification of Clight results in 42000 lines of Coq – approximately equivalent to 3 years of man-hours of work [3, Sec. 3.3]. As you can see, replicating the results of CompCert will require an enormous amount of work. Formal verifications such as Alive (See Sec. 2.2.2) and Veriopt (See Sec. 2.4) undertake the smaller subset, namely the 1$^{\text{st}}$ and 2$^{\text{nd}}$ step, of the compiler verification research thread [1].

2.2.2 Alive

Alive takes on a subset of compiler verification by verifying that code optimizations inside LLVM are correct [6]. For example, a compiler will optimize:

$$(LHS = x * 2) \implies (RHS = x << 1) \tag{2.1}$$

(2.1) explains that LHS is transformed into RHS [6, Sec. 2.1]. While this may seem trivial, there would be a lot of edge cases where the behavior translation might be incorrect (e.g., buffer overflows).

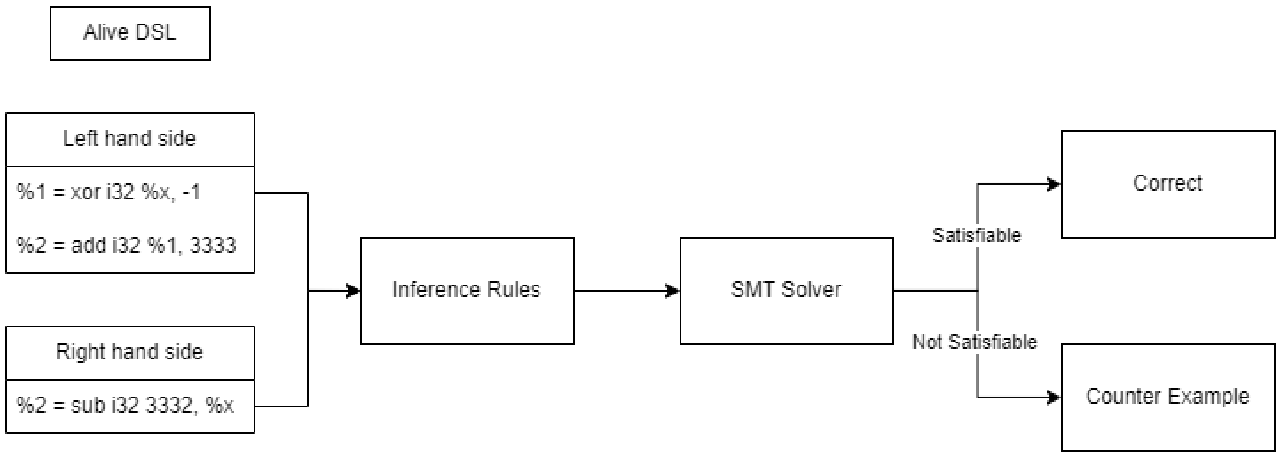

Figure 2.2: How Alive verifies $(LHS = (X \oplus -1) + C) \implies (RHS = (C-1) - X)$ [6, p. 1]

To verify that code optimizations are correct, Alive utilizes inference-based exhaustive testing (See Sec. 2.1.3) that allows the tool to encode machine code behaviors to inference rules [6, Sec. 3.1.1]. These inference rules are then passed to SMT solvers to check for their satisfiability (See Fig. 2.2). If the translation is proven to be correct, then it means that the underlying optimization code is correct.

Alive encodes LLVM's [16] underlying IR semantics through their DSL [6, Fig. 1]. The DSL specification is made to be similar to LLVM's IR semantics to allow developers to easily integrate Alive with the development of DSL. This represents the *certificate* that the underlying optimization is formally verified to be correct.

Alive takes this further by creating Alive2: a system to translate LLVM IR into Alive's IR [7]. This allows the developers to entirely focus on developing LLVM, while completely ignoring the specifications of Alive. This is found to be effective, as differential testing of LLVM's unit tests and Alive2 discover multiple errors inside the unit test behaviors themselves [7, Sec. 8.2]. Alive & Alive2 covers the whole 1st and 2nd steps of the compiler optimization research thread [1, p. 5].

2.3 Isabelle

Isabelle is an interactive theorem prover that utilizes a multitude of tools for automatic proving – similar to Coq, used by CompCert 2.2.1. Isabelle emphasizes breaking down a proof for a theorem toward multiple smaller goals that are achievable called tactics. Tactics are functions, written in the implementation of Isabelle, that work on a proof state [14]. Tactics either output a direct proof towards the goal or break them down into smaller sub-goals in a divide-and-conquer manner. These tactics work on the foundation that theory definitions can be modified into a set of inference rules that could be automatically reasoned with by the system.

add in isabelle benchmarks

Finding the right proving methods and arguments to utilize is one of the biggest issues for proving a theorem [14]. Many tools in Isabelle's arsenal can help the user progress towards their proof [5]. However, the most notable ones are Sledgehammer, Quickcheck, and Nitpick.

mention isabelle benchmark

2.3.1 Sledgehammer

Sledgehammer is one of the tools in Isabelle that *could* automatically prove a theorem. It utilizes the set of inference rules as conjectures that can be cross-referenced with relevant facts (lemmas, definitions, or axioms) from Isabelle [14, Sec. 3]. Afterward, Sledgehammer passes them into resolution prover and SMT solvers that try to solve it [14, Sec. 3.3]. If reasonable proof is found, Sledgehammer reconstructs the inference rules back into a *relatively* human-readable proof definition in the style of Isabelle/Isar [14, Sec. 3.4].

Utilizing Sledgehammer has been proven to be an effective method of theorem proving. Sledgehammer, combined with external SMT solvers, can solve 60.1% of proof goals, with a 44.7% success rate for non-trivial goals [17, Sec. 6]. Despite their potential, as Blanchette et al. note [14, p. 2]:

> "...most automatic proof tools are helpless in the face of an invalid conjecture. Novices and experts alike can enter invalid formulas and find themselves wasting hours (or days) on an impossible proof; once they identify and correct the error, the proof is often easy."

To make it easier for users to avoid this trap, Isabelle complements the automatic theory proving with counterexample generators such as Quickcheck and Nitpick.

2.3.2 Quickcheck

Quickcheck is one of the counterexample generators in Isabelle. It works by utilizing the code generation features of Isabelle by translating conjectures into ML (or Haskell) code [13]. This allows Quickcheck to discover counterexamples quickly by assigning free variables on the code via random, exhaustive, or narrowing test data generators. However, this would also mean that Quickcheck is limited to *executable* and *some* well-defined unbounded proof definitions [13].

Random testing of a conjecture assigns free variables with pseudo-random values [13, Sec. 3.1]. This strategy tends to be fast, with the ability to generate millions of test cases within seconds. However, random testing could easily overlook obvious counterexamples. Furthermore, random testing is also limited to well-defined proof definitions [13]. As such, exhaustive and narrowing test data generators are more suitable for proof definitions that are non-trivial or have unbounded variables.

Exhaustive and narrowing test data generators systematically generate values up to their bounds [13]. However, exhaustive test data generators fail to find counterexamples of proofs that have unbounded variables. Narrowing test data generators improves it by evaluating proof definitions symbolically rather than taking variables at face value. This is possible due to term rewriting static analysis done on proof definitions [13, Sec. 5].

Based on observed results, Bulwahn notes that random, exhaustive, and narrowing testing are comparable in terms of performance; with exhaustive testing finding non-trivial counterexamples easily compared to random testing [13, Sec. 7]. As much of proof definitions are defined over unbounded variables, exhaustive testing is the default option for Quickcheck.

2.3.3 Nitpick

Nitpick is an alternative to find counterexamples for proof definitions. Instead of enumerating the bounds of free variables inside the system of Isabelle, Nitpick passes conjectures – translation of proof definitions into inference rules – into SAT solvers [14, Sec. 5]. SAT solvers search for the premises that falsify the given conjecture. If a conjecture has bounds over finite domains, Nitpick will *eventually* find the counterexample. Conjectures with unbounded variables will be partially evaluated [14, Sec. 5.2]. However, it could not determine whether infinite bounds result in a satisfiable conjecture.

Nitpick and Quickcheck shouldn't be compared to one another. Instead, they are tools that should be used interchangeably to determine whether a conjecture will be possible to prove [13]. The performance of Nitpick is comparable to Quickcheck, with the addition that Nitpick can find counterexamples to proof definitions that are not executable within Isabelle's code generation [13, Sec. 7].

2.3.4 Limitations

It is worthy to note that Quickcheck has some limitations over arbitrary type definitions [13, Sec. 3.1] and conditional conjectures [13, Sec. 4]. For arbitrary type definitions, Quickcheck is unable to transform the conjectures into internal Isabelle datatypes. As such, users need to define a way to *construct* their datatype, which would be used by Quickcheck to build test generators. For conditional conjectures, Quickcheck would evaluate the given conjectures with no regards to their premises [13, Sec. 4]. As such, users would need to specify their own test generators that would take the premises into account [13, Sec. 4.1]. Furthermore, this limitations extends to Nitpick. For (Co)inductive datatypes, Nitpick needs the user to properly define their encoding of (Co)inductive datatypes, and manually provide selectors/discriminators if Nitpick cannot automatically infer one [14, Sec. 5.4].

2.4 Veriopt

In comparison to CompCert and Alive, the theoretical aspects of compiler verification are similar. GraalVM's IR is made up of two components: control-flow nodes and data-flow nodes; which are combined as a sea-of-nodes data structure [9]. However, Veriopt's DSL only concerns the subset of GraalVM's IR, which is the side-effect-free data-flow nodes [4]. Side-effect-free data-flow nodes are comparatively easier to prove and optimize, as it is considered to be defined – as opposed to LLVM's undefined and poisoned variables [7].

Fig. 2.3 defines the structure of the DSL for an optimization rule. Veriopt's DSL is implemented in Isabelle [5], which represents optimization rules as inductive datatypes that allows efficient reasoning within Isabelle [9, Sec. 3] [18]. The **optimization** keyword represents the proof definition that must be proven in Isabelle. Note that 2 proof obligations must be met to consider that the side-effect-free optimization is correct: (1) proof that the optimization rule will terminate; (2) proof that each pass of the optimization rule will result in a refinement of the expression [4]. Note that these proofs need to be provided by users.

optimization *InverseLeftSub:* $(x\ -\ y)\ +\ y \longmapsto x$

1. $trm(x)\ <\ trm(BinaryExpr\ BinAdd\ (BinaryExpr\ BinSub\ x\ y)\ y)$
2. $BinaryExpr\ BinAdd\ (BinaryExpr\ BinSub\ x\ y)\ y \sqsupseteq x$

Figure 2.3: Sample of Veriopt's optimization rule and proof obligations DSL [4, Fig. 3]

Currently, there are some tools that the developers of GraalVM could use to provide a *certificate* for the compiler code [4, Sec. 7]. A semi-automated approach exists in the form of source code annotations [4, Sec. 5.1]. However, this semi-automated approach is inadequate due to only matching if the optimization rules exists in Veriopt's current theory base without any logical reasoning done. Providing proof obligations for an optimization rule will be challenging for developers who are not *experts* in program verification, as it requires the developers of GraalVM to be familiar with Isabelle – something that ideally Veriopt would like to avoid. Similar tools such as Alive [6] are preferable. Hence, that is where this project contributes.

Chapter 3

Possible Approaches

Veritest represents another tool that the developers of GraalVM could use to provide a *certificate* for their implementation of optimization rules. To assist in verifying the optimization rule, the tool would need to be able to classify each of the optimization rules (See Fig. 3.1). Furthermore, there are several key non-functional requirements for the system that need to be satisfied (See Fig. 3.2).

1. The optimization rule is false;

   This would require the tool to generate obvious counterexamples via Quickcheck (See 2.3.2) or Nitpick (See 2.3.3).

2. The optimization rule is true;

   This would require the tool to verify that Sledgehammer (See 2.3.2) can provide proof obligations for the optimization rule **without** dynamically defining proof tactics.

3. The optimization rule would require manual proving by *"proof experts"*.

   This means that the optimization is non-trivial: Isabelle is not able to find an obvious counterexample, and proving would require additional tactics or sub-goals to be defined.

Figure 3.1: The classification of an optimization rule

1. Developers of GraalVM need to be able to integrate this easily into their test suite;
2. Developers of GraalVM can easily use this without understanding Isabelle;
3. *If possible*, the system doesn't require enormous computing resources locally.

Figure 3.2: Non-functional requirements of the Framework

Understanding Isabelle's implementation is crucial to realize the goals of this project. At a glance, there are four possible approaches to the project:

- Utilize Isabelle Client - Server interactions (See Sec. 3.2);
- Extend Isabelle/Scala (See Sec. 3.3);
- Utilize Isabelle CLI (See Sec. 3.4)
- Create an Interpreter for GraalVM's optimization DSL (See Sec. 3.5).

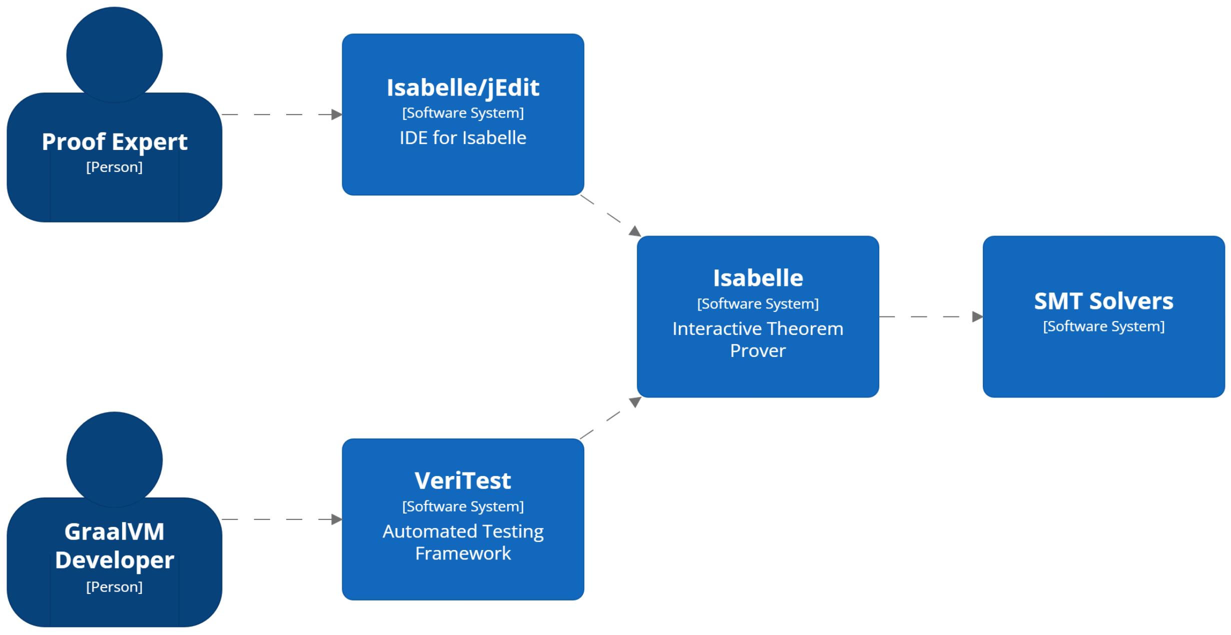

Figure 3.3: Proposed Solution

change veritest label from figure

Fig. 3.3 depicts the overview of the proposed solution's system landscape. In theory, it would utilize Isabelle in a similar manner as Isabelle/jEdit [19]. Therefore, it should be able to use the same Isabelle functionalities as Isabelle/jEdit does.

3.1 Isabelle System Overview

Isabelle is made up of two significant components: Isabelle/ML and Isabelle/Scala [19, Ch. 5] (See Fig. 3.4). Isabelle/ML acts as the core functionality of Isabelle, harboring all the tools needed for proving theorems. Isabelle/Scala acts as the system infrastructure for Isabelle/ML – hiding all implementation details of Isabelle/ML.

3.2 Utilizing Isabelle Server

Isabelle Server acts as the core Isabelle process that allows theorems and all the required facts to be loaded up and processed by Isabelle/ML [19, Ch. 4]. Interactions to Isabelle/Server would require a duplex socket connection over TCP [19, Sec. 4.2]. To simplify the communication between the framework and Isabelle/Server, we utilize Isabelle Client [19, Sec. 4.1.2] – a proxy

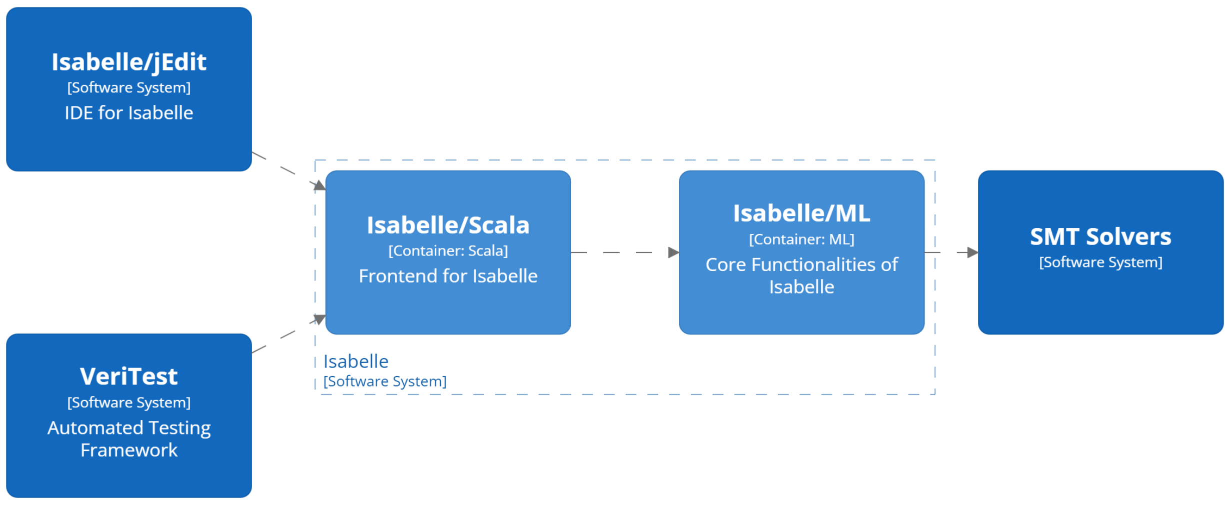

Figure 3.4: Isabelle System Overview

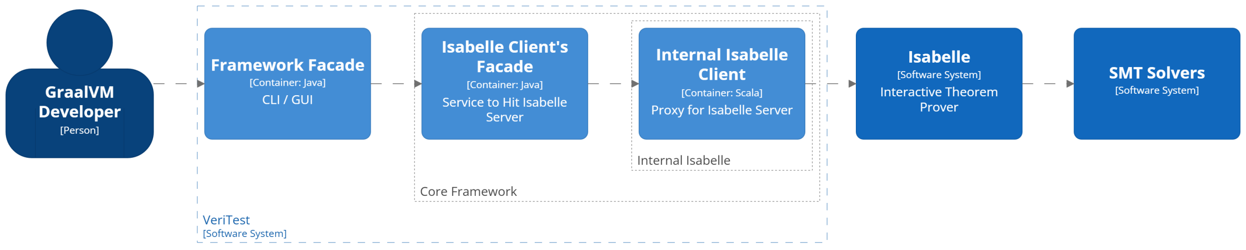

Figure 3.5: Utilizing Isabelle Client - Server interaction

for Isabelle/Server that handles all the communication protocols of Isabelle/Server (See Fig. 3.5).

Isabelle Server can load theorems and process requests in parallel [19, Sec. 4.2.6]. This implies that it would require a *facade* that implements a demultiplexer for asynchronous messages on Isabelle Client (See Fig. 3.5). As such, this solution – *theoretically* – it would allow the framework to offload the computing resources of loading and processing optimization proofs at external sites (See Ch. 5). VeriTest currently implements this approach, which is elaborated in detail in Ch. 4.

3.3 Extending Isabelle/Scala

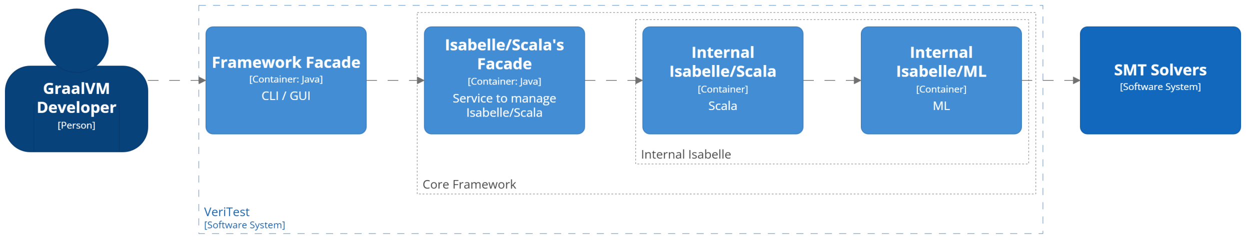

Figure 3.6: Extending Isabelle/Scala

Isabelle can be extended by accessing Isabelle/Scala functions [19, Ch. 5]. Extending Isabelle/Scala would require the framework to utilize Isablle's Scala compiler [19, Sec. 5.1.4]. Utilizing Isabelle/Scala would also mean that VeriTest would be capable of utilizing Isabelle/ML as well. Subsequently, this also implies that it is possible to extend and fine-tune existing automated tools to fit the requirements of this project. Fig. 3.6 depicts the proposed solution for this option.

However, the sheer complexity of Isabelle implies that extending Isabelle/Scala would require much of the project's timeline to understand Isabelle/Scala, which the time constraints of this thesis project does not allow. Furthermore, extending Isabelle/Scala *could* mean that the framework would take up much of the computing resources to execute Isabelle/ML functions locally.

3.4 Utilizing Isabelle CLI

Veriopt's current semi-automated approach [4, Sec. 5.1] utilizes Isabelle CLI, a wrapper for Isabelle/Scala functions [5]. As such, this solution would represent an Isabelle management service, where each optimization rule will be passed into an Isabelle/Scala function inside an Isabelle process and managed accordingly. In essence, the solution is similar to Sec. 3.3. However, the solution will require additional processing overhead in the form of multiple local Isabelle processes for each of the optimization rules, instead of a single VeriTest and Isabelle instance.

3.5 Interpreter for DSL

Building an interpreter for GraalVM's optimization DSL acts as a last resort to the project. To implement this, it would require a significant amount of time to rework the DSL into the framework, and designing tools similar to Quickcheck (See Sec. 2.3.2) in order to satisfy the system requirements. Reinventing the wheel would not be productive for the project, and it would result in a tool that is far inferior to Isabelle. Therefore, this option should be avoided.

Chapter 4

Implementation

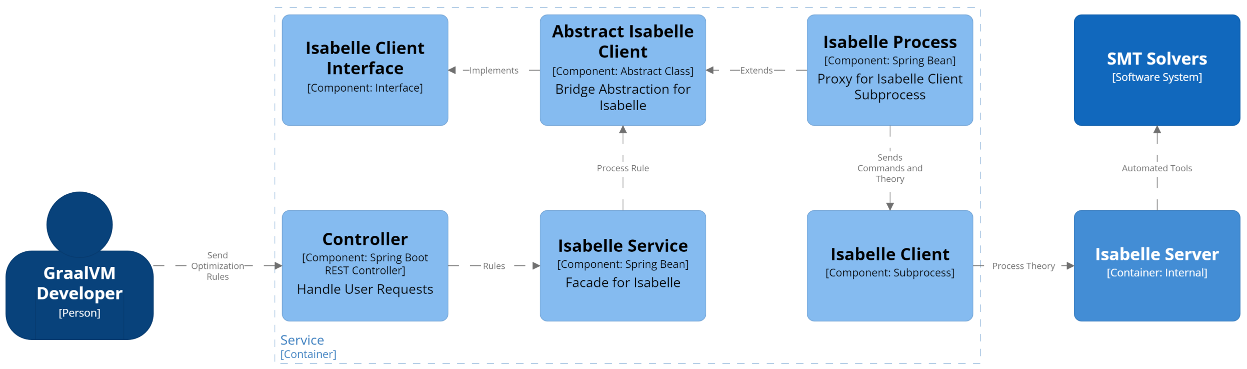

Figure 4.1: Software Architecture of Veritest

Figure 4.1 outlines the architecture implemented for VeriTest to implement a solver-based tool for finding incorrect optimizations, referred by Lopes and Regehr [1, p. 5]. VeriTest is available in online repository[1]. The API documentation for VeriTest can be accessed through Postman[2]. There are several key design decisions needed to achieve the functional and non-functional requirements of this project (See Sec. 3), all of which are outlined in the following sections.

4.1 Isabelle as Proof Engine

We leverage Isabelle's automated tools – Sledgehammer to find proofs; Nitpick and Quickcheck to find counterexamples – to carry out the analysis for each of the optimization rules. This is possible due to the Intermediate Representation (IR) of GraalVM represented as an inductive datatype inside Isabelle [9, Sec. 3]. This encoding allows Isabelle to automatically translate the encoding into datatypes internal to Isabelle [18]. The efficacy of this method is outlined in Section 5.

Furthermore, as the inductive datatype acts as the grammar of Veriopt's DSL, erroneous syntax can be detected automatically by Isabelle. This is due to the fact that the inductive datatype expects certain types to be present in order for the optimization to by syntactically correct.

[1]https://github.com/achmadafriza/veritest-dev

[2]https://documenter.getpostman.com/view/12004801/2sA35D64D8

$$\textbf{optimization}\ \textit{WrongMultiplication:}\ (x\ *\ 1)\ \longmapsto\ x \tag{4.1}$$

$$\textbf{optimization}\ \textit{CorrectMultiplication:}\ (x * (\textit{ConstantExpr}\ (\textit{IntVal}\ b\ 1))) \longmapsto\ x \tag{4.2}$$

In (4.1), we see that the optimization is intuitively correct. However, as $(x * 1)$ is translated into *BinaryExpr BinMul x 1*, Isabelle expects the syntax to be in the form of *BinaryExpr BinMul IRExpr IRExpr* [4, Definition 1]. With type unification, we can infer that x is of type *IRExpr*. However, 1 is internally encoded in Isabelle as an Isabelle word, not an IR expression [4, Definition 2]. We can amend the optimization rule by defining it as (4.2). As such, syntax errors can be detected.

4.2 Mutual Exclusion

$$\begin{aligned}
&\textbf{optimization}\ \textit{WrongOptimization} &&: \quad x + y \longmapsto x - y \qquad (4.3)\\
&\qquad\qquad\qquad \textbf{\textit{sorry}} && \\
&\textbf{optimization}\ \textit{ImpactedOptimization}&&: \quad x * 1 \longmapsto x \qquad (4.4)\\
&\qquad\qquad\qquad \textbf{\textit{nitpick}} &&
\end{aligned}$$

Figure 4.2: Example of conflicting optimization rules

As optimization rules are *proposed* in VeriTest, the proposed optimization rule might be incorrect due to counterexamples. On figure 4.2, we see that optimization rule (4.3) is obviously false due to arithmetic rules. But as we can omit proof for such optimization rules, by including the keyword **sorry**, this optimization rule is regarded as proven by Isabelle. This can impact the verification of optimization rule (4.4), falsifying the optimization rule, by the following reasoning:

$$z * 1\ \longmapsto\ z,\ (4.4) \tag{4.5}$$

$$\begin{aligned}
&\textit{Let}\ z = (x + y) && (4.6)\\
&(x + y) * 1\ \longmapsto\ (x + y) && (4.7)\\
&(x + y) * 1\ \longmapsto\ (x + y)\ \longmapsto\ (x - y),\ \text{From (4.3)} && (4.8)\\
&(x + y) * 1\ \longmapsto\ (x - y) && (4.9)
\end{aligned}$$

$$\begin{aligned}
&\textit{Let}\ x = 1,\ y = 1 && (4.10)\\
&(1 + 1) * 1\ \longmapsto\ (1 - 1) && (4.11)\\
&2 * 1\ \longmapsto\ 0 && (4.12)\\
&2\ \longmapsto\ 0 && (4.13)\\
&\textit{False} && (4.14)
\end{aligned}$$

As such, it is critical for each of the *proposed* optimization rule to be mutually exclusive with one another, only interacting with proven theories inside Veriopt's current theory base. This is aided by Isabelle's session framework [19, Ch. 2], where each session only interacts with the imported theorems and lemmas inside of it, equivalent to Isabelle sessions defined for Isabelle/jEdit. As such, optimization rules are analyzed by spawning a session for each of the rules and commands, depending on the current state of the analysis.

4.3 Interfacing with Isabelle

With regards to the non-functional requirement:

> "Developers of GraalVM need to be able to integrate this easily into their test suite"

While developer experience is important with concerns of developer workflow, the only concern for this project is to demonstrate the full capabilities of the analysis done inside VeriTest. The extent of *how* GraalVM would integrate VeriTest into their test suite is out of the scope of this project. As such, it is imperative that VeriTest provides a *generic* interface towards analyzing optimization rules.

```
interface IsabelleProcessInterface extends Closeable:
    (BlockingQueue, LockCondition) open()
    TaskId submitTask(TaskType, args)
```

Figure 4.3: Pseudocode for Isabelle Process Interface

We utilize the Isabelle Client-Server interface to interact with Isabelle. Interacting with Isabelle Server can be done in several ways: through Isabelle Client, and through a TCP socket [19, Ch. 4]. For the purpose of VeriTest, we utilize Isabelle Client to simplify the implementation. Because of the possible alternatives, VeriTest abstracts the interactions through `IsabelleProcessInterface`, as outlined in figure 4.3. Isabelle Process component outlined in figure 4.1 implements this interface, and acts as a proxy for Isabelle Client.

```
interface IsabelleClient extends Closeable:
    Async<Task> startSession(request);
    Async<Task> stopSession(request);
    Async<Task> useTheory(request);
```

Figure 4.4: Pseudocode for Isabelle Client

Furthermore, as any analysis done would only concern *submitting* an optimization rule to analyze, VeriTest generalizes the interaction through `IsabelleClient` interface, as illustrated in figure 4.4. This interface generalizes any type of commands that would be invoked in Isabelle Server [19, Sec. 4.4]. This allows other components to focus on the implementation of the analysis, rather than worrying about how to interface with Isabelle.

```
abstract class AbstractIsabelleClient implements IsabelleClient:
    var Isabelle: IsabelleProcessInterface;

    Async<Task> startSession(request):
        return Isabelle.submitTask(START_SESSION, request)
            then waitForCompletion()
            then (TaskId) -> getResult(TaskId)
            then switch (result):
                case SuccessTask response -> return response;
                case ErrorTask error -> throw error;

    ...
```

Figure 4.5: Pseudocode for Abstract Isabelle Client

As any interaction with Isabelle would involve the procedures of `IsabelleClient`, we utilize a bridge pattern to abstract Isabelle's interaction, as seen in Abstract Isabelle Client in figure 4.1. The `AbstractIsabelleClient`[3] defines the procedures of how to interact with Isabelle, utilizing Isabelle Process, which can be seen in figure 4.5. `stopSession()` and `useTheory()` procedures use the same algorithm as `startSession()` to define their interactions. Note that `then` keyword represent asynchronous function compositions. `waitForCompletion()` waits for the asynchronous functions to complete. `getResult()` is a procedure that handles demultiplexing tasks that allows for concurrent processes, which is elaborated in section 4.4.

It is worthy to consider that Isabelle responses in the form of `Task` represents various types of responses from `startSession()`, `useTheory()`, `stopSession()`, and even miscellaneous error messages from Isabelle [19, Sec. 4.4]. Each task would contain at least three fields: (1) their task identifier; (2) the type of message being sent, whether the corresponding task is finished or have errors; and (3) a list of messages from Isabelle. Through pattern matching Isabelle's messages, we can classify the result of the analysis, as elaborated on section 4.5. We parse Isabelle responses in the form of a string with a customized polymorphic deserializer called `TaskDeserializer`[4], which utilizes Jackson's `ObjectMapper`[5] – that transforms a string into a abstract JSON tree – with added functionality through Java Reflections API. This method of parsing is used extensively in figure 4.7 that elaborates how to interface with Isabelle Client's subprocess.

Through this approach to interface with Isabelle, we provide modularity and extensibility for VeriTest, which would immensely assist GraalVM developers integration to their test suite and future work to be done.

4.4 Parallel Execution

To support VeriTest's goal of providing fast analysis towards optimization rules, VeriTest utilizes concurrent processing by using asynchronous functions. The implementation of Isabelle Process and Abstract Isabelle Client utilizes a producer-consumer pattern, ensuring that interactions towards Isabelle Client would not be a bottleneck for other optimization rules.

[3]AbstractIsabelleClient.java & IsabelleProcess.java extending `AbstractIsabelleClient`

[4]TaskDeserializer.java

[5]https://fasterxml.github.io/jackson-databind/javadoc/2.7/com/fasterxml/jackson/databind/ObjectMapper.html

```
class IsabelleProcess implements IsabelleProcessInterface:
    var processQueue : BlockingQueue;

    (BlockingQueue, LockCondition) open():
        startIsabelleSubprocess();
        queue := new BlockingQueue();

        var daemon := Daemon();
        daemon.processQueue := queue;
        daemon.notEmpty := new LockCondition();
        daemon.asyncQueue := new ReentrantLock();

        return (daemon.asyncQueue, daemon.notEmpty);

    synchronized TaskId submitTask(TaskType, args):
        writeToProcess(TaskType, args);

        var taskResponse := this.syncQueue.waitAndTake();

        return taskResponse.taskId;

    ...
```

Figure 4.6: Pseudocode for writing to process

```
class Daemon:
    var processQueue : BlockingQueue;

    var notEmpty : LockCondition;
    var asyncQueue : BlockingQueue;

    void run():
        while process isAlive:
            var string := readFromProcess();

            var response := TaskDeserializer.parse(string);
            switch (response):
                case ImmediateTask -> this.processQueue.waitAndPut(response);
                case AsyncTask -> {
                    this.asyncQueue.put(response);
                    signalAllCondition(this.notEmpty);
                }
```

Figure 4.7: Pseudocode for reading from process

As Isabelle Client is a subprocess for VeriTest, we can split the input and output stream to efficiently process command invocations. The input stream would act as the producer towards the BlockingQueue, while the output stream would be continuously consumed by a separate thread – as illustrated by figure 4.6 and 4.7. This ensures that command invocations would only be IO bound by the input stream.

Isabelle Server have two types of responses that originates from command invocations:

immediate responses and asynchronous responses [19, Sec 4.2.6]. Immediate responses denote the task identifier to differentiate between asynchronous responses, which is used by `IsabelleProcess`[6]. Asynchronous responses denote the actual progress of the command invocation, identified by their task identifier. As we can see on figure 4.7, the types of responses are differentiated and sent to different queues.

```
class AbstractIsabelleClient implements IsabelleClient:
    var Isabelle: IsabelleProcessInterface;

    ...

    var asyncQueue, notEmpty := Isabelle.open()

    Async<Task> getResult(taskId):
        while true:
            while asyncQueue is empty:
                waitCondition(notEmpty);

            if asyncQueue.peek().taskId == taskId:
                return asyncQueue.take();
```

Figure 4.8: Pseudocode for getting asynchronous results

Asynchronous responses are consumed by the `getResult()` method as illustrated by figure 4.8. Sufficient locking mechanisms without spinning the locks are done to ensure that concurrent processes accessing the same queue would not lead to starvation. As the queue itself is a `BlockingQueue`, and the purpose of the procedure is only to demultiplex the asynchronous responses, merely checking if the task has the correct identifier is enough to ensure mutual exclusion of concurrent processes.

Asynchronous functions are managed by a thread pool, separating IO bound processes into a different executor to ensure freedom from threadpool-induced deadlock. Furthermore, asynchronous functions implement a circuit-breaker pattern when a successful result is available from the function, which can be seen in appendix **??**. Through this, VeriTest is able to provide a *comparatively* fast analysis for optimization rules with only Isabelle as a bottleneck (See chapter 5).

4.5 Generating the Analysis

VeriTest provides RESTful API that encapsulates all the analysis that would be done towards the optimization rule in order to provide the functional requirements of VeriTest, and satisfy the non-functional requirement:

"Developers of GraalVM can easily use this without understanding Isabelle."

The analysis is done by invoking Isabelle's automated tools and pattern matching their response into their classification. The order of which the analysis is carried out can be seen in figure 4.9. After significant evaluation and analysis, we extended their classification into multiple categories (See Fig. 4.10).

[6]IsabelleProcessFacade.java

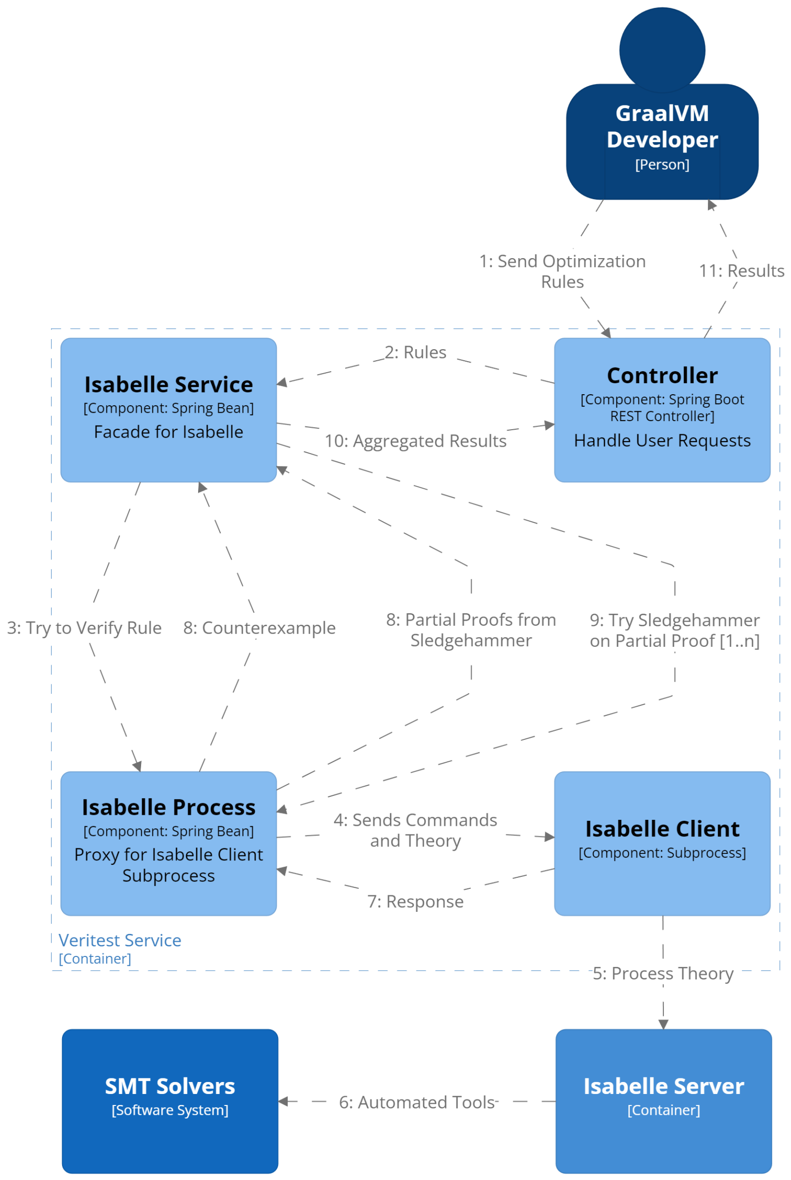

Figure 4.9: Sequence Diagram for VeriTest

4.5.1 Sledgehammer Invocations

It is interesting to note that there are multiple invocations of Sledgehammer in order to find proofs towards an optimization rule. This is due to the fact that Sledgehammer can both return a partial proof towards an optimization rule due to the two proof obligations needed to prove an optimization rule, and return several proof options that can prove an optimization rule. A recursive algorithm is used in order to comprehensively prove an optimization rule, as illustrated by figure 4.11.

4.5.2 Finding Counterexamples through Nitpick & Quickcheck

Finding counterexamples are much more straightforward compared to Sledgehammer invocations. If any of the messages inside Isabelle's response contains a counterexample string, then a counterexample is found. This algorithm works with both Nitpick and Quickcheck invocations. As such, we can use an algorithm such as figure 4.12 to find incorrect optimization

1. The optimization rule is false;

   Quickcheck or Nitpick are able to find a counterexample.

2. The optimization rule is automatically proven;

   Based on the optimization rule, no proof obligation are necessary to proof that the optimization rule is true.

3. The optimization rule has no subgoals;

   The optimization rule should be able to be automatically proven, but Isabelle classified the optimization as an error. It is worthy to note that optimization rules that have no subgoals are specific to Isabelle2023, and should be amended by the next iteration of Isabelle.

4. The optimization rule is proven;

   Sledgehammer is able to prove the optimization rule.

5. The optimization rule is malformed due to syntax errors;

   The optimization rule does not follow the syntax of the DSL.

6. The optimization rule is malformed due to type errors;

   This means that the optimization rule provided has incompatible types.

7. The optimization rule cannot be classified.

   VeriTest is unable to verify the optimization rule, due to the complexity of said rule.

Figure 4.10: Optimization rule analysis classification

rules.

4.5.3 Detecting Malformed Optimization Rules and Automatic Proof

While we utilize Isabelle's automated tools to analyze our optimization rules, invoking a time consuming tool to detect malformed optimization rules is unnecessary. For some of the optimization rules (e.g., Fig. 4.13), the rule is automatically verified by Isabelle due to it not having any necessary proof obligations to prove. As such, we can leverage this fact to invoke a quick and efficient automated tool in the form of a **dot** (.) that only does the reasoning internally. An algorithm such as figure 4.14 allows us to check for malformed optimization rules, type unification errors, and other types of errors.

```
class IsabelleService:
    var Client: IsabelleClient;

    Async<Result> trySledgehammer(Theory):
        return generateTheory(SLEDGEHAMMER, Theory)
            then Client.startSession(request)
            then Client.useTheory(generated)
            then recursiveSledgehammer(Theory, TaskResponse);

    Result recursiveSledgehammer(Theory, TaskResponse):
        /* Base Case: Errors returned from Isabelle */
        if TaskResponse contains errors:
            return FAILED;

        /* Base Case: Found Proof */
        if TaskResponse contains "No Proof State" or "No Subgoal":
            return (FOUND_PROOF, TaskResponse.proofs)

        var possibleProofs := filterProofs(TaskResponse);

        /* Base Case: Proof not found */
        if possibleProofs is empty:
            return FAILED;

        var futures : List;
        for proof in possibleProofs:
            var childRequest := generateChildRequest();
            childRequest.proofs := TaskResponse.proofs + [proof];

            futures.add(trySledgehammer(childRequest));

        do allOfReturnOnSuccess(futures);
        then waitForCompletion();

        var result := filterAnySuccessfulFuture(futures);

        return result;
```

Figure 4.11: Pseudocode for Sledgehammer recursive invocations

4.6 Dependencies and Containerization

VeriTest is available as a containerized application[7] ready for immediate use through Docker[8]. All of the required dependencies of VeriTest are built through multiple build stages in order to reduce the time needed to build VeriTest through build caching. A critical component of VeriTest is utilizing Isabelle2023's docker image to avoid rebuilding Isabelle from scratch [19, Sec. 7.1].

Furthermore, VeriTest pre-builts all of the necessary lemmas and theorems from Veriopt's theory base in order to reduce the time needed to start a session for analyzing an optimization rule. It is important to note that Isabelle sessions needs to be built in the *correct order*, beginning with Isabelle/HOL, dependencies of the Canonicalizations session from Veriopt's

[7]https://hub.docker.com/repository/docker/achmadafriza/veritest
[8]https://www.docker.com/

```
class IsabelleService:
    var Client: IsabelleClient;

    Async<Result> tryNitpick(Theory):
        return generateTheory(NITPICK, Theory);
            then Client.startSession(request);
            then Client.useTheory(generated);
            then (Task) -> {
                if Task contains error:
                    return FAILED;

                if Task.messages contains counterexample:
                    return (FOUND_COUNTEREXAMPLE, counterexample);
                else:
                    return FAILED;
            };

    ...
```

Figure 4.12: Pseudocode for finding counterexamples

$$\textbf{optimization}\ \textit{Auto:}\ (\textit{true}\ ?\ x\ :\ y)\ \longmapsto\ x\ .$$

Figure 4.13: Example of optimization rule that can automatically be verified

```
class IsabelleService:
    var Client: IsabelleClient;

    Async<Result> tryAuto(Theory):
        return generateTheory(AUTO, Theory);
            then Client.startSession(request);
            then Client.useTheory(generated);
            then (Task) -> {
                if Task not contains error:
                    return FOUND_AUTO_PROOF;

                if Task.messages contains "syntax error" or "undefined type":
                    return MALFORMED;

                if Task.messages contains "type unification error":
                    return TYPE_ERROR;

                if Task.messages contains "no subgoal":
                    return NO_SUBGOAL;

                return (FAILED, Task.messages);
            };

    ...
```

Figure 4.14: Pseudocode for finding malformed optimization rules

theory base, to the Canonicalizations session. The detail of the containerization can be seen in appendix **??**.

4.7 Supporting Tool

VeriTest also improves Veriopt's current extraction tool [4, Sec. 7] by creating an extraction script capable of extracting source code annotations inside GraalVM's compiler and extracting Veriopt's existing optimization rules inside their theory base. This extraction tool also serves as a test generation script for VeriTest, converting extracted optimization rules into JSON files. The details of this tool can be seen in appendix **??**.

Chapter 5

Results & Discussion

In order to determine the thoroughness of VeriTest's analysis, we evaluated VeriTest based on Veriopt's current, *proven*, collection of optimization rules inside their theory base [4]. Furthermore, as GraalVM's compiler source code is currently annotated with optimization rules following Veriopt's DSL [4, Sec. 5.1], we can evaluate VeriTest by evaluating the source code annotations through VeriTest.

Evaluation is done by exporting the existing Veriopt's expression canonicalization optimization rules using VeriTest's extraction tool (See Sec. 4.7) and iterating them several times as a benchmark on an AMD Ryzen 9 7900X processor with 64GB of RAM inside a Windows 10 WSL environment.

5.1 Evaluation of Veriopt's Current Theory Base

add table from conference paper

Our findings suggests that there is a baseline runtime for every verification of an optimization rule. This is because, surprisingly, there is a key flaw inside Isabelle Server. Kobschätzki [20] notes that Isabelle Server triggers a race condition with multiple users and sessions, unless a delay of several seconds is placed in between subsequent commands. Isabelle Server's source code also confirms that each command invocation doesn't support concurrent use. Since every command invocation is preceded – and followed by – session invocations, this flaw inside Isabelle Server presents a bottleneck for processing optimization rules.

describe the malformed rule and counterexample

The evaluated runtime suggests a significant variation of runtime for some results. This is due to VeriTest attempting to exhaust every possible proof and counterexample for the optimization rule, as illustrated by section 4.5. As the complexity of optimization rules vary, the depth of the recursive tree depends on the proof obligations that Sledgehammer can prove. Failure to find a proof suggests that it *may* be able to be proven with adequate expertise in formal proofs.

5.1.1 Incomplete Proofs

While we do discover that VeriTest is capable of utilizing existing lemmas inside Veriopt's current theory base to find proofs for the optimization rule, we discovered that *some* of the lemmas are in fact only partially proven and are still on progress. This introduces a moderate drawback for VeriTest. However, this can be remedied by adjusting VeriTest's current algorithm (See Fig. 4.11) to consider Isabelle's usage of partially proven theories, and remove such proof if it exists (See appendix **??**).

add figure for number of optimization rules that still incorporates incomplete lemmas

describe the figure

add figure for results without incorporating incomplete proofs

describe the figure

tell that as the theory base grows, the percentage of proof should also grow

5.2 Evaluation of Malformed Rules

In order to determine the completeness of VeriTest's analysis in finding incorrect optimization rules, we devised a suite of test cases describing malformed optimization rules. These malformed rules are generated through mutation operators devised by Offutt et al. [15]. The mutated optimization rules should represent possible scenarios where a GraalVM developer might make mistakes in the syntax analogous to the Java syntax (i.e., using Logical Or (`||`) instead of Bitwise Or (`|`)). The list of mutation operators applied and the mutated optimization rules can be seen in appendix **??**.

add figure for results for malformed rules

describe the figure

While VeriTest is capable of finding syntax errors, our investigation discovered that VeriTest is unable to discover a counterexample for the majority of the mutated optimization rules. While it may seem trivial intuitively, it does not appear to be so for Isabelle. While the reasons for it are inconclusive, and are in fact beyond the scope of this project, several factors that might explain why.

5.2.1 Limitations for Finding Counterexamples

$$e_1 \sqsupseteq e_2 = (\forall\, m\; p\; v.\; [m,\ p] \vdash e_1 \mapsto v \longrightarrow [m,\ p] \vdash e_2 \mapsto v)$$

Figure 5.1: Term refinement proof obligation, Adapted from [4, Definition 6]

Revisiting Quickcheck's limitations (See Sec. 2.3.4), it is possible that Quickcheck's exhaustive test generator is unable to transform complex conjectures into datatypes that it can generate tests for, even when appropriate constructors are defined. For conditional conjectures for each optimization rule, such as figure 5.1, Quickcheck may exhaust all possible values of IR expressions that define the context of $[m,\ p]$ at the expression level, which it fails to do so.

$$\mathit{val}[e_1] \neq \mathit{UndefVal} \wedge \mathit{val}[e_2] \neq \mathit{UndefVal} \longrightarrow \mathit{val}[e_1] = \mathit{val}[e_2]$$

Figure 5.2: Conditional evaluation of e_1 and e_2

Instead of exhaustively enumerating all the possible contexts for an expression, it may be possible for Quickcheck to conditionally evaluate the given conjecture as a value, as illustrated in figure 5.2. As long as e_1 and e_2 are not undefined, the values of e_1 and e_2 are compared to be equal or not. This semi-automatic method of conditionally evaluating expression semantics would need VeriTest to be able to parse GraalVM's IR DSL and statically analyze and construct appropriate equations in order to verify it, which may represent potential future work to be done.

summary of what a quotient type is –> should i put this in the appendix?

In the case of Nitpick, extensive debugging suggests that there may be an inherent flaw inside either the encoding of GraalVM's expression semantics. Inside Veriopt's DSL, the encoding for GraalVM's IR expressions is defined as functions and datatypes that would influence how Isabelle would reason about it. Such definitions would be used to translate optimization rules into conjunctions for SMT solvers to find satisfiability. However, we found that among the translated conjunctions, especially for the term refinement proof obligation, Nitpick is unable to support *representative functions* to properly map an Integer quotient type (See appendix **??**) into the respective first-order relational logic [21, Ch. 8]. This is explicitly mentioned as a known bug inside Nitpick, and should be amended by defining a *term postprocessor* that converts such quotient type into a standard mathematical notation [21, Sec. 3.7] or correcting underspecified functions [21, Ch. 8]. To quote from Blanchette [21, Ch. 8]:

> "Axioms or definitions that restrict the possible values of the undefined constant or other partially specified built-in Isabelle constants (e.g., *Abs_* and *Rep_* constants) are in general ignored. Again, such nonconservative extensions are generally considered bad style."

5.3 Evaluation of GraalVM's Source Code Annotations

add table of graal annotation evaluation

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Chapter 6

Recommendations & Future Work

Maecenas non massa. Vestibulum pharetra nulla at lorem. Duis quis quam id lacus dapibus interdum. Nulla lorem. Donec ut ante quis dolor bibendum condimentum. Etiam egestas tortor vitae lacus. Praesent cursus. Mauris bibendum pede at elit. Morbi et felis a lectus interdum facilisis. Sed suscipit gravida turpis. Nulla at lectus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Praesent nonummy luctus nibh. Proin turpis nunc, congue eu, egestas ut, fringilla at, tellus. In hac habitasse platea dictumst.

Vivamus eu tellus sed tellus consequat suscipit. Nam orci orci, malesuada id, gravida nec, ultricies vitae, erat. Donec risus turpis, luctus sit amet, interdum quis, porta sed, ipsum. Suspendisse condimentum, tortor at egestas posuere, neque metus tempor orci, et tincidunt urna nunc a purus. Sed facilisis blandit tellus. Nunc risus sem, suscipit nec, eleifend quis, cursus quis, libero. Curabitur et dolor. Sed vitae sem. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Maecenas ante. Duis ullamcorper enim. Donec tristique enim eu leo. Nullam molestie elit eu dolor. Nullam bibendum, turpis vitae tristique gravida, quam sapien tempor lectus, quis pretium tellus purus ac quam. Nulla facilisi.

Duis aliquet dui in est. Donec eget est. Nunc lectus odio, varius at, fermentum in, accumsan non, enim. Aliquam erat volutpat. Proin sit amet nulla ut eros consectetuer cursus. Phasellus dapibus aliquam justo. Nunc laoreet. Donec consequat placerat magna. Duis pretium tincidunt justo. Sed sollicitudin vestibulum quam. Nam quis ligula. Vivamus at metus. Etiam imperdiet imperdiet pede. Aenean turpis. Fusce augue velit, scelerisque sollicitudin, dictum vitae, tempor et, pede. Donec wisi sapien, feugiat in, fermentum ut, sollicitudin adipiscing, metus.

Donec vel nibh ut felis consectetuer laoreet. Donec pede. Sed id quam id wisi laoreet suscipit. Nulla lectus dolor, aliquam ac, fringilla eget, mollis ut, orci. In pellentesque justo in ligula. Maecenas turpis. Donec eleifend leo at felis tincidunt consequat. Aenean turpis metus, malesuada sed, condimentum sit amet, auctor a, wisi. Pellentesque sapien elit, bibendum ac, posuere et, congue eu, felis. Vestibulum mattis libero quis metus scelerisque ultrices. Sed purus.

Donec molestie, magna ut luctus ultrices, tellus arcu nonummy velit, sit amet pulvinar elit justo et mauris. In pede. Maecenas euismod elit eu erat. Aliquam augue wisi, facilisis congue, suscipit in, adipiscing et, ante. In justo. Cras lobortis neque ac ipsum. Nunc fermentum massa at ante. Donec orci tortor, egestas sit amet, ultrices eget, venenatis eget, mi. Maecenas vehicula leo semper est. Mauris vel metus. Aliquam erat volutpat. In rhoncus sapien ac tellus. Pellentesque ligula.

Cras dapibus, augue quis scelerisque ultricies, felis dolor placerat sem, id porta velit odio eu elit. Aenean interdum nibh sed wisi. Praesent sollicitudin vulputate dui. Praesent iaculis viverra augue. Quisque in libero. Aenean gravida lorem vitae sem ullamcorper cursus. Nunc adipiscing rutrum ante. Nunc ipsum massa, faucibus sit amet, viverra vel, elementum semper, orci. Cras eros sem, vulputate et, tincidunt id, ultrices eget, magna. Nulla varius ornare odio.

Donec accumsan mauris sit amet augue. Sed ligula lacus, laoreet non, aliquam sit amet, iaculis tempor, lorem. Suspendisse eros. Nam porta, leo sed congue tempor, felis est ultrices eros, id mattis velit felis non metus. Curabitur vitae elit non mauris varius pretium. Aenean lacus sem, tincidunt ut, consequat quis, porta vitae, turpis. Nullam laoreet fermentum urna. Proin iaculis lectus.

Sed mattis, erat sit amet gravida malesuada, elit augue egestas diam, tempus scelerisque nunc nisl vitae libero. Sed consequat feugiat massa. Nunc porta, eros in eleifend varius, erat leo rutrum dui, non convallis lectus orci ut nibh. Sed lorem massa, nonummy quis, egestas id, condimentum at, nisl. Maecenas at nibh. Aliquam et augue at nunc pellentesque ullamcorper. Duis nisl nibh, laoreet suscipit, convallis ut, rutrum id, enim. Phasellus odio. Nulla nulla elit, molestie non, scelerisque at, vestibulum eu, nulla. Ut odio nisl, facilisis id, mollis et, scelerisque nec, enim. Aenean sem leo, pellentesque sit amet, scelerisque sit amet, vehicula pellentesque, sapien.

Chapter 7

Conclusion

Conclusion: what conclusions can be drawn from the results of your research?

7.1 Summary & Conclusions

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetuer nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.

Sed eleifend, eros sit amet faucibus elementum, urna sapien consectetuer mauris, quis egestas leo justo non risus. Morbi non felis ac libero vulputate fringilla. Mauris libero eros, lacinia non, sodales quis, dapibus porttitor, pede. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi dapibus mauris condimentum nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam sit amet erat. Nulla varius. Etiam tincidunt dui vitae turpis. Donec leo. Morbi vulputate convallis est. Integer aliquet. Pellentesque aliquet sodales urna.

7.2 Possible Future Work

Nullam eleifend justo in nisl. In hac habitasse platea dictumst. Morbi nonummy. Aliquam ut felis. In velit leo, dictum vitae, posuere id, vulputate nec, ante. Maecenas vitae pede nec dui dignissim suscipit. Morbi magna. Vestibulum id purus eget velit laoreet laoreet. Praesent sed leo vel nibh convallis blandit. Ut rutrum. Donec nibh. Donec interdum. Fusce sed pede sit amet elit rhoncus ultrices. Nullam at enim vitae pede vehicula iaculis.

Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Aenean nonummy turpis id odio. Integer euismod imperdiet turpis. Ut nec leo nec diam imperdiet lacinia. Etiam eget lacus eget mi ultricies posuere. In placerat tristique tortor. Sed porta vestibulum metus. Nulla iaculis sollicitudin pede. Fusce luctus tellus in dolor. Curabitur auctor velit a sem. Morbi sapien. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Donec adipiscing urna vehicula nunc. Sed ornare leo in leo. In rhoncus leo ut dui. Aenean dolor quam, volutpat nec, fringilla id, consectetuer vel, pede.

Nulla malesuada risus ut urna. Aenean pretium velit sit amet metus. Duis iaculis. In hac habitasse platea dictumst. Nullam molestie turpis eget nisl. Duis a massa id pede dapibus ultricies. Sed eu leo. In at mauris sit amet tortor bibendum varius. Phasellus justo risus, posuere in, sagittis ac, varius vel, tortor. Quisque id enim. Phasellus consequat, libero pretium nonummy fringilla, tortor lacus vestibulum nunc, ut rhoncus ligula neque id justo. Nullam accumsan euismod nunc. Proin vitae ipsum ac metus dictum tempus. Nam ut wisi. Quisque tortor felis, interdum ac, sodales a, semper a, sem. Curabitur in velit sit amet dui tristique sodales. Vivamus mauris pede, lacinia eget, pellentesque quis, scelerisque eu, est. Aliquam risus. Quisque bibendum pede eu dolor.

Chapter 8

Bibliography

[1] N. P. Lopes and J. Regehr, “Future directions for optimizing compilers,” arXiv preprint, Sep. 2018. arXiv: `1809.02161 [cs.PL]`.

[2] N. J. Wahl, “An overview of regression testing,” *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 1, pp. 69–73, Jan. 1999, ISSN: 0163-5948. DOI: `10.1145/308769.308790`.

[3] X. Leroy, “Formal verification of a realistic compiler,” *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, ISSN: 0001-0782, 1557-7317. DOI: `10.1145/1538788.1538814`.

[4] B. J. Webb, I. J. Hayes, and M. Utting, “Verifying term graph optimizations using Isabelle/HOL,” in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2023, pp. 320–333, ISBN: 9798400700262. DOI: `10.1145/3573105.3575673`.

[5] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic* (Lecture Notes in Computer Science). Berlin, Heidelberg: Springer, 2002, vol. 2283, ISBN: 3-540-43376-7. DOI: `10.1007/3-540-45949-9`.

[6] J. Lee, C.-K. Hur, and N. P. Lopes, “AliveInLean: A verified LLVM peephole optimization verifier,” in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds., 2019, pp. 445–455, ISBN: 978-3-030-25543-5. DOI: `10.1007/978-3-030-25543-5_25`.

[7] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, “Alive2: Bounded translation validation for LLVM,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Virtual Canada: ACM, Jun. 2021, pp. 65–79, ISBN: 978-1-4503-8391-2. DOI: `10.1145/3453483.3454030`.

[8] Oracle, *GraalVM: Run programs faster anywhere*, 2020. [Online]. Available: `https://github.com/oracle/graal` (visited on 09/13/2023).

[9] B. J. Webb, M. Utting, and I. J. Hayes, “A formal semantics of the GraalVM intermediate representation,” in *Automated Technology for Verification and Analysis*, Z. Hou and V. Ganesh, Eds., ser. LNCS, vol. 12971, Oct. 2021, pp. 111–126, ISBN: 978-3-030-88885-5. DOI: `10.1007/978-3-030-88885-5_8`.

[10] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds., *Structured programming*. GBR: Academic Press Ltd., 1972, ISBN: 978-0-12-200550-3.

[11] W. M. McKeeman, “Differential Testing for Software,” vol. 10, no. 1, pp. 100–107, 1998.

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, New York, NY, USA: Association for Computing Machinery, Jun. 2011, pp. 283–294, ISBN: 978-1-4503-0663-8. DOI: `10.1145/1993498.1993532`.

[13] L. Bulwahn, "The New Quickcheck for Isabelle," en, in *Certified Programs and Proofs*, C. Hawblitzel and D. Miller, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2012, pp. 92–108, ISBN: 978-3-642-35308-6. DOI: `10.1007/978-3-642-35308-6_10`.

[14] J. C. Blanchette, L. Bulwahn, and T. Nipkow, "Automatic Proof and Disproof in Isabelle/HOL," en, in *Frontiers of Combining Systems*, ser. LNCS, C. Tinelli and V. Sofronie-Stokkermans, Eds., vol. 6989, 2011, pp. 12–27, ISBN: 978-3-642-24363-9 978-3-642-24364-6. DOI: `10.1007/978-3-642-24364-6_2`.

[15] J. Offutt, P. Ammann, and L. Liu, "Mutation Testing implements Grammar-Based Testing," en, in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, Raleigh, NC, USA: IEEE, Nov. 2006, pp. 12–12, ISBN: 978-0-7695-2897-7. DOI: `10.1109/MUTATION.2006.11`. [Online]. Available: `http://ieeexplore.ieee.org/document/4144731/` (visited on 05/25/2024).

[16] *The LLVM Compiler Infrastructure Project.* [Online]. Available: `https://llvm.org/` (visited on 08/20/2023).

[17] J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending Sledgehammer with SMT Solvers," en, *Journal of Automated Reasoning*, vol. 51, no. 1, pp. 109–128, Jun. 2013, ISSN: 0168-7433, 1573-0670. DOI: `10.1007/s10817-013-9278-5`.

[18] J. Biendarra, J. Blanchette, M. Desharnais, *et al.*, "Dening (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL," en, May 2024. [Online]. Available: `https://isabelle.in.tum.de/dist/doc/datatypes.pdf` (visited on 05/26/2024).

[19] M. Wenzel, "The Isabelle System Manual," en, [Online]. Available: `https://isabelle.in.tum.de/dist/Isabelle2023/doc/system.pdf` (visited on 05/06/2024).

[20] J. Kobschätzki, *Unexpected Behavior with Isabelle Server 2023*, Jan. 2024. [Online]. Available: `https://lists.cam.ac.uk/sympa/arc/cl-isabelle-users/2024-01/msg00006.html` (visited on 05/06/2024).

[21] J. Blanchette, "A Users Guide to Nitpick for Isabelle/HOL," en, [Online]. Available: `https://mirror.cse.unsw.edu.au/pub/isabelle/dist/Isabelle2022/doc/nitpick.pdf` (visited on 09/13/2023).

Appendices

Appendix A

Example Appendix Item

Appendix: Appendices are useful for supplying necessary details or explanations which do not seem to fit into the main text, perhaps because they are too long and would distract the reader from the central argument. Appendices are also used for program listings.

Quisque facilisis auctor sapien. Pellentesque gravida hendrerit lectus. Mauris rutrum sodales sapien. Fusce hendrerit sem vel lorem. Integer pellentesque massa vel augue. Integer elit tortor, feugiat quis, sagittis et, ornare non, lacus. Vestibulum posuere pellentesque eros. Quisque venenatis ipsum dictum nulla. Aliquam quis quam non metus eleifend interdum. Nam eget sapien ac mauris malesuada adipiscing. Etiam eleifend neque sed quam. Nulla facilisi. Proin a ligula. Sed id dui eu nibh egestas tincidunt. Suspendisse arcu.

Maecenas dui. Aliquam volutpat auctor lorem. Cras placerat est vitae lectus. Curabitur massa lectus, rutrum euismod, dignissim ut, dapibus a, odio. Ut eros erat, vulputate ut, interdum non, porta eu, erat. Cras fermentum, felis in porta congue, velit leo facilisis odio, vitae consectetuer lorem quam vitae orci. Sed ultrices, pede eu placerat auctor, ante ligula rutrum tellus, vel posuere nibh lacus nec nibh. Maecenas laoreet dolor at enim. Donec molestie dolor nec metus. Vestibulum libero. Sed quis erat. Sed tristique. Duis pede leo, fermentum quis, consectetuer eget, vulputate sit amet, erat.

Donec vitae velit. Suspendisse porta fermentum mauris. Ut vel nunc non mauris pharetra varius. Duis consequat libero quis urna. Maecenas at ante. Vivamus varius, wisi sed egestas tristique, odio wisi luctus nulla, lobortis dictum dolor ligula in lacus. Vivamus aliquam, urna sed interdum porttitor, metus orci interdum odio, sit amet euismod lectus felis et leo. Praesent ac wisi. Nam suscipit vestibulum sem. Praesent eu ipsum vitae pede cursus venenatis. Duis sed odio. Vestibulum eleifend. Nulla ut massa. Proin rutrum mattis sapien. Curabitur dictum gravida ante.