THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

VeriTest: Automated Testing Framework for Veriopt's DSL

Project Proposal

by

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

School of Electrical Engineering and Computer Science,
University of Queensland

Supervisor

Brae J. Webb, Mark Utting, Ian J. Hayes
School of EECS
{B.Webb,M.Utting,Ian.Hayes}@uq.edu.au

21 September 2023

Contents

1 Introduction

While compilers are generally believed to be correct, unwanted errors would sometimes happen. This is due to: common pitfalls of the language it's written in are *just* accepted by the community; edge cases of the language are not well considered; or by 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 a 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]. This project follows up on previous works done to introduce formal semantics for GraalVM's [4] Intermediate Representation (IR) [5], [6] implemented in Isabelle [7]. There are similar works that have been done, i.e. CompCert [3] and Alive [8], [9]; all of which integrates the theoretical aspects of formal verification into the practicality of using it in a production setting.

This project will attempt to bridge the subset of the gap between the formal semantics of GraalVM and integrating it into GraalVM's test suite [6]. Hence, the project introduces VeriTest: an Automated Testing Framework for GraalVM's optimization DSL. VeriTest will represent a fast feedback tool that GraalVM developers could integrate 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. Referring to the current development workflow of GraalVM [6, Sec. 5.1]:

1. GraalVM developers write or modify an optimization proof – written in Veriopt's GraalVM IR DSL – as a comment in the code.
2. The DSL would be passed into VeriTest by the developers with the following goals:
   (a) Find fast, obvious, feedback towards the DSL rule;
   (b) If it can't find any, then it would, later on, be passed to *"proof experts"*.

Part 2 of the workflow would be the main questions that this project will attempt to solve, mainly:

1. Is it possible to extend the current workings of Veriopt in Isabelle to a tool that could provide feedback for the developers?

   The feasibility of extending Isabelle will need to be explored as part of the project. There are several options for the project to explore (in order of ideal solutions):

   (a) Utilizing `Isabelle Server` - `Client` interactions [10, Ch. 4] to generate a test suite and simple proofs [11]–[14] (See Sec. 3.2);
   (b) Extend the system of `Isabelle/Scala` to utilize the full functionality of Isabelle [10, Ch. 5] (See Sec. 3.3);
   (c) Creating an interpreter for the DSL and applying a set of rules as a test suite (See Sec. 3.4).

2. Would the feedback be useful to GraalVM developers? (See Sec. 3.6)

3. How fast could the feedback be provided to GraalVM developers? (See Sec. 3.6)

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, pp. 5] – similar to Alive2's [9] solution on formal verification (See Sec. 2.3.2); perhaps a future direction in Veriopt.

Furthermore, to explore the viability of this project, we need to consider past relevant works that have been done. Namely, we need to explore:

1. The types of software testing done in the industry, and why it's unreliable (See Sec. 2.1);
2. How to leverage automated tools such as Isabelle (See Sec. 2.2);
3. How to verify the correctness of compiler (See Sec. 2.3);
4. How different projects vary in their approach to verifying the correctness of a compiler (See Sec. 2.3.1, 2.3.2, and 2.3.3).

2 Background

2.1 Software Testing

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 [15].

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 [15]. For example, [16, Sec. 2] would generate test cases and execute 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 [15], 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** would be preferable in this project.

2.1.3 Inference-based Exhaustive Testing

Exhaustive tests such as [11] 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 would be as simple as determining if the bounds of a variable would 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 would be incorrect [12, Ch. 5]. Exhaustive searching allows developers to only focus on defining the behavior of the program, rather than defining tests to define the behavior.

2.2 Isabelle

Isabelle is an interactive theorem prover that utilizes a multitude of tools for automatic proving. It emphasizes breaking down a proof for a theorem towards multiple smaller goals that are achievable called tactics. Tactics are functions written in the implementation of Isabelle that work on a proof state [12]. It either outputs a direct proof towards the goal or breaks it 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.

Finding the right proving methods and arguments to utilize is one of the biggest issues for proving a theorem [12]. There are many tools in Isabelle's arsenal that can help the user progress towards their proof [7]. However, the most notable ones are Sledgehammer, Quickcheck, and Nitpick.

2.2.1 Sledgehammer

Sledgehammer is one of the tools in Isabelle that *could* automatically prove a theorem. It utilizes the set of inference rules as conjectures which can be cross-referenced with relevant facts (lemmas, definitions, or axioms) from Isabelle [12, Sec. 3]. Afterwards, Sledgehammer passes them into resolution provers and SMT solvers which try to solve it [12, Sec. 3.3]. If reasonable proof is found, Sledgehammer would then reconstruct the inference rules back into a *relatively* human-readable proof definition in the style of Isabelle/Isar [12, 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. notes [12, pp. 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 automatic theory proving with counterexample generators such as Quickcheck and Nitpick.

2.2.2 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 [11]. 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 [11].

Random testing of a conjecture assigns free variables with pseudo-random values [11, 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 proof definitions that are well defined [11]. 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 [11]. However, exhaustive test data generators would fail to find counterexamples of proofs that have unbounded variables. Narrowing test data generators would improve on that by evaluating proof definitions symbolically rather than taking variables at face value. This is possible due to term rewriting static analysis done on the proof definitions [11, 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 [11, Sec. 7]. As much of the proof definitions are defined over unbounded variables, exhaustive testing is the default option for Quickcheck.

2.2.3 Nitpick

An alternative to find counterexamples for proof definitions would be Nitpick. 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 [12, Sec. 5]. SAT solvers would then search for premises that would falsify the given conjecture. If a conjecture has bounds over finite domains, Nitpick would *eventually* find the counterexample. Conjectures with unbounded variables would be partially evaluated [12, Sec. 5.2]. However, it could not determine whether infinite bounds would 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 would be possible to prove [11]. The performance of Nitpick is comparable to Quickcheck, with Nitpick being able to find counterexamples to proof definitions that are not executable within Isabelle's code generation [11, Sec. 7].

2.3 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 would match the input program exactly. However, this is usually not the case [3]. Chefs have their own 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 would try to make them faster in their own 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], [6], [8], [9].

Verifying compiler correctness is not easy. While the correctness of a software *could* be verified by defining behaviors through regression testing, compiler bugs are infamous for being hard to spot [2], [3]. Hence, it would 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 2.3.1) – creating a completely verified compiler for the C language platform. Formally verifying compilers on the scale of CompCert would require vast amounts of time and resources, which projects often don't have.

As such, there are smaller-scale projects such as Alive [8] & Alive2 [9] that focus on behavior translation errors in LLVM's peephole optimizer (See 2.3.2). Veriopt tries 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 [5], [6] (See 2.3.3).

2.3.1 CompCert

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

1. With deterministic programs, a compiler would compile 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 phases must be accompanied by the formal definition of their Intermediate Representation (IR) semantics [3], [5].

3. Lastly, to formally verify each of the optimization phases, 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 2.3.3).

   (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 2.3.2).

CompCert formally verifies each step in the source, intermediate, and target languages that goes 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 2.2) [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 3rd 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 would require an enormous amount of work. Formal verifications such as Alive (See Sec. 2.3.2) and Veriopt (See Sec. 2.3.3) undertakes the smaller subset, namely 1st and 2nd step, of compiler verification research thread [1].

2.3.2 Alive

Alive tackles a subset of compiler verification by verifying that code optimizations inside LLVM are correct [8]. For example, a compiler would optimize $(LHS = x * 2) \implies (RHS = x << 1)$ (LHS is transformed into RHS) [8, 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.

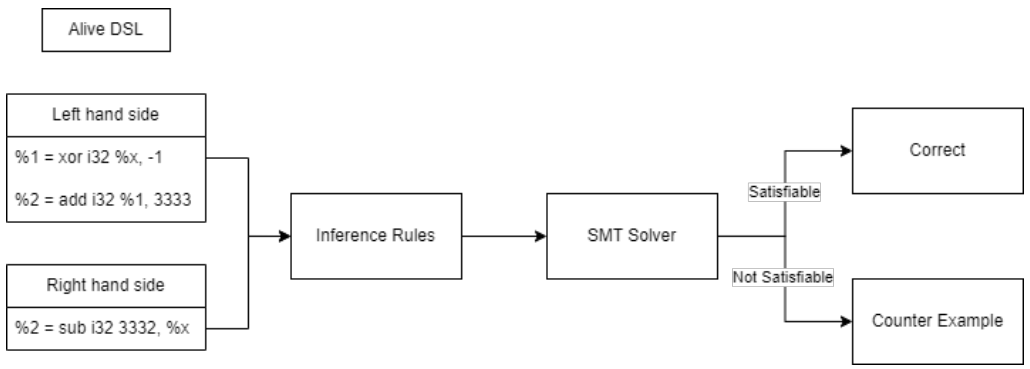

Figure 1: How Alive verifies $(LHS = (X \oplus -1) + C) \implies (RHS = (C - 1) - X)$ [8, pp. 1]

To verify that code optimizations are correct, Alive utilizes inference-based exhaustive testing (See 2.1.3) that allows the tool to encode machine code behaviors to inference rules. These inference rules are then passed to SMT solvers to check for their satisfiability (See

Fig. 1). If the translation is proven to be correct, then it would mean that the underlying optimization code is correct.

Alive encodes LLVM's [18] underlying IR semantics through their own DSL [8, 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 would represent 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 [9]. This allows the developers to entirely focus on developing LLVM, while completely ignoring the specifications of Alive. This has been found to be effective, as differential testing of LLVM's unit tests and Alive2 discovers multiple errors inside the unit test behaviors itself [9, Sec. 8.2]. Alive & Alive2 covers the whole 1st and 2nd step of compiler optimizations research thread [1, pp. 5].

2.3.3 Veriopt

In comparison to CompCert and Alive, the theoretical aspects of compiler verification are really 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 [5]. However, Veriopt's DSL only concerns the subset of GraalVM's IR, which is the side-effect-free data-flow nodes [6]. Side-effect-free data-flow nodes are comparatively easier to prove and optimize, as it would be considered defined – as opposed to LLVM's undefined and poisoned variables [9].

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

Termination Proof Obligation $trm(x) < trm(BinaryExprBinAdd(BinaryExprBinSubxy)y)$

Refinement Proof Obligation *BinaryExpr BinAdd (BinaryExpr BinSub x y) y* $\sqsupseteq x$

Figure 2: Sample of Veriopt's DSL [6, Fig. 3]

fix this formatting

Figure 2 defines the structure of DSL for an optimization phase. The ***optimization*** keyword represents the proof definition that must be proven in Isabelle. Note that there are 2 proof obligations that must be met in order to consider that the side-effect-free optimization is correct: proof that the optimization phase would terminate; proof that each pass of the optimization phase would result in a refinement of the expression [6]. Note that these proofs would need to be provided by the users.

Currently, there are some tools that the developers of GraalVM could use to provide a *certificate* towards the compiler code [6, Sec. 7]. A semi-automated approach exists in the form of source code annotations [6, Sec. 5.1]. However, integrating new behaviors that would require new *certificates* would be challenging, as the approach would only describe the behavior of the code – instead of formally proving that the behavior is indeed correct.

Providing proof obligations for an optimization phase would be challenging for developers who are not *experts in program verification*. Veriopt's DSL is implemented in Isabelle [7], which comes with tools that assist in proving higher order logic (See 2.2). However, using such tools would require the developers of GraalVM to be familiar with Isabelle – something that ideally Veriopt would like to avoid. Similar tools such as Alive [8] would be preferable. Hence, that's where this project would like to contribute.

3 Methodology

The goal of this project is to create VeriTest: an Automated Testing Framework for GraalVM's optimization DSL. This will represent another tool that the developers of GraalVM could use to provide a *certificate* towards their implementation code. To assist in verifying the optimization phase, the tool would need to be able to classify each of the optimization phases (See Fig. 3). Furthermore, there are several key non-functional requirements for the system that need to be satisfied (See Fig. 3).

1. The optimization phase is obviously false;

   This would require the tool to generate obvious counterexamples via Quickcheck (See 2.2.2) or Nitpick (See 2.2.3).

2. The optimization phase is obviously true;

   This would require the tool to verify that Sledgehammer (See 2.2.2) can provide proof obligations for the optimization phase **without** dynamically defining proof tactics.

3. The optimization phase 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 subgoals to be defined.

Figure 3: Classification of an optimization phase

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 4: Non-functional requirements of the Framework

To implement the tool, it would require the project to explore whether it is even possible to utilize Isabelle as the core system of the framework. Hence, understanding Isabelle's implementation is crucial in order to achieve the goals of this project. At a glance, there are three options to the project:

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

Figure 5 depicts the overview of the proposed solution's system landscape. In theory, it would utilize Isabelle in a similar manner as Isabelle/jEdit [10]. Therefore, it should be able to use the same Isabelle functionalities as Isabelle/jEdit does.

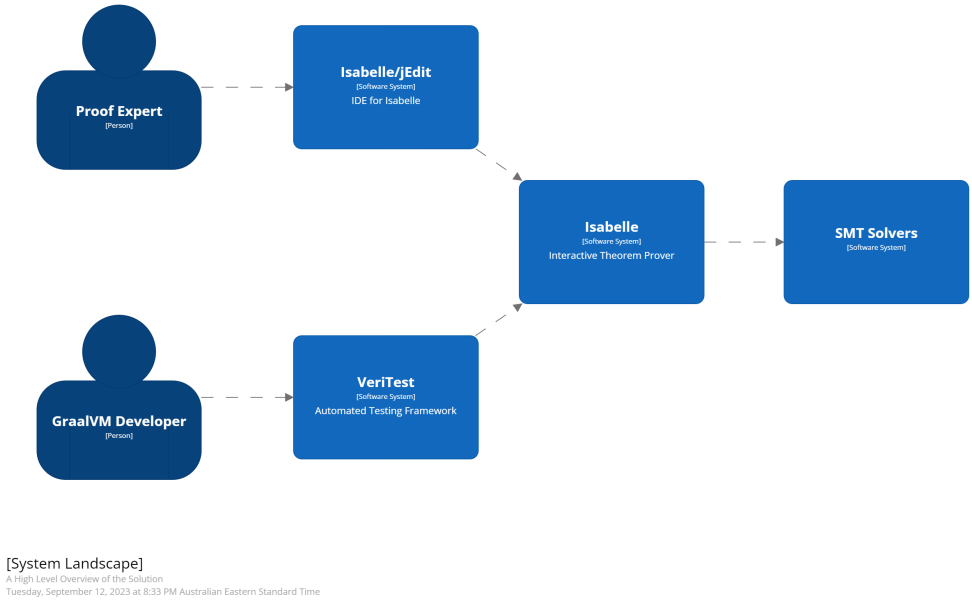

Figure 5: Proposed Solution

3.1 Isabelle System Overview

Isabelle is made up of two significant components: Isabelle/ML and Isabelle/Scala [10, Ch. 5]. 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 the 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 [10, Ch. 4]. Interactions to Isabelle/Server would require a duplex socket connection over TCP [10, Ch. 4.2]. To simplify the communication between the framework and Isabelle/Server, we utilize Isabelle Client [10, Ch. 4.1.2] – a proxy for Isabelle/Server that handles all the communication protocols of Isabelle/Server.

Isabelle Server can load theorems and process requests in parallel [10, Ch. 4.2.6]. As such, this solution would be ideal for the project, as it would allow the framework to offload the computing resources of loading and processing optimization proofs on external sites. However, it would require a *Facade* that's able to demultiplex asynchronous messages on Isabelle Client (See Fig. 7). Furthermore, the full capabilities of Isabelle Server need to be explored in order to implement this option.

3.3 Extending Isabelle/Scala

Isabelle can be extended by accessing Isabelle/Scala functions [10, Ch. 5]. Extending Isabelle/Scala would require the framework to utilize Isablle's Scala compiler [10, Ch. 5.1.4]. Consequently, it means that Isabelle/ML would be bundled with Isabelle/Scala. As such, this option would be able to utilize all the functionalities of Isabelle. Figure 8 depicts the proposed solution for this option.

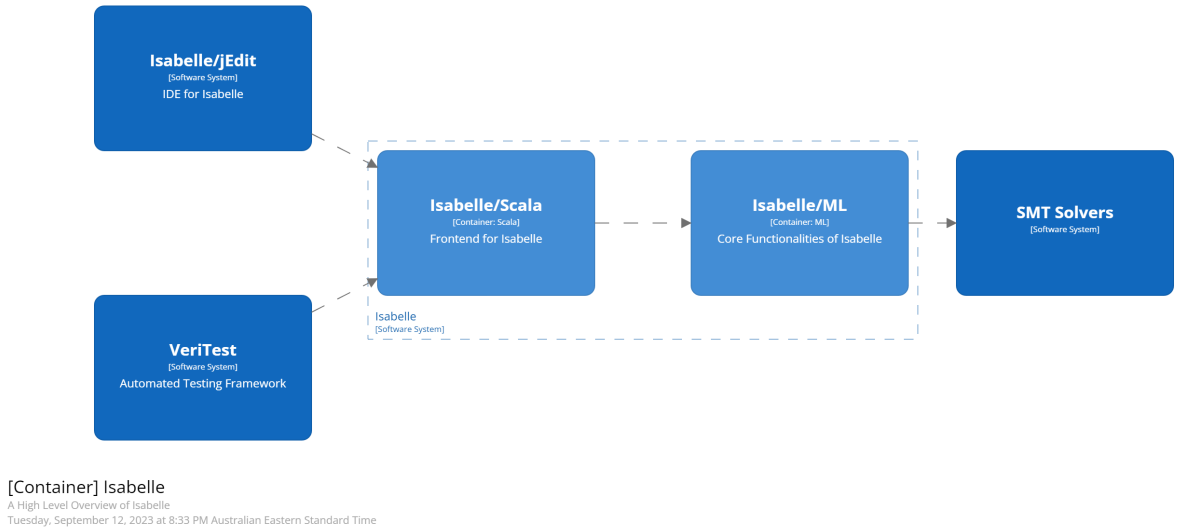

Figure 6: Isabelle System Overview

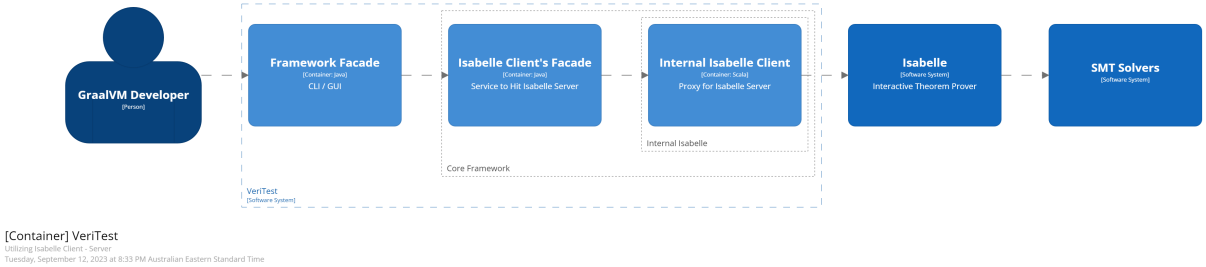

Figure 7: Utilizing Isabelle Client - Server interaction

However, the full complexity of Isabelle/Scala is unknown. Currently, Isabelle/Scala is not well documented and would require much of the project's timeline to understand Isabelle/Scala. 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 Interpreter for DSL

Building an interpreter for GraalVM's optimization DSL acts as a last resort to the project. In order 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 2.2.2) 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 *if possible*.

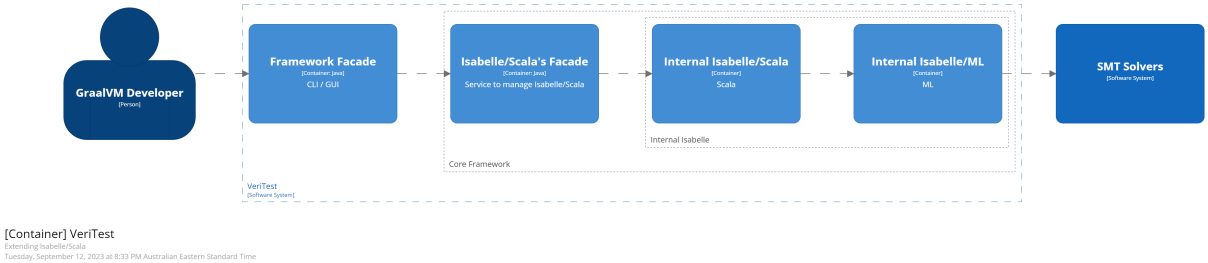

Figure 8: Extending Isabelle/Scala

3.5 User Interaction

The user's experience in using the framework is out of the scope of this project. However, the full capabilities of the system must be demonstrated by how the user interacts with the framework; i.e. parallel processing of an optimization phase. Therefore, a frontend that could showcase that would be a RESTful API, where there can be multiple users using the same system with the same capabilities. However, it is expected that each of the classifications that is run inside the framework shouldn't interfere with one another. This means that each user request should be stateless.

3.6 Evaluation

To evaluate the usefulness of the framework, the project would need to determine whether the proposed solution could classify an optimization phase, following Fig. 3. Each of the test cases is then evaluated based on their accuracy of classification. To generate each of the test cases, the project could refer to Veriopt's current workings on the optimization phase proofs and modify them. This would answer the 2^{nd} research question of this project.

To evaluate the 3^{rd} research question of this project, we could simulate the GraalVM developer's usage of VeriTest multiple times in the same environment, averaging the running time of each simulation. In essence, we would be stress-testing the tool to see its limitations. To stress-test the framework, we could use the same test cases over concurrent simulations, as it is expected for the system to compartmentalize each request.

4 Project Plan

This project would require several milestones to be completed:

1. Creating a Proof of Concept (PoC) for VeriTest;
2. Refining the PoC to be usable in a production setting (November - February).

However, due to the nature of the possible solutions of the framework, it could mean that the details and deadline of the milestones could shift dramatically, as each of the solutions (See Sec. 3) differs in their concept. Therefore, each of the milestones would depict several common goals that need to be achieved in order to provide value to the project. To illustrate, Fig. 4, 4, and 4 have timelines for each of the possible solutions.

Furthermore, it would require a considerable amount of time for me to complete the requirements for the course. Therefore, the timelines of each milestones should be over-estimated by some factor to allow myself to cope with the demands of thesis and other courses requirements. These requirements would be done alongside the project milestones, namely:

1. Progress Seminar (Due 9 October 2023);
2. Conference Paper (Due 9 May 2024);
3. Poster & Demonstration (Due 17 May 2024);
4. Thesis Submission (Due 3 June 2024).

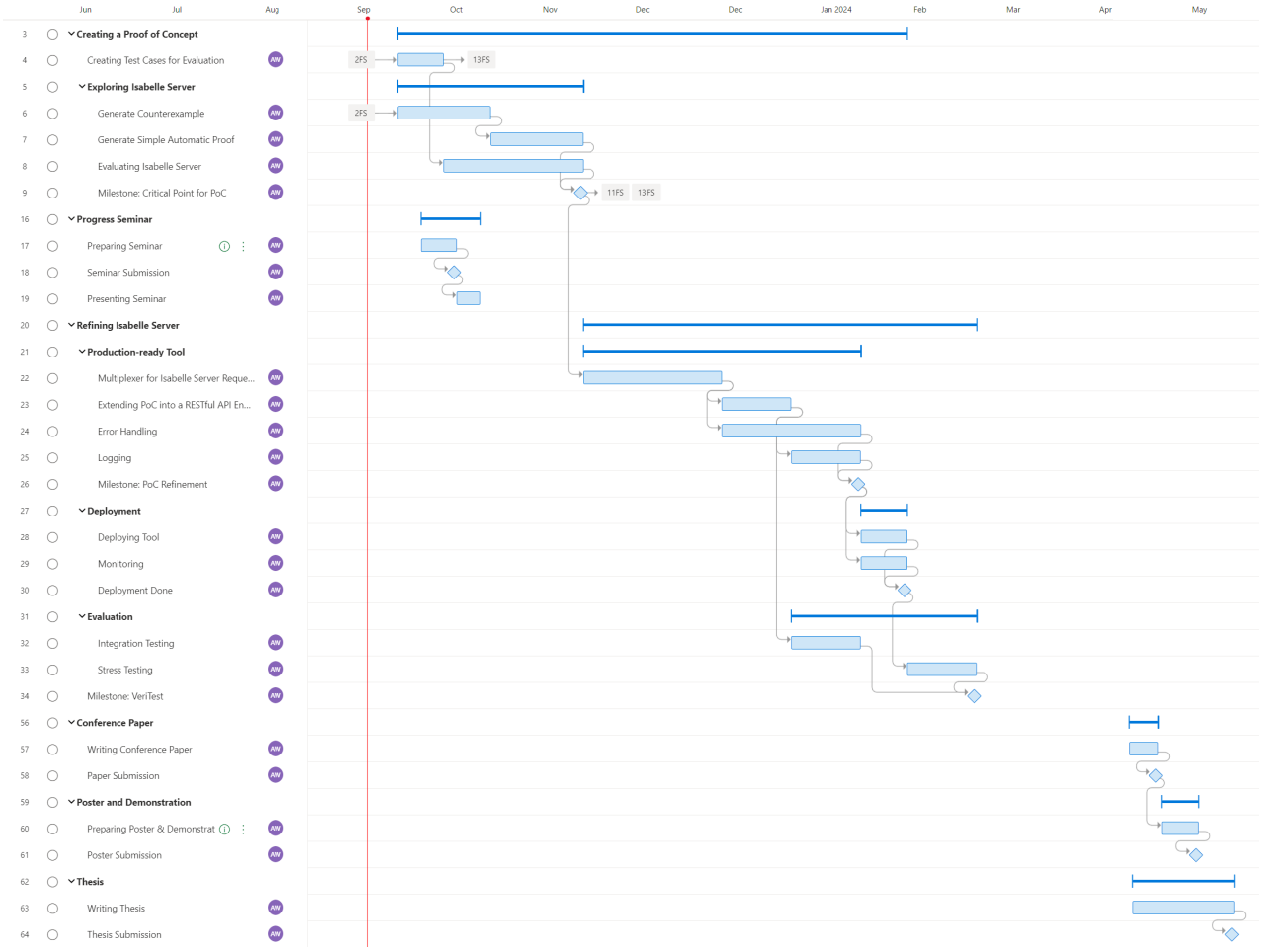

Figure 9: Project timeline for Isabelle Server solution (See Sec. 3.2)

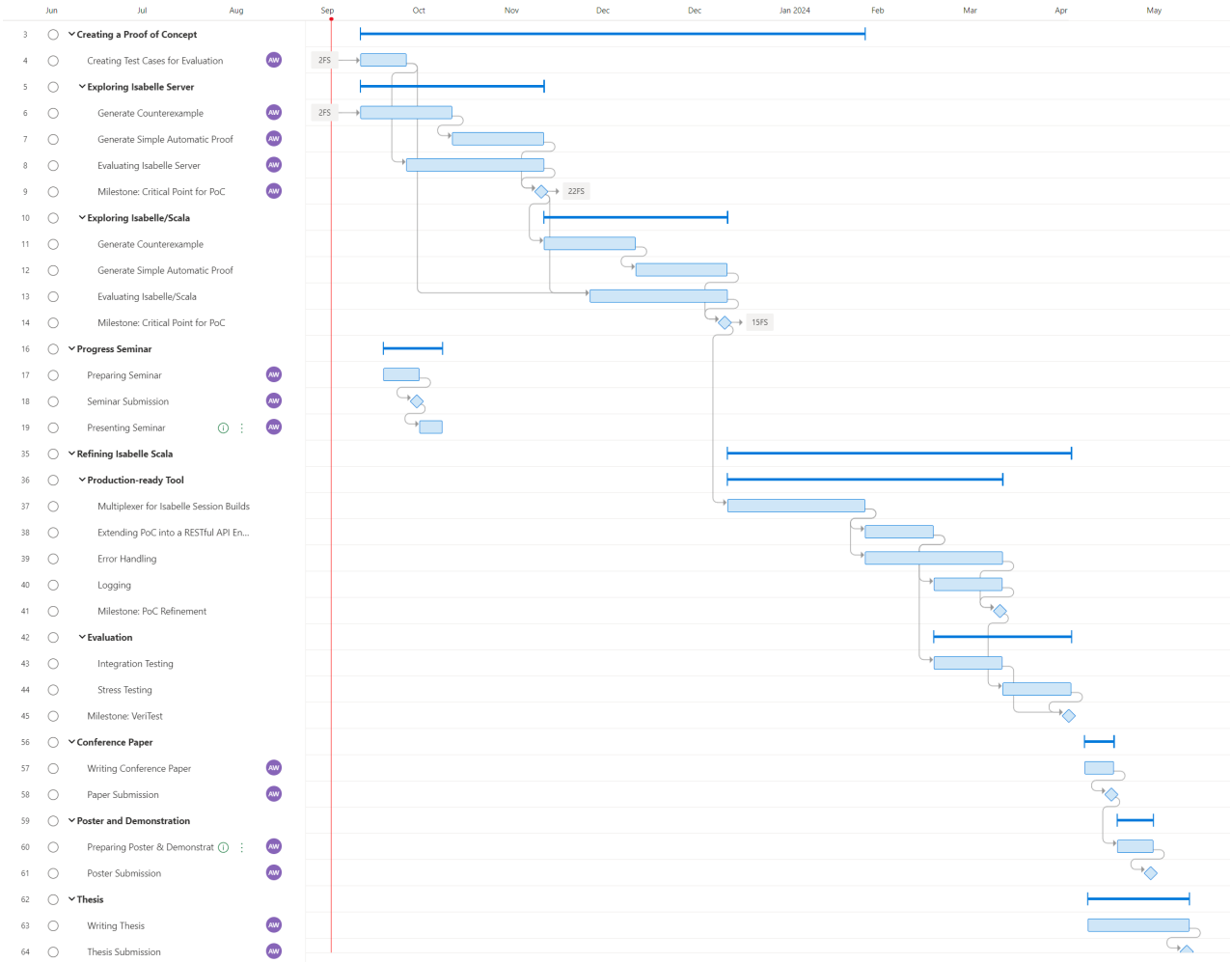

Figure 10: Project timeline for Isabelle/Scala solution (See Sec. 3.3)

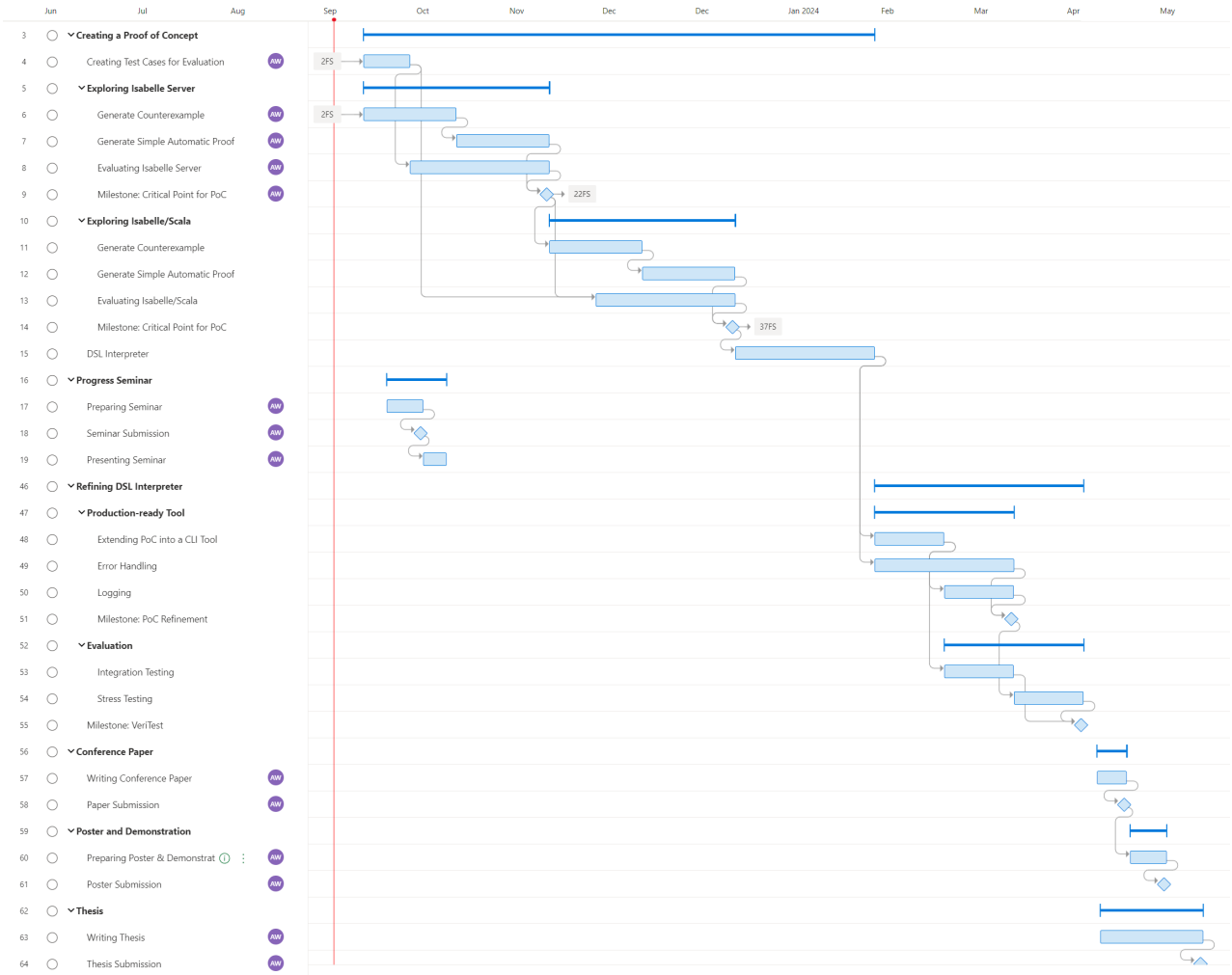

Figure 11: Project timeline for DSL Interpreter solution (See Sec. 3.4)

4.1 Milestones

4.1.1 Milestone 1: Creating a Proof of Concept

To create a PoC for solution Isabelle Server (Sec. 3.2) and Isabelle/Scala (Sec. 3.3), the PoC needs to be able to generate counterexamples and simple automatic proofs for an optimization rule. The generated proof then can be evaluated by its accuracy. Furthermore, the evaluation will include determining the feasibility of the solution.

If both of the solutions are not feasible, then the last resort would be DSL Interpreter (Sec. 3.3). This milestone would represent an extension of Isabelle's [7] command line interface, and would work on the previous knowledge of generating counterexamples and simple automatic proofs from previous solutions. Evaluating this solution's milestone would be trivial, as it would build upon the semi-automatic solutions provided in Veriopt [6, Sec. 5.1].

Depending on the feasibility of each solutions, we would have a *critical point* in the timeline where the project could decide whether to continue with the solution or not. Fig. 4 illustrates a critical point where *if* the Isabelle Server solution is not feasible, the timeline goes through a different critical path that evaluates the feasibility of the Isabelle/Scala solution.

4.1.2 Milestone 2: Refining the Proof of Concept

Refining the PoC would have several phases, which could be done in parallel:

1. Creating a production-ready service;
2. Deployment of the service;
3. Evaluating the service.

The production-ready service will build upon the non-functional requirements of VeriTest (See Fig. 3). The evaluation phase refers to Sec. 3.6 and evaluates whether the service does indeed meet the non-functional requirements.

Each of the consequent solutions after Isabelle Server would have several phases and tasks omitted. This is due to the nature of the solutions itself. Isabelle/Scala solution omits the deployment phase, as it does mean that Isabelle Server cannot be used to offload some of the processing to external Isabelle session build servers. DSL Interpreter solution omits Isabelle session build tasks, as the solution would repeatedly build sessions for each optimization phase. Reaching the DSL Interpreter solution would also mean that it would not be feasible to extend the PoC into a RESTful service – since each Isabelle session build takes a considerable amount of time.

4.2 Risk Assessment

Risk	Consequence	Mitigation
Complexity of utilizing Isabelle is understated	Critical	This proposal has already outlined the feasibility of each solution and should mitigate the risk. Additionally, trimming the scope of the project could be an option, but should be a last resort. Furthermore, implementing consistent intervals of monitoring and controlling phase of the project should mitigate the risk considerably.
The solution does not meet key non-functional requirements	High	Frequent discussions about system specifications and advice on implementation with supervisors.
Problems with personal devices, i.e. short-circuited motherboard	Medium	Frequent backups to the cloud and utilizing UQ-provided labs for development.
Occupational Health and Safety	Low	The majority of the development would be done in a low-risk setting. Additionally, following UQ guidelines for workstation ergonomic assessment should lower the risk.

Table 1: Risk Assessment for VeriTest

4.3 Ethics Assessment

Minimal ethical considerations are present in the project, due to the scope of the project only involving software.

References

[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," en, *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 1, pp. 69–73, Jan. 1999, ISSN: 0163-5948. DOI: 10.1145/308769.308790. [Online]. Available: https://dl.acm.org/doi/10.1145/308769.308790 (visited on 08/19/2023).

[3] X. Leroy, "Formal verification of a realistic compiler," en, *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1538788.1538814. [Online]. Available: https://dl.acm.org/doi/10.1145/1538788.1538814 (visited on 08/19/2023).

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

[5] 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. Lecture Notes in Computer Science, vol. 12971, Cham: Springer International Publishing, Oct. 2021, pp. 111–126, ISBN: 978-3-030-88885-5. DOI: 10.1007/978-3-030-88885-5_8.

[6] 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*, ser. CPP 2023, Boston, MA, USA: Association for Computing Machinery, 2023, pp. 320–333, ISBN: 9798400700262. DOI: 10.1145/3573105.3575673. [Online]. Available: https://doi.org/10.1145/3573105.3575673.

[7] 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.

[8] 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., Cham: Springer International Publishing, 2019, pp. 445–455, ISBN: 978-3-030-25543-5. DOI: 10.1007/978-3-030-25543-5_25.

[9] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: Bounded translation validation for LLVM," en, 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. [Online]. Available: https://dl.acm.org/doi/10.1145/3453483.3454030 (visited on 08/19/2023).

[10] M. Wenzel, "The Isabelle System Manual," en, [Online]. Available: https://isabelle.in.tum.de/dist/Isabelle2022/doc/system.pdf (visited on 09/13/2023).

[11] 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.

[12] J. C. Blanchette, L. Bulwahn, and T. Nipkow, "Automatic Proof and Disproof in Isabelle/HOL," en, in *Frontiers of Combining Systems*, C. Tinelli and V. Sofronie-Stokkermans, Eds., vol. 6989, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 12–27, ISBN: 978-3-642-24363-9 978-3-642-24364-6. DOI: 10.1007/978-3-642-24364-6_2. [Online]. Available: http://link.springer.com/10.1007/978-3-642-24364-6_2 (visited on 08/19/2023).

[13] 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).

[14] J. Blanchette, M. Desharnais, and L. C. Paulson, "A Users Guide to Sledgehammer for Isabelle/HOL," en, [Online]. Available: https://mirror.cse.unsw.edu.au/pub/isabelle/dist/Isabelle2022/doc/sledgehammer.pdf (visited on 09/13/2023).

[15] W. M. McKeeman, "Differential Testing for Software," en, vol. 10, no. 1, pp. 100–107, 1998.

[16] 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. [Online]. Available: https://dl.acm.org/doi/10.1145/1993498.1993532 (visited on 08/19/2023).

[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. [Online]. Available: http://link.springer.com/10.1007/s10817-013-9278-5 (visited on 08/22/2023).

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