UCLouvain

epl

École polytechnique de Louvain

Secured Rust

Author: **Patriciu Vasile Vlaicu**
Supervisor: **Charles Pecheur**
Readers: **Tom Rousseaux, Kim Mens**
Academic year 2024–2025
Master [120] in Computer Science

Abstract

This thesis addresses the challenge of proving the logical correctness of Rust code through weakest precondition calculus while staying as close as possible to the source code. Unlike existing solutions that often focus on unsafe code or focus on the Mid-Level Intermediate Representation (MIR) generated by the Rust compiler, we propose Secured Rust (Secrust), an extensible tool designed to provide formal verification of safe Rust methods.

Taking advantage of Cargo, the Rust package manager, Secrust is a cargo-based package that generates control flow graphs (CFGs), identifies execution paths, and using weakest-precondition (WP) calculus derives final logical implications for verification. These implications are then checked using Microsoft Research's Z3 SMT solver.

The contributions of this thesis include the design and implementation of Secrust, CFG generation, postcondition substitution mechanisms with WP calculus, and Z3 integration. Secrust's results demonstrate its ability to verify logical correctness while laying the groundwork for further and broader language features.

Secrust is currently implemented to work on a subset of the Rust language, handling loops, conditions and most arithmetical operations.

Acknowledgments

First and foremost, I extend my sincere appreciation to **Charles Pecheur**, whose **Méthodes de conception de programmes** course laid the theoretical groundwork for this thesis.

I am deeply thankful to **Tom Rousseaux**, who guided my thesis with patience and dedication. His ability to refocus my efforts when I veered off track and his readiness to explain theoretical concepts were instrumental in ensuring the smooth progression of this thesis.

I also acknowledge the contributions of **Axel Legay**, who provided invaluable insights during his seminar course. His guidance on how to approach research has significantly contributed to the development of this work.

Finally, I want to thank my **family** and **friends** for their support and encouragement throughout my studies.

Contents

1 Introduction **5**

2 Background Material **6**
2.1 Rust Memory Model and Relevance to Verification . . . 6
2.1.1 Safe and Unsafe Code in Rust . . . 6
2.2 Control Flow Graphs (CFGs) . . . 7
2.2.1 Basic Paths . . . 8
2.3 Program Verification . . . 9
2.3.1 Deriving the Weakest-Precondition . . . 9

3 Related Work **11**
3.1 Existing Solutions . . . 11
3.1.1 Verification through Bounded Model Checking . . . 11
3.1.2 Unsafe Rust Verification Frameworks . . . 11
3.1.3 Annotation-Based Verification Tools . . . 11
3.2 Positioning of Secured Rust . . . 12

4 Running Example **13**
4.1 Rust Method . . . 13
4.2 CFG and Basic Paths Generation . . . 14
4.3 Verifying a Basic Path . . . 15
4.4 Deriving Logical Implications for Verification . . . 16
4.5 Verification Results . . . 17

5 Solution **18**
5.1 Introduction to Cargo and Crates . . . 18
5.1.1 The `secrust` Crate . . . 18
5.1.2 The `Cargo.toml` File . . . 19
5.2 Integration into Rust Projects . . . 19
5.3 Overview of the Secrust Crate . . . 21
5.3.1 High-Level Workflow of the Crate . . . 21

5.3.2 Role of lib.rs 21
5.4 The cfg_builder Module 23
5.4.1 Submodules of cfg_builder 23
5.4.2 Workflow of cfg_builder 23
5.4.3 Key Structures and Methods of the Module 24
5.4.4 Path Decomposition and Basic Paths Extraction 26
5.4.5 Summary 28
5.5 The wp_calculus Module 28
5.5.1 Key Responsibilities 28
5.5.2 Core Methods 28
5.5.3 Visual Representation 29
5.5.4 Summary 31
5.6 The verifier Module 31
5.6.1 The Z3 SMT Solver and Its Rust Crate 31
5.6.2 Why Use Z3 in Rust? 31
5.6.3 The Z3 Rust Crate 32
5.6.4 z3_verifier Submodule 33
5.6.5 z3_parser Submodule 35

6 Validation **38**
6.1 Condition Verification 38
6.1.1 Successful Verification 38
6.1.2 Counterexample Found 40
6.2 Programmatic Condition Validation for Arithmetic Methods 41

7 Use of AI **44**
7.1 AI-Driven Support 44
7.2 Personalized AI 44
7.3 Reflection on AI Usage 45

8 Conclusion **46**
8.1 Key Contributions 46
8.2 Challenges and Lessons Learned 47
8.2.1 Interpreting Rust Source Code 47
8.2.2 Integrating the Z3 Solver 48
8.2.3 Lessons Learned 48
8.3 Final Thoughts 48

9 Future Work **49**
9.1 Extending Rust Syntax Support 49
9.2 Industry Case: MultiversX Smart Contracts 50

Chapter 1

Introduction

Renowned as the most admired programming language since 2016 [1], Rust stands out for being among the top programming languages both in terms of memory safety and performance [2], making it a desirable choice for developing reliable software. The importance of logical correctness becomes evident in real-world applications where software reliability is critical. For instance, medical devices require absolute correctness to ensure patient safety, and decentralized smart contracts need robust verification to prevent financial losses. In these cases the program must also do what it is intended to do; memory safety alone is insufficient.

While Rust ensures memory safety at compile time, it does not ensure logical correctness. The primary goal of this thesis is to logically verify Rust methods using macro-based annotations (preconditions, invariants, and postconditions). The challenge lies in integrating a verification tool with Rust projects while minimizing complexity and overhead. Key issues include tool integration, extensibility, and staying as close as possible to Rust source code.

The main contributions of the project are the parsing of Rust code to generate control flow graphs and their DOT representation [1], identifying basic execution paths, performing postcondition substitution and WP calculus. Finally we verify the resulting implications using Microsoft Reaserch's Z3 SMT solver [3].

The project is open source and published on `https://crates.io/crates/secrust`, where both the package and the repository are accessible.

This paper is structured as follows: following the **Introduction**, Chapter 2 provides the **Background Material** necessary to establish the theoretical framework of this work. Chapter 3 examines **Related Work** and existing solutions followed by the **Positioning of Secured Rust** and the problem it addresses. Chapter 4 introduces a **Running Example** to illustrate our approach.

In Chapter 5, the **Solution** is presented through both conceptual and technical details. Chapter 6 demonstrates the **Validation** of the solution with examples. Chapter 7 explores the **Use of AI in the Project**. The **Conclusion**, offered in Chapter 8, is followed by a discussion of potential improvements and extensions in Chapter 9, **Future Work**. Finally, the document concludes with a list of references in the **Bibliography**.

[1]DOT is a plain text graph description language used by Graphviz software to represent structured information, such as control flow graphs.

Chapter 2

Background Material

2.1 Rust Memory Model and Relevance to Verification

The Rust language distinguishes itself through its ownership model and borrow checker, which enforce memory safety at compile time. Unlike many other programming languages that rely on runtime checks or garbage collection, Rust ensures at compile time:

- Single ownership of variables, guaranteeing clear handling of memory allocation and deallocation.
- Borrowing rules prevent mutable references from coexisting with immutable ones, avoiding data races and undefined behavior.
- References cannot outlive the data they point to, preventing dangling pointers.

2.1.1 Safe and Unsafe Code in Rust

While Rust enforces memory safety in most cases, it provides the `unsafe` keyword to bypass these compile-time checks when performance or flexibility demands it [4]. `unsafe` code allows operations like dereferencing raw pointers, calling external functions, or manipulating memory directly. However, using `unsafe` introduces the risk of memory corruption or undefined behavior, similar to other low-level languages like C or C++.

In this work, we focus exclusively on Rust's *safe* code, which is used in the vast majority of cases. With it, we avoid the need for detailed analysis of low-level memory operations. Instead, we concentrate on verifying that programs satisfy their specifications.

2.2 Control Flow Graphs (CFGs)

Control flow graphs provide a representation of programs, where nodes represent basic blocks of code, and edges represent transitions between these points. CFGs can be used to visualize and analyze execution paths. Each edge in a CFG corresponds to a possible transition dictated by the program's control flow, including branches and loops.

Figure 2.1 illustrates a sample CFG generated by Secrust for a loop-based program.

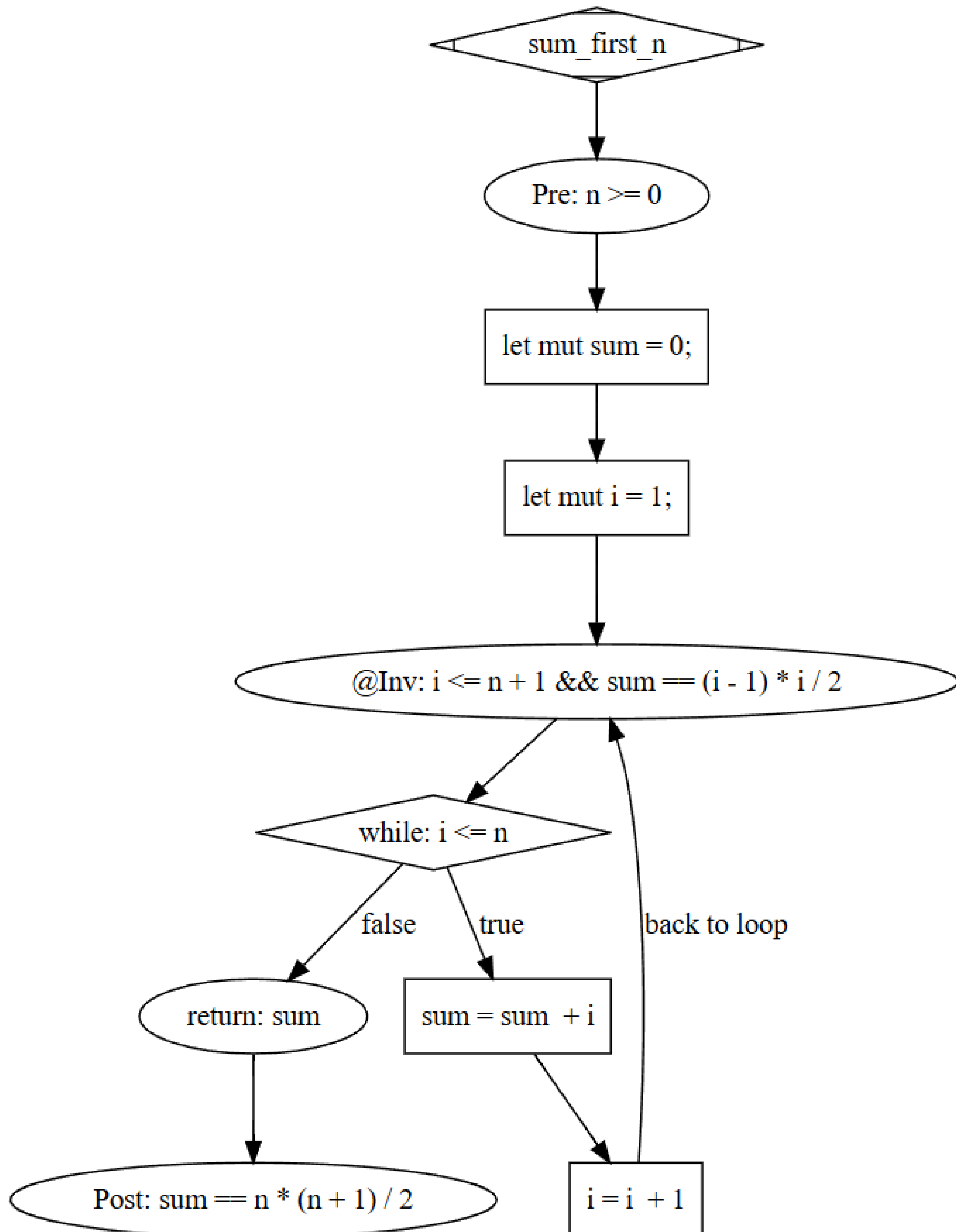


Figure 2.1: CFG generated by Secrust for the `sum_first_n` integers method

2.2.1 Basic Paths

A basic path is a continuous sequence of program instructions executed without any internal branching, starting and ending at control flow cut points. By identifying cut points, we partition the program into these paths, where each path corresponds to a unique traversal of the control flow graph. Each loop contains at least one cut point. A path may contain assumptions about conditions at branching points and includes relevant preconditions, postconditions, or invariants assertions.

Figure 2.2 illustrates a sample basic path leading up to the loop in the Figure 2.1.

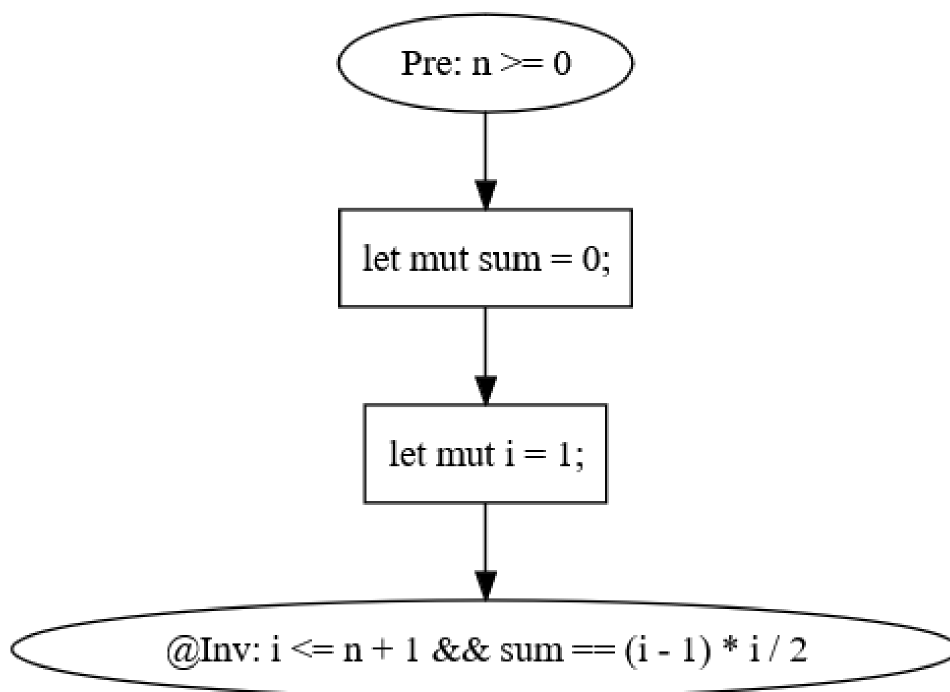


Figure 2.2: Basic path leading up to the loop in `sum_first_n`

2.3 Program Verification

Program verification seeks to formally reason about the correctness of programs by proving that they behave as intended. Central to this are CFGs, basic execution paths, Hoare triples, and weakest precondition calculus.

A Hoare triple, denoted as $[P]S[Q]$, is a logical construct that asserts: if the precondition P holds before executing a statement S, then the postcondition Q will hold after S executes, provided S terminates. It offers us a way to reason about programs at both the statement and block level, ensuring that specific conditions hold throughout the program's execution.

Weakest precondition calculus, is described as follows by Edsger Dijkstra in *A Discipline of Programming* [5, page 16]:

> "The condition that characterizes the set of all initial states such that activation will certainly result in a properly terminating happening leaving the system in a final state satisfying a given post-condition."

In other words, we are looking for the least restrictive condition necessary to ensure that our desired outcome Q is satisfied after executing the program.

2.3.1 Deriving the Weakest-Precondition

Going back to the Hoare triple [P]S[Q], the WP is the weakest precondition of S for Q, also noted wp(S, Q). Q is true after S only and only if wp(S, Q) is true before S. Or in other words, [P]S[Q] only and only if P $\implies wp(S,Q)$. To derive the WP we have to follow a number of instructions, and apply them on the postcondition by moving backward, namely following the basic path from the bottom up:

Assignment instruction:

- Backward rule: $[Q[V := E]]V := E; [Q]$
- WP calculus: $wp(V := E;, Q) = Q[V := E]$

We replace occurrences of V in the postcondition Q with the value E we encounter in the code.

Assume instruction:

- Backward rule: $[C \implies Q]$ **assume** $C; [Q]$
- WP calculus: $wp(\text{assume } C;, Q) = C \implies Q$

We assume C is true, we ignore execution where C is false.

Assert instruction:

- Backward rule: $[C \wedge Q]$ **assert** $C; [Q]$
- WP calculus: $wp(\text{assert } C;, Q) = C \wedge Q$

We assert C as true, C must be provable before proceeding.

Sequential Composition:

- Backward rule: $if [P]S1[R]$ and $[R]S2[Q]$ $then$ $[P]S1S2[Q]$
- WP calculus: $wp(S1S2, Q) = wp(S1, wp(S2, Q))$

Inductive Assertions

To verify loops we use the **Inductive Assertion Method**:

1. **Cut Points:** Identify critical points $@L_i$ in the program's control flow, such as loop entries and exits.
2. **Assertions:** Assign logical assertion P_i, which are preconditions, invariants, and postconditions, to each cut point $@L_i$.
3. **Path Verification:** For each basic path between two cut points, prove the Hoare triple $[P_i]S[P_j]$.

By applying the inductive assertions method, we can prove that if $[P_i]S_{ij}[P_j]$ holds for every basic path between the cut points $@L_i$ and $@L_j$, then $[Pre]S[\text{Post}]$ holds for the program as a whole.

Chapter 3

Related Work

Driven by its memory safety guarantees, formal verification of Rust programs has been tackled from diverse perspectives. Existing tools and methodologies typically concentrate on ensuring memory safety for `unsafe` Rust code, with some prioritizing bug prevention, others emphasizing functional correctness, and a few attempting to address most of these aspects simultaneously.

3.1 Existing Solutions

3.1.1 Verification through Bounded Model Checking

CBMC-Based Tools. Tools like **Kani** [6] and **Crust** [7] use bounded model checking to explore execution paths systematically. Kani translates Rust's Mid-level Intermediate Representation (MIR) — a control-flow-centric simplified form of Rust code — into Goto-C for CBMC, while Crust translates Rust code directly into C. These tools identify memory safety violations and verify functional correctness, limited within their bounded domains.

3.1.2 Unsafe Rust Verification Frameworks

RustBelt [8] is a tool built using Iris [9], a framework for higher-order concurrent separation logic, and the formal proof management system Rocq Prover [10] (previously known as the Coq proof Asistant). It focuses its efforts on λRust — a simplified subset of the Rust language — to address challenges such as ownership discipline, thread-safe type bounds, and borrowing semantics and also to verify that unsafe operations are safely encapsulated in otherwise safe code.

MirChecker [11] operates on Rust's MIR, performing abstract interpretation and static analysis to identify runtime bugs and potential memory issues. It complements Rust's compile-time checks by detecting runtime bugs the compiler cannot catch such as integer overflows, out-of-bounds array accesses, and division by zero. Since rust is a predominantly safe language, most of the bugs detected through MirChecker are not memory-safety bugs but runtime panics that abort code execution.

3.1.3 Annotation-Based Verification Tools

Prusti [12] leverages separation logic behind the scenes to verify Rust programs and reason about memory allocation, access, and modification. It focuses on panic freedom and functional correctness. Developers can use incremental annotations to specify preconditions, postconditions, and loop invariants. It heavily relies on the MIR representation

and uses Viper [13] as its backend verification engine.

Creusot [14] is a deductive verification tool that translates the MIR of Rust programs annotated with the PEARLITE specification language into WhyML, the input language of the Why3 [15] tool for deductive program verification, to derive verification conditions.

As we can see, current tools, such as Prusti and Creusot, take advantage of MIR's focus on control flow to avoid building CFGs from scratch and perform their analysis on Rust's MIR. MIR provides advantages by stripping away high-level constructs and abstractions present in source code, but it is done at the risk of making verification less intuitive for developers.

3.2 Positioning of Secured Rust

One of the key constraints of this thesis is to operate directly on Rust source code rather than intermediate representations like MIR. By focusing on Rust's memory safety guarantees, Secrust focuses exclusively on verifying logical correctness with a verification process that is tightly integrated with the source code as we derive the final logical implications directly from the written code and conditions.

As such, Secured Rust offers 3 very clear advantages:

- **CFGs of Source Code:** Unlike tools that rely on intermediate representations such as MIR, Secured Rust constructs CFGs directly from the source code. Developers can also visualize the CFGs and basic paths generated in the `DOT` graph description language, making it easier to understand and interpret verification results.
- **Focus on Logical Correctness:** Rust's memory safety features guarantee the absence of data races and other common memory-related bugs in `safe` Rust code. Secured Rust capitalizes on this by entirely focusing on logical correctness.
- **Developer-Friendly Annotations and Verification:** Secured Rust allows developers to annotate their code with `pre!`, `post!`, and `invariant!` macros to specify conditions directly in the source code. These annotations are used to derive the weakest precondition implications directly from user input, and developers can check the updated post conditions and resulting implications in the terminal to closely monitor the verification process.
- **Ease of Integration:** Secured Rust is designed to integrate seamlessly into existing Rust projects by taking advantage of Rust's package manager, `Cargo`. As a Rust crate (package) `Secrust` is easily installed using cargo, with developers only needing to install the Z3 SMT solver on their system. Once installed, developers can incorporate `secrust` into their projects, annotate their methods, and perform formal verification. Chapter 5 provides a more in-depth discussion of Secrust integration.

Chapter 4

Running Example

To illustrate how Secrust works, we provide a walkthrough of the verification process applied to a simple Rust function.

We start with an annotated Rust code inside a rust file and explain the high-level steps that Secrust performs. We will analyze the following method used to calculate the sum of the first n integers.

4.1 Rust Method

```
use secrust::{pre, post, invariant};

fn sum_first_n(n: i32) -> i32{
    pre!(n >= 0);
    let mut sum = 0;
    let mut i = 1;
    invariant!(i <= n + 1 && sum == (i - 1) * i / 2);
    while i <= n {
        sum = sum + i;
        i = i + 1;
    }
    post!(sum == n * (n + 1) / 2);
    return sum;
}

fn main() {
    let n = 0;
    let sum = sum_first_n(n);
    print!("Sum is: {} \n", sum);
}
```

Code Listing 4.1: Example of annotated Rust code using Secrust

4.2 CFG and Basic Paths Generation

The first step in the verification process involves generating the control flow graph (CFG) and extracting basic paths.

Secrust automatically creates cut points in the CFG where there are annotations or conditions. It will then generate basic paths from the main CFG generated.

When Secrust is run with the `--dot` flag, the generated CFG and its basic paths are outputted as DOT files inside a `graphs` directory. Figure 4.1 shows the overall CFG, annotated to highlight the execution paths and cut points. Each distinct path through the CFG is extracted, as demonstrated in Figure 2.2.

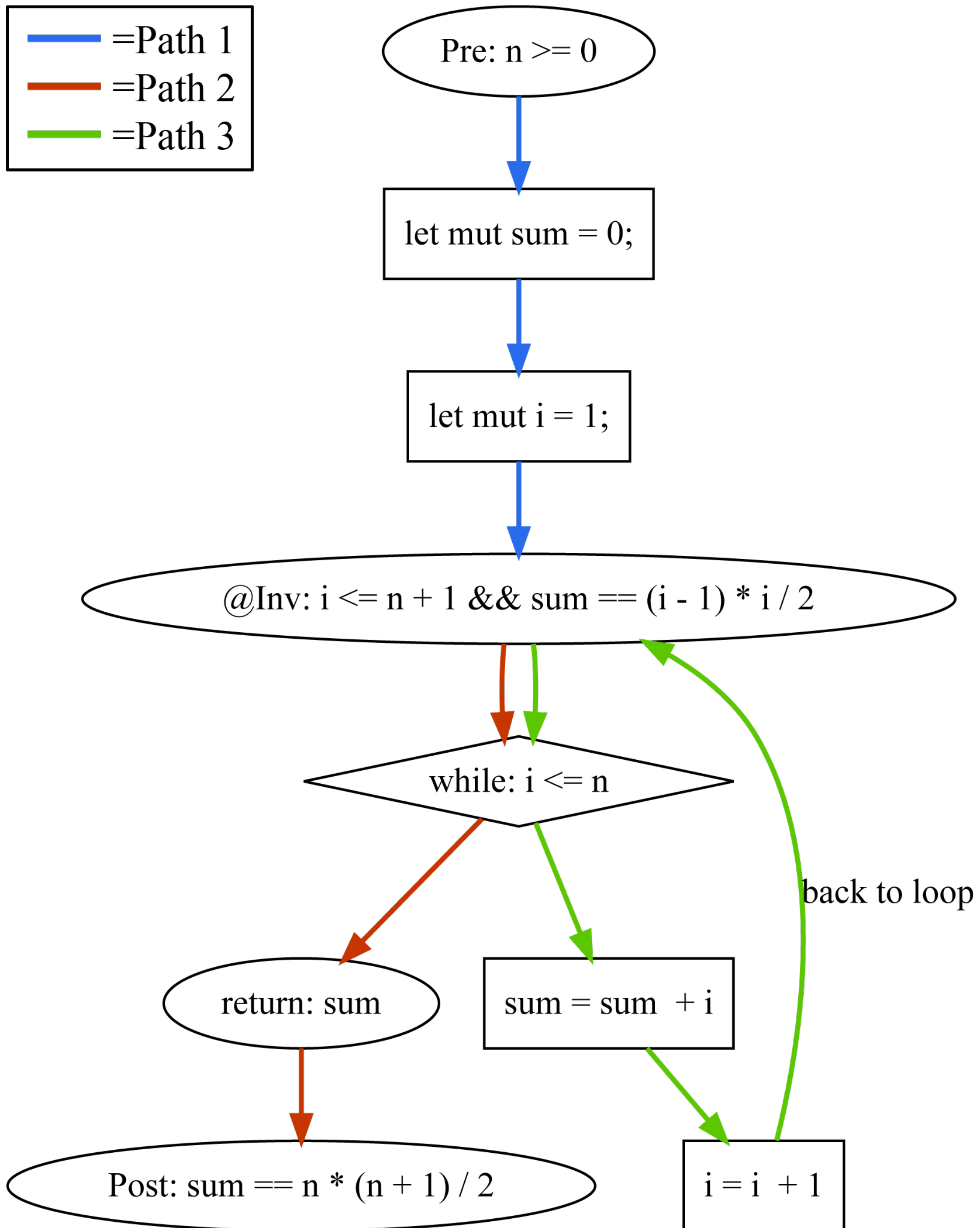


Figure 4.1: A manually edited version of the CFG shown in Figure 2.1 to highlight all basic paths extracted by Secrust.

4.3 Verifying a Basic Path

Once the CFG is generated and all basic paths are identified, Secrust will process each basic path and derive a logical implication using weakest precondition (WP) calculus described in chapter 2. The implication represents the logical conditions that must hold for the path to be verified.

To illustrate, we focus on the 3rd basic path (highlighted in green on Figure 4.1).

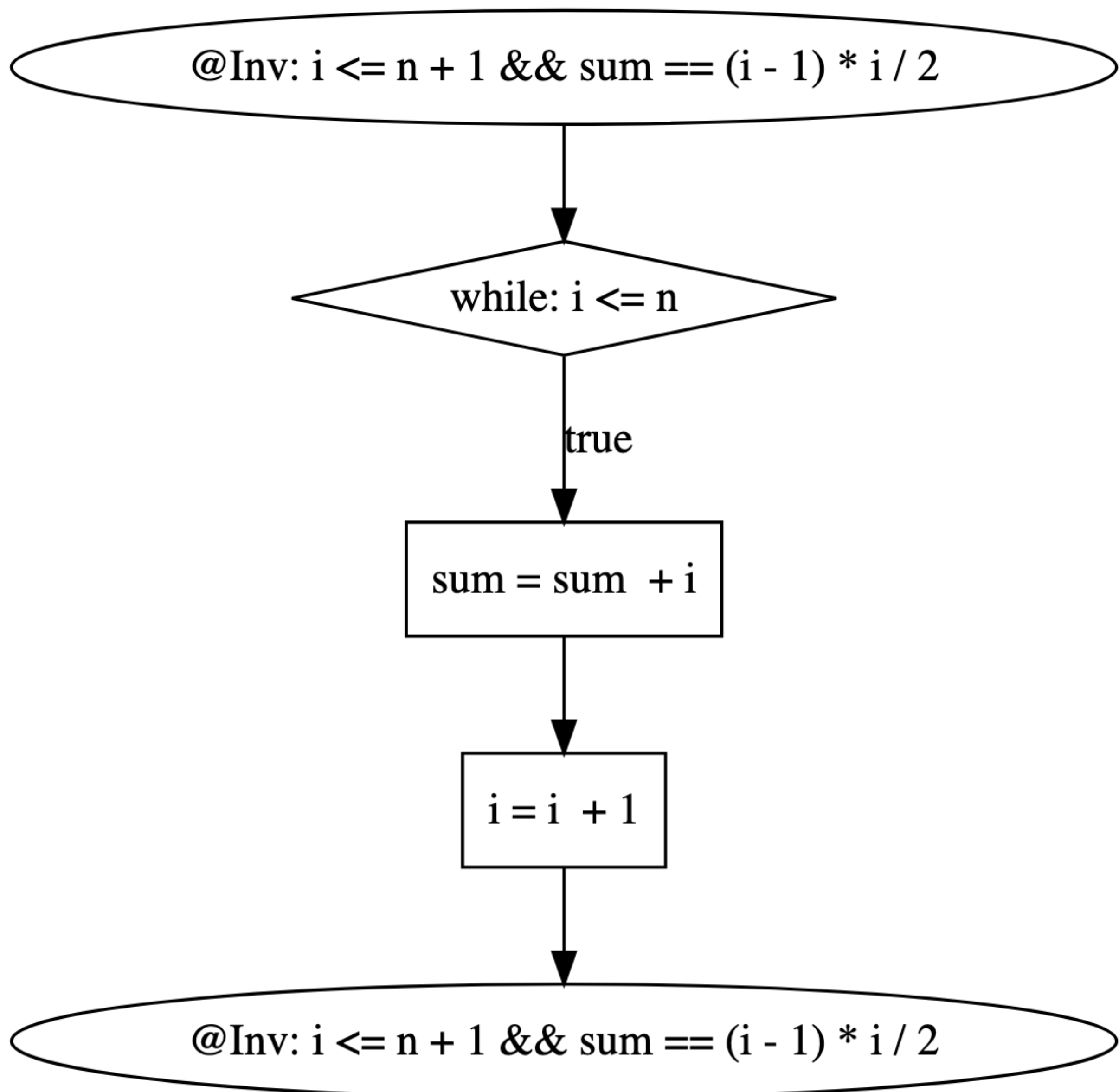


Figure 4.2: Visualization of the DOT representation for basic path 3 generated by Secrust.

4.4 Deriving Logical Implications for Verification

To derive the final logical implication, Secrust traverses the basic path in a backward direction, starting from the bottom node, which represents the postcondition. As it moves upward through the graph, the postcondition is iteratively modified by substituting variables updated at each step. Additionally, conditional assumptions and assertions are chained together in the order they appear within the path.

In our example, we take the bottom invariant

$$i \leq n + 1 \quad \&\& \quad \text{sum} == (i - 1) * i/2$$

and start going up the graph. We will look step by step at the nodes and how they modify the condition, starting with the first two nodes.

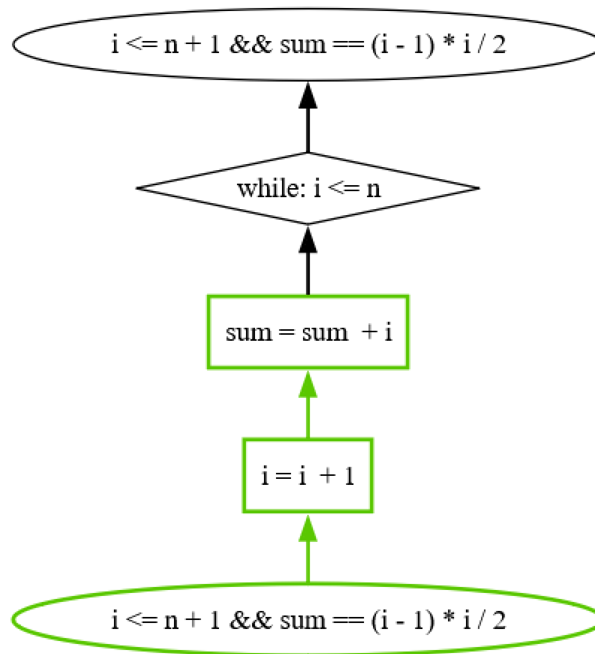


Figure 4.3: Bottom invariant and first two instruction nodes of Figure 4.2 in backward direction

To simplify the explanation, we focus on groups of nodes that require the same instructions, starting with the bottom invariant and the following two assignment nodes in this example, as shown in Figure 4.3.

As we can see, the nodes contain assignments of the `i` and `sum` variables, modifying them to `i+1` and `sum+i`, respectively. Secrust checks the bottom invariant to see if it contains these variables and substitutes them with their new values.

Our bottom invariant becomes:

$$(i + 1) \leq n + 1 \quad \&\& \quad (\text{sum} + i) == ((i + 1) - 1) * (i + 1)/2$$

We can now focus on the updated bottom invariant and the following while condition.

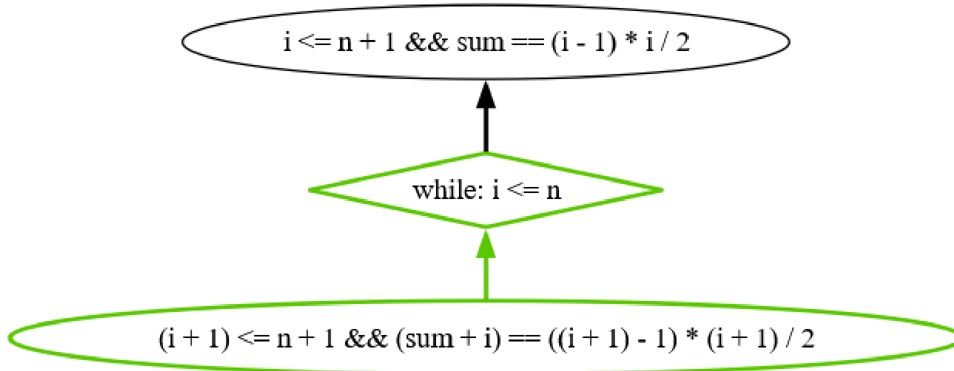


Figure 4.4: Updated bottom invariant and while condition from Figure 4.2 in backward direction

As the path 3 we are currently analyzing is the path resulting from the `true` branch of the while condition, we have to assume the condition is `true`. Thus, the condition $i \leq n$ now implies the updated bottom invariant:

$$i \leq n \Rightarrow ((i+1) \leq n+1 \quad \&\& \quad (\text{sum}+i) == ((i+1)-1)*(i+1)/2)$$

Now only the starting invariant remains in the path:

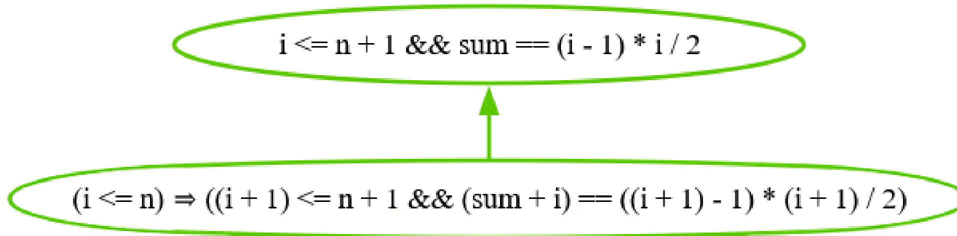


Figure 4.5: Updated bottom condition and starting invariant of Figure 4.2 in backward direction

The invariant is the precondition assertion of this basic path. By adding the precondition, we obtain this final implication:

$$(i \leq n+1 \quad \&\& \quad \text{sum} == (i-1)*i/2) \Rightarrow$$
$$(i \leq n) \Rightarrow$$
$$((i+1) \leq n+1 \quad \&\& \quad (\text{sum}+i) == ((i+1)-1)*(i+1)/2)$$

4.5 Verification Results

Once the final implication is derived from the basic path, it is translated into a Z3 Abstract Syntax Tree (AST) using the Z3 Rust crate. The crate acts as an interface to the Z3 solver installed on the system, handling the translation of the AST into the SMT-LIB format —a standardized language for expressing logical formulas used by satisfiability solvers [16]— for verification, as shown in Code Listing 4.2. Using Z3, the negated form of the implication is checked for satisfiability. In our case, since the negated implication is unsatisfiable, the condition is verified to hold.

```
(=> (and (<= i (+ n 1)) (= sum (div (* (- i 1) i) 2)))
    (=> (<= i n)
        (and (<= (+ i 1) (+ n 1))
             (= (+ sum i) (div (* (- (+ i 1) 1) (+ i 1)) 2)))))
```

Code Listing 4.2: SMT-LIB format of the final implication

Further details about the construction of implications and their translation into Z3 format can be found in Chapter 5, where the implementation details are discussed in depth.

Chapter 5

Solution

Our solution takes the form of a Rust crate, which can be installed through the Rust package manager, Cargo. Unlike approaches that necessitate cloning a repository, Secrust leverages the Rust toolchain for installation, requiring users only to have the Z3 solver installed on their system. The crate is available at https://crates.io/crates/secrust.

To better understand this approach, we will briefly explain how Cargo and crates work before detailing how Secrust works and can be set up in a Rust project.

5.1 Introduction to Cargo and Crates

Cargo is the Rust package manager and build system. It is a command-line tool that automates many tasks in Rust development, including dependency management, building projects, running tests, and creating executables or libraries. At it's heart, Cargo handles packages called **crates**, a unit of compilation in Rust which can either be compiled into a library or an executable binary.

Crates are published on the official Rust package registry, `crates.io` and Cargo facilitates managing dependencies specified in a project's `Cargo.toml` file, which describes the project's configuration, dependencies, and metadata.

5.1.1 The `secrust` Crate

The `secrust` project is structured as a Rust crate. It provides both a library and a binary executable. The library defines annotations like `pre!`, `post!`, and `invariant!`, which are used to add verification conditions to Rust code. The binary, accessible via the custom command `cargo secrust-verify`, serves as a command-line interface (CLI) tool that verifies annotated Rust code and generates DOT format CFGs of the code.

Binary Integration via Cargo: The crate's `Cargo.toml` file specifies a binary target. This configuration makes it possible for users to execute the `cargo secrust-verify` command after installing the crate.

5.1.2 The Cargo.toml File

Below is an annotated explanation of the Cargo.toml file for secrust:

```
[package]
name = "secrust"                   # The name of the crate
version = "0.1.0-alpha.3"          # The version of the crate
edition = "2021"                   # Rust edition to use
...
[dependencies] # crates used in Secrust
clap = { version = "4", features = ["derive"] } # Parse args
petgraph = "0.6"                   # Library used to build graph
    structures
proc-macro2 = "1.0"                # Macro system for Rust
syn = { version = "1.0", features = ["full", "visit", "extra-
    traits"] }                     # Parser for Rust code
quote = "1.0"                      # Used to handle macro args
serde = { version = "1.0", features = ["derive"] }  # For
    serialization/deserialization
serde_json = "1.0"                 # JSON support
regex = "1.5"                      # Regular expressions
z3 = "0.12.1"                      # Z3 SMT solver integration

[lib]
path = "src/lib.rs"                # Specifies main library file

[[bin]]
name = "cargo-secrust-verify"       # Command to run the binary
path = "src/main.rs"                # Main file for the binary
```

Code Listing 5.1: The Cargo.toml file for secrust.

Cargo.toml Sections:

- [package]: Metadata about the crate, such as its name, version, and Rust edition.
- [dependencies]: Lists external libraries (crates) required by secrust.
- [lib]: Defines the library entry point (src/lib.rs).
- [[bin]]: Configures the binary executable. The binary is named cargo-secrust-verify, which allows it to be called as a subcommand of Cargo.

5.2 Integration into Rust Projects

To integrate secrust into a Rust project, follow these steps:

1. **Install Z3:** Ensure that the Z3 solver is installed and available on your system.
2. **C Compiler:** Ensure you have a C compiler installed on your machine, as Cargo requires it to build the z3-sys dependency.

3. **Install Secrust:** Use Cargo, the Rust package manager, to install the `secrust` crate: [1]

```
1     cargo install secrust --version 0.1.0-alpha.3
```

4. **Annotate Rust Functions:** Add annotations such as `pre!`, `post!`, and `invariant!` to your functions to define preconditions, postconditions, and loop invariants. Refer to the example code in Code Listing 4.1.

5. **Run Verification:** Execute the `secrust` verification process using the following command:

```
1     cargo secrust-verify src/main.rs --dot
```

The `--dot` flag generates Control Flow Graph DOT files for visualization. These files are saved in the `src/graphs` directory, organized by filename analyzed.

Figure 5.1 provides a high-level overview of the interactions between the `secrust` crate and your project files.

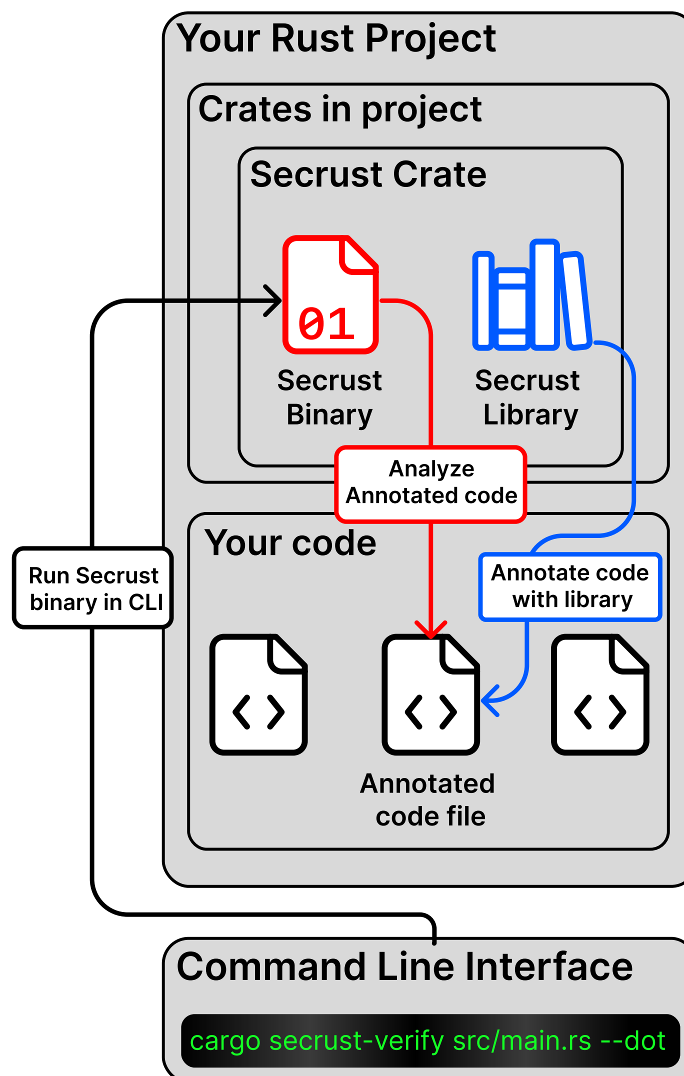


Figure 5.1: Overview of the `secrust` workflow.

[1] Please check the latest version on `https://crates.io/crates/secrust`. You can either add the dependency to your project's `Cargo.toml` using the latest version number or install it with `cargo install secrust --version [latest]`, replacing `[latest]` with the most recent version available. This is recommended until a stable version is officially released.

5.3 Overview of the Secrust Crate

The `Secrust` crate is designed to be modular, with each module focusing on a specific aspect of the verification process. The modular design simplifies maintenance, supports scalability to handle additional Rust syntax, and separates concerns to keep growing parts of the project manageable as complexity increases. Figure 5.2 illustrates the higher-level workflow of these modules.

5.3.1 High-Level Workflow of the Crate

When a user invokes the CLI command `cargo secrust-verify src/main.rs --dot`, the following sequence is triggered:

1. The `main.rs` file in the Secrust binary processes the CLI arguments to determine:
    - The file to analyze.
    - Whether the `--dot` flag is present, which specifies if DOT files should be generated in the `src/graphs` directory.
2. Once the arguments are parsed, `main.rs` invokes the `run_verification` method from `lib.rs`.

5.3.2 Role of `lib.rs`

The `lib.rs` file serves as the core of the crate. It defines:

- The **annotation macros** (`pre!`, `post!`, `invariant!`) used in Rust code. These macros are placeholders that accept any Rust code as arguments and are later identified in the AST during analysis.
- The **`run_verification`** method, which:
    - Parses the file provided in the CLI argument into an AST using the `syn` crate.
    - Calls the `cfg_builder` module to create a `CfgBuilder` structure.
    - Invokes `build_cfg()` to generate the CFG for annotated methods in the analyzed file.
    - Uses `generate_basic_paths()` to decompose each method CFG into basic paths.
    - Applies Weakest Precondition calculus to derive the final implications for each basic path by calling `apply_wp_calculus(&basic_paths)` from the `CfgBuilder` structure, implemented in the `wp_calculus` module.
    - Verifies each derived implication using the `verify_str_implication` method from the `verifier` module.
    - Optionally generates DOT files for the CFG and basic paths if the `--dot` flag is set.

This high-level workflow is illustrated in Figure 5.2. In the following sections we'll explore each module in details.

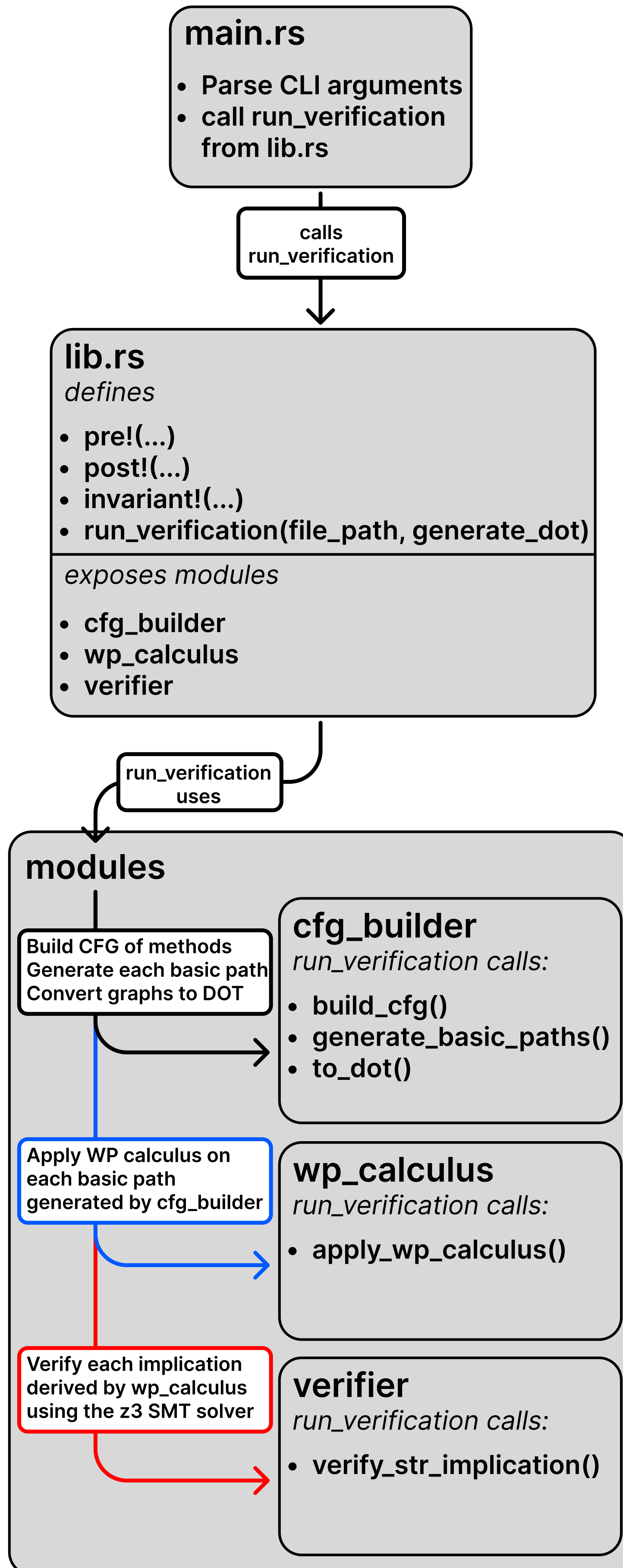


Figure 5.2: High-level workflow of the Secrust crate and its modules.

5.4 The `cfg_builder` Module

The `cfg_builder` module is a crucial part of the `Secrust` crate as constructing a Control Flow Graph (CFG) is essential for our methodology of formal verification using Weakest Precondition calculus on source code. At the time of writing, we found no existing tool to generate a CFG directly from Rust source code as most projects leveraging Rust CFGs rely on the Mid-Intermediate Representation (MIR) produced by the Rust compiler.

Since the use of MIR was not compatible with our goal of staying as close to the original Rust source code as possible, we had to design and develop this functionality from scratch. To achieve this, we interpret the Abstract Syntax Tree (AST) of the Rust code generated by the `syn` crate and construct the CFG step-by-step. The `cfg_builder` currently supports essential Rust control-flow constructs, such as `while` and `for` loops, `if-else` conditions, method calls, and block statements.

The module inspects the functions present in the file to detect any relevant annotation macros (such as `pre!`, `post!`, or `invariant!`). If such annotations are present, the module analyzes the function in detail to generate the corresponding CFG. Additionally, even in the absence of these annotations, the module will generate a CFG for the function if the `build_cfg!()` macro is explicitly included.

The module is composed of several submodules, each responsible for specific tasks in the construction of the CFG.

5.4.1 Submodules of `cfg_builder`

The `cfg_builder` module contains the following submodules:

- **`builder.rs`**: The main file that defines the `CfgBuilder` structure. It coordinates the entire CFG construction process and serves as the entry point for the module.

- **`node.rs`**: Defines the `CfgNode` structure, which represents nodes in the CFG. Each node may represent a function, statement, precondition, postcondition, invariant, or other control-flow elements (like loops and conditions). It also contains the `DOT` format representation of each node.

- **`find_paths.rs`**: Responsible for finding the basic paths in the CFG. It decomposes the CFG into paths that will later be used in the WP calculus process.

- **`handle_condition.rs`**: Handles `if` and `else` conditions in the Rust source code and ensures their appropriate branching representation in the CFG.

- **`handle_loops.rs`**: Processes loop structures (such as `while` and `for` loops) and represents them in the CFG.

- **`handle_macros.rs`**: Manages the handling of Rust macros.

- **`handle_call.rs`**: Handles Rust method calls with possible custom assertions in a config file.

- **`handle_return.rs`**: Handles `return` statements in the Rust source code and ensures that they are properly represented in the CFG.

5.4.2 Workflow of `cfg_builder`

The high-level workflow of the `cfg_builder` is as follows:

1. **[lib.rs]**

- Call `CfgBuilder::new()` from `builder.rs` to create a new CfgBuilder struct.
- Parse the source file into an AST using the `syn` crate.
- Call `builder.build_cfg(ast)` to start building the CFG.

2. [**builder.rs**]

   - **Visit File:** The `builder.build_cfg()` method will now call the `visit_file` method to processes the AST, identifying Rust functions and statements.
   - **Node Creation:** Nodes and edges are created accordingly for each relevant element in the AST, such as functions, conditions, loops, and annotations. This is done using submodules like `handle_condition.rs` and `handle_loops.rs`.
   - **Post-processing:** The graph is post-processed to retain only the essential nodes and ensure that postconditions are added at the end of the graph. During this step, auxiliary nodes such as `MergePoint` nodes, which are used as intermediate helpers, are removed to simplify the graph structure.

3. [**lib.rs**]

   - **Path Decomposition:** Call `generate_basic_paths` from `find_paths` submodule of `cfg_builder` to decompose the CFG into basic paths.

5.4.3 Key Structures and Methods of the Module

1. `CfgBuilder`

The main structure of the `cfg_builder` module is `CfgBuilder` and is defined in the `builder` submodule. This structure contains key fields such as:

- `graph`: A directed graph representing the control flow of a Rust method. It is implemented using the `DiGraph` structure for directed graphs from the `petgraph` crate.
- `current_node`: Tracks the current node being processed in the CFG.

2. `CfgNode` The `CfgNode` structure defines the types of nodes that can exist in the CFG. Some examples of node types are:

- `Function`: Represents the start of a function.
- `Precondition`, `Postcondition`, `Invariant`: Represent assertions in the code.
- `Statement`: Represents regular statements in the Rust source code.
- `Condition`: Represents `if`, `else`, and also loop conditions.
- `Return`: Represents return statements in Rust functions.
- `MergePoint`: Represents the point where multiple execution paths converge, often at the end of `if-else` branches or loop exits.
- `Cutoff`: Marks the logical "jump-back" point for loops when no explicit invariant is defined. It helps ensure proper CFG connections during loops.

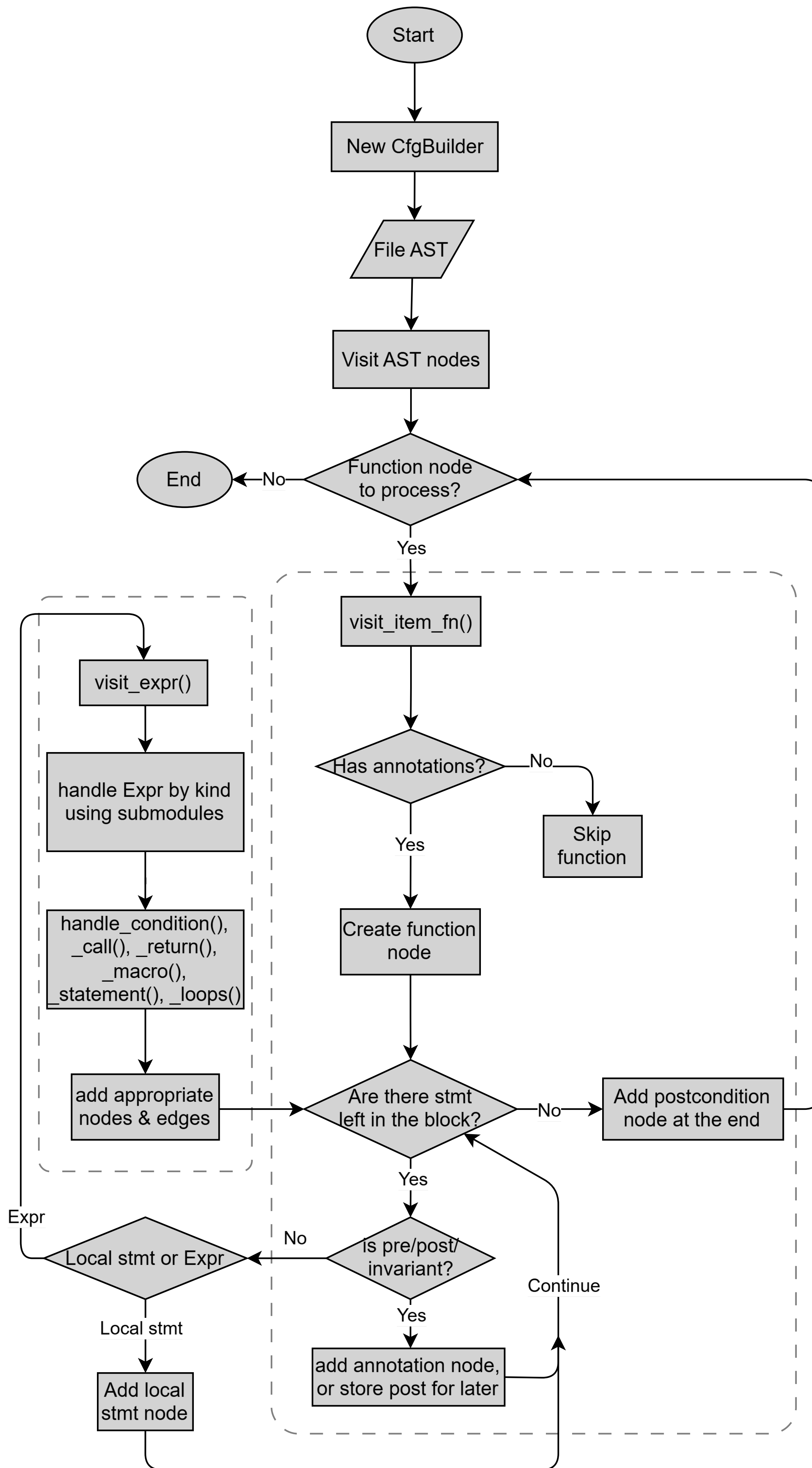


Figure 5.3: High-level workflow of how the code builds CFGs with the `builder` submodule with detailed `visit_expr()` and `visit_item_fn()` methods. Statements are called "stmt".

Figure 5.3 presents a high-level workflow of how the `cfg_builder` module builds CFGs for Rust methods. While this figure provides a conceptual understanding of the module's execution flow, it abstracts away certain technical details.

Specifically, the module leverages the `syn` crate's `Visit` trait, which statically determines and calls appropriate methods based on the type of node being visited in the Rust code's AST. For example, when visiting statements, `visit_stmt` is called, and when visiting expressions, `visit_expr` is invoked. The diagram focuses on showing the primary decision points and the overall flow of the traversal process.

5.4.4 Path Decomposition and Basic Paths Extraction

Once the CFG is constructed for a Rust method, the next critical step in the verification process is to extract basic paths since each independent execution path must be analyzed separately.

Responsibilities of the `find_paths.rs` submodule

The `find_paths.rs` file defines methods that traverse the CFG and extract all its basic paths. The core methods are as follows:

- **`generate_basic_paths()`**: The main method behind the generation of basic paths from the CFG. It identifies key nodes in the graph, such as Preconditions, Postconditions, Invariants, and Cutoff Nodes, and initiates the path extraction process for each of them.
- **`find_paths()`**: A recursive depth-first search method that traverses the CFG to construct basic paths. It adds each visited node to the current path and terminates when it reaches a terminal node, such as a Postcondition, Return, or Cutoff.
- **`process_loop_invariant_path()`**: Handles paths that loop back to an invariant node. If a loop is detected, the method duplicates the invariant node, breaks the loop and appends it to the end of the path.
- **`write_paths_to_dot_files()`**: Saves each generated basic path as a DOT file. These files, stored in the `src/graphs` directory.

The high-level workflow of the `find_paths` submodule is presented in Figure 5.4 on the following page. It provides an overview of how paths are decomposed from key points in the CFG.

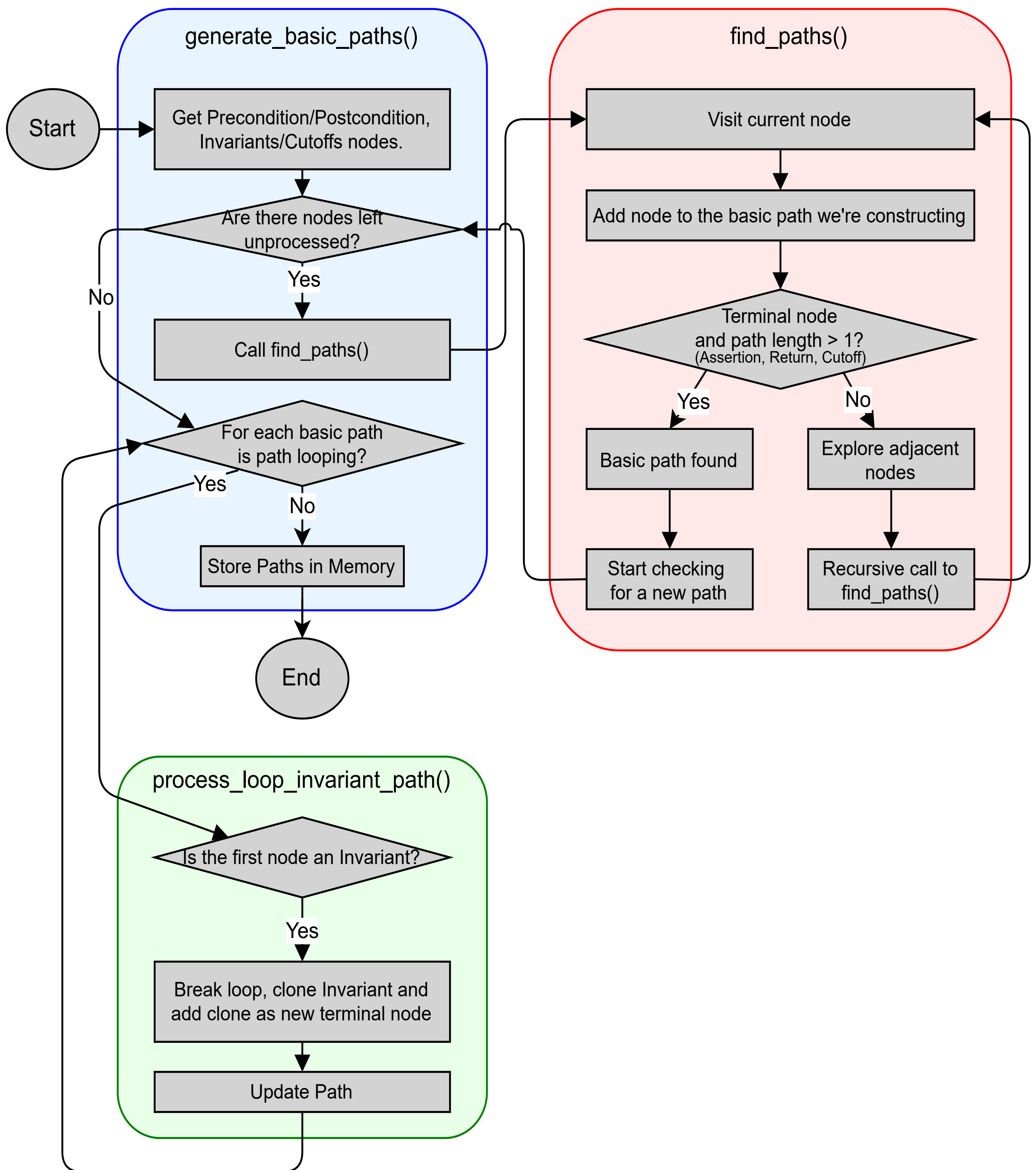


Figure 5.4: High-level workflow of the `find_paths` submodule.

5.4.5 Summary

The `cfg_builder` module is responsible for constructing the Control Flow Graph of annotated Rust methods in a file and extracting their basic paths. Extending support for new Rust syntax is straightforward, requiring updates to the `Visit` trait of the `syn` crate nodes and the addition of corresponding submodules in `builder.rs`.

5.5 The `wp_calculus` Module

The `wp_calculus` module is used to derive the weakest precondition (WP) from each basic path. This process involves starting from the end of each path (where conditions such as postconditions and invariants are located) and working backward toward the precondition, applying the WP calculus rules explained in Section 2.3.1. The goal is to produce a single logical formula for each path which we will later verify.

5.5.1 Key Responsibilities

The module's primary responsibilities are:

- Traverse basic paths backward, substituting variables in postcondition or invariant nodes at the end of each path based on their state at each node using `recursive_substitution()`.
- Chain conditions using logical implication, represented by the » (right shift) operator[2]. This operator is used to produce a unified condition for each path.
- Ensure proper operator precedence by adding parentheses around substituted values.

5.5.2 Core Methods

`apply_wp_calculus()` Is the main method of the module. It performs the following steps:

- **Initialization:** Initialize a `variable_state` map to track updates to variable values and a `working_condition` variable to hold the terminal postcondition or invariant to be processed.
- **Reverse Path Traversal:** Traverse each basic path in reverse, beginning from the postcondition and progressing toward the precondition.

When traversing a basic path we process nodes in the following way:

- **Postcondition/Invariant Node:** This is the first type of node encountered during reverse traversal. We set `working_condition` to this node, as it is the condition we are working on.
- **Assignment Node:**
  - Parse the assignment statement using `parse_assignment()` to extract the variable and its assigned value.
  - Update variables in the `working_condition` using `recursive_substitution()`.
  - Update the `variable_state` map with the new values for future substitutions.

[2] We currently use the » (right shift) operator to represent logical implications, as we rely on Rust tokens in our logical expressions to analyze code using the AST provided by the `syn` crate.

- **Condition Node:**
  - Check whether the condition is part of a false branch using `is_false_branch()`.
  - Negate the condition if it belongs to a false branch.
  - Chain the condition into the `working_condition` using logical implication (»).
- **Precondition Node:** Chain the precondition as the final implication in the logical formula.
- **Output Generation:** Append the resulting logical condition to the list of updated postconditions for each basic path.

`recursive_substitution()` This method is used to recursively substitute variables within our working condition:

- **Base Case:** If the expression is a variable having the same name as the updated variable, replace it with the new value.
- **Recursive Case:** For complex AST expressions (such as `Assign`, `Binary`, `Call`, `If`, `Macro`...), apply substitutions to their subparts such as `left`, `right`, `args`, and conditions.
- **No Match:** Return the expression unchanged.

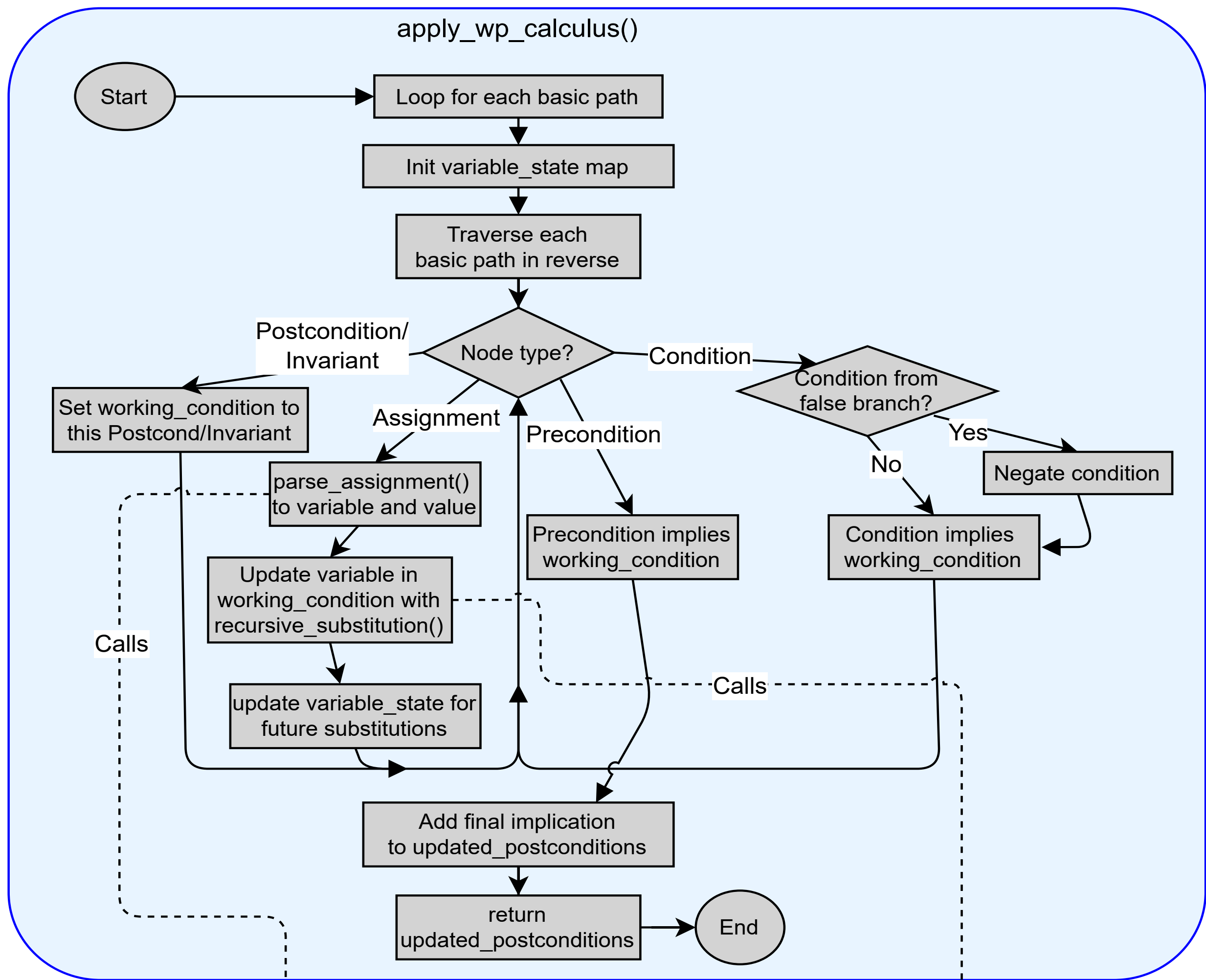
`parse_assignment()` Parses an assignment statement and extracts the variable name and its assigned value:

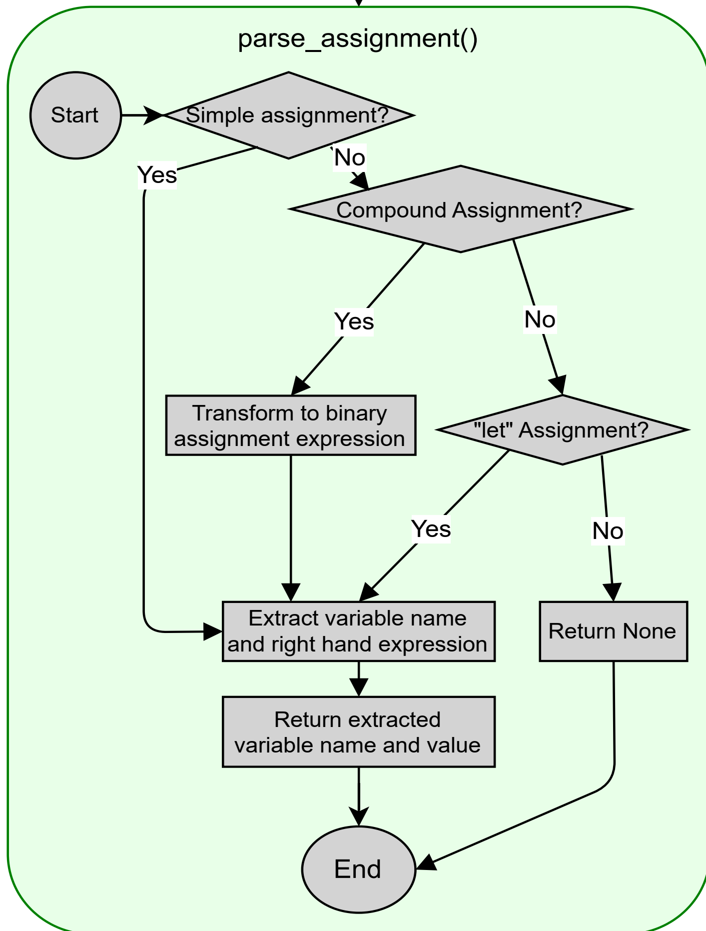
- **Simple Assignment:** such as `x = 5;`
- **Compound Assignment:** such as `x += 1;` (transformed to a binary expression).
- **Let Statement:** such as `let mut sum = 0;`.

If no valid assignment is found, it returns `None`.

5.5.3 Visual Representation

Figure 5.5 illustrates the high-level workflow of the `apply_wp_calculus` method, as well as the supporting methods `parse_assignment()` and `recursive_substitution()`.





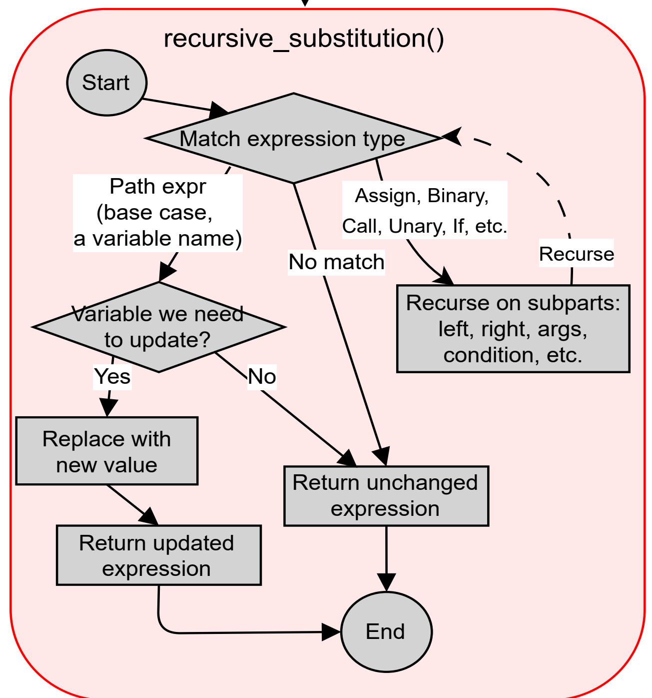


Figure 5.5: High-level workflow of the core methods of the `wp_calculus` module.

5.5.4 Summary

The `wp_calculus` module applies the Weakest Precondition calculus to generate logical conditions for each basic path in the CFG. It combines:

- Reverse path traversal,
- Recursive variable substitution,
- Logical condition chaining.

These logical conditions are used in the `verifier` module to verify the correctness of the annotated Rust code.

5.6 The `verifier` Module

The `verifier` module tests the correctness of logical conditions generated for each basic path by verifying them with the Z3 SMT solver. It is split into two submodules: `z3_parser` and `z3_verifier`. These submodules handle the translation of Rust expressions into Z3-compatible logical formulas, and the verification of conditions respectively.

5.6.1 The `Z3` SMT Solver and Its Rust Crate

Z3 is a state-of-the-art Satisfiability Modulo Theories (SMT) solver developed by Microsoft Research [3]. It is used to determine the satisfiability of logical formulas under various background theories, such as arithmetic, and more.

In the context of the `verifier` module, Z3 plays a central role in checking the validity of logical conditions generated for basic paths. The Z3 solver operates on formulas expressed in an extension of the SMT-LIB format, a standardized language for describing SMT problems.

The Rust crate `z3` provides the tools necessary to construct an equivalent representation of these SMT-LIB formulas using Z3's AST within Rust. This allows us to create and manipulate logical conditions directly in Rust, which can then be verified for satisfiability or validity by the Z3 solver.

5.6.2 Why Use Z3 in Rust?

Using Z3 in Rust is advantageous for several key reasons:

- **Rust Integration:** The `z3` crate provides a bridge between Rust and Z3, enabling the construction and manipulation of logical conditions directly in Rust without needing to manually write SMT-LIB code.
- **API and Documentation:** Z3 offers extensive documentation, and the `z3` crate provides a clean API for building Z3 formulas using the Rust syntax.
- **AST Manipulation and Automation:** The `z3` crate allows the construction and manipulation of an AST, enabling automated condition building. To achieve this, we have to translate the `syn` AST (which represents Rust code) into the `z3` AST by ourselves, but by reasoning in terms of ASTs, we maintain a consistent framework that aligns our approach to source code analysis and logical condition generation.

- **Theory Support:** Z3 is performant and is supporting a wide range of theories, including arithmetic, bit-vectors, arrays, and quantifiers [3]. This provides flexibility for future extensions, ensuring that if we need to reason about more complex Rust constructs or data structures, we won't be limited by the capabilities of the SMT solver.

5.6.3 The Z3 Rust Crate

The `z3` crate provides high-level abstractions for building and manipulating Z3 formulas. It offers APIs for:

- Creating Z3 contexts and solvers.
- Constructing Z3 expressions (AST nodes) using Rust's type system.
- Asserting conditions into the solver.
- Checking satisfiability and retrieving models (counterexamples) when conditions fail.

In particular, the crate maps SMT-LIB constructs to Rust types, such as:

- `z3::ast::Int` for integer variables and expressions.
- `z3::ast::Bool` for boolean conditions and logical operations.
- `z3::Solver` to manage the solver state and interact with Z3.

These types provide methods for logical operations. For example this is how we would manually code the final condition derived for the basic path 3 at the end of Section 4.4:

```
1  // Z3 context and solver
2  let cfg = Config::new(); // New config
3  let ctx = Context::new(&cfg); //New context
4  let mut solver = Solver::new(&ctx); // New solver
5
6  // Variables for the conditions, in context
7  let n = ast::Int::new_const(&ctx, "n");
8  let i = ast::Int::new_const(&ctx, "i");
9  let sum = ast::Int::new_const(&ctx, "sum");
10
11 // Path 3 final condition:
12 // (i <= n + 1 && sum == (i - 1) * i / 2) implies
13 // (i <= n implies ((i + 1) <= n + 1 && (sum + i) == ((i + 1)
       - 1) * (i + 1) / 2))
14
15 // Compute next values for i and sum
16 let i_next = i.clone() + ast::Int::from_i64(&ctx,1); // i + 1
17 let sum_next = sum.clone() + i.clone(); // sum + i
18
19 // Current invariant: i <= n + 1 && sum == (i - 1) * i / 2
20 let condition_path_3_invariant_current = z3::ast::Bool::and(&
     ctx, &[
21     // i <= n + 1
22     &i.le(&(n.clone() + ast::Int::from_i64(&ctx, 1))),
```

```
23      // sum == (i - 1) * i / 2
24      &sum._eq(
25          &(&(i.clone() - ast::Int::from_i64(&ctx, 1))
26          * i.clone()
27          / ast::Int::from_i64(&ctx, 2))
28      ),
29  ]);
30
31  // Check if i <= n
32  let condition_path_3_i_le_n = i.le(&n); // i <= n
33
34  // Next invariant: (i + 1) <= n + 1 && (sum + i) == ((i + 1)
        - 1) * (i + 1) / 2
35  let condition_path_3_invariant_next = z3::ast::Bool::and(&ctx
        , &[
36      &i_next.le(&(n.clone() + ast::Int::from_i64(&ctx, 1))),
            // i + 1 <= n + 1
37      &sum_next._eq(
38          &(&(i_next.clone() - ast::Int::from_i64(&ctx, 1))
39          * i_next.clone()
40          / ast::Int::from_i64(&ctx, 2))
41      ), // sum + i == ((i + 1) - 1) * (i + 1) / 2
42  ]);
43
44  // Build the implication chain
45  let condition_path_3 = z3::ast::Bool::implies(
46      &condition_path_3_invariant_current,
47      &z3::ast::Bool::implies(
48          &condition_path_3_i_le_n,
49          &condition_path_3_invariant_next
50      ),
51  );
```

Code Listing 5.2: Example of z3 crate usage in Rust by hardcoding the final condition of path 3 in Section 4.4

The condition can be visually inspected by calling its `.to_string()` method, which outputs the condition in SMT-LIB format. This representation, as shown in Code Listing 5.3, provides a clear view of the logical structure.

```
1 (=> (and (<= i (+ n 1)) (= sum (div (* (- i 1) i) 2)))
2     (=> (<= i n)
3         (and (<= (+ i 1) (+ n 1))
4              (= (+ sum i) (div (* (- (+ i 1) 1) (+ i 1)) 2)))))
```

Code Listing 5.3: SMT-LIB format of the condition manually coded in Code Listing 5.2

5.6.4 `z3_verifier` Submodule

This submodule provides the interface to interact with the Z3 solver. Its main responsibilities are to verify logical conditions by asserting them in the Z3 solver and to print counterexample models when conditions fail.

There are two core methods in this submodule that we use to call the `z3_parser` and to interface with the Z3 verifier.

`verify_str_implication()` This method verifies a logical condition provided as a Rust string:

1. Initializes the Z3 solver.
2. Parses the input condition into a Rust AST using the `syn` crate.
3. Calls `z3_parser::generate_condition_and_vars()` to translate the AST into Z3-compatible formulas and variables.
4. Uses `verify_condition()` to check the condition's validity with the solver.

`verify_condition()` This method verifies a Z3 logical condition:

1. Pushes the solver's state and asserts the negation of the condition (proof by contradiction).
2. Checks the condition with the Z3 solver:
    - **Unsatisfiable:** The original condition is valid.
    - **Satisfiable:** A counterexample model is printed, detailing variable assignments.
    - **Unknown:** The solver could not determine validity.

The Figure 5.6 illustrates the high-level workflow of the core methods in the `z3_verifier` submodule.

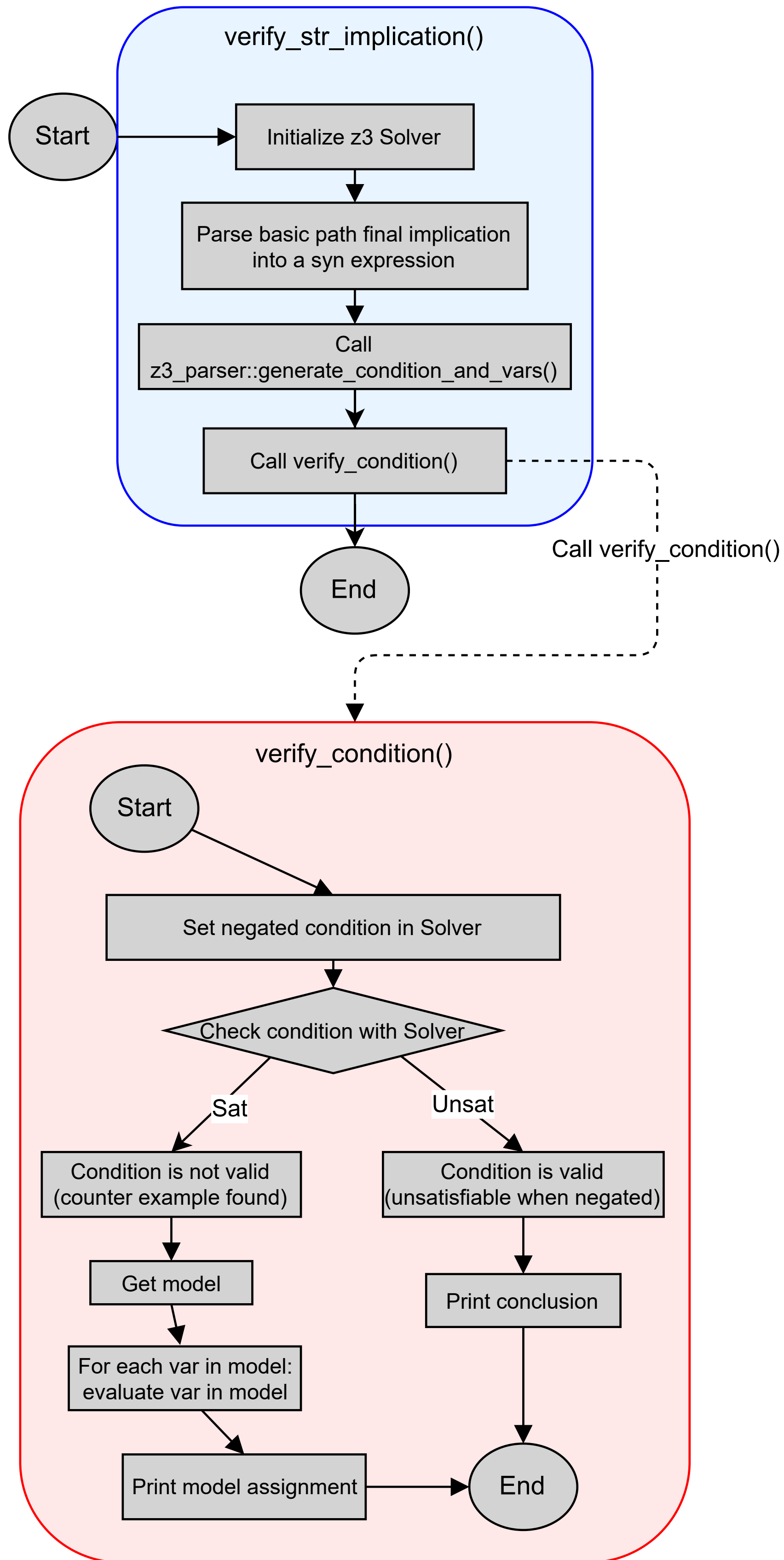


Figure 5.6: High-level workflow of the core methods of the `z3_verifier` submodule.

5.6.5 `z3_parser` Submodule

This submodule converts Rust logical expressions (ASTs) into Z3-compatible conditions and variable representations. It is responsible for parsing Rust expressions and translating

them into Z3 formulas, managing the creation and reuse of Z3 variables, and handling operations such as implications, negations, and arithmetic expressions.

Core Methods

`generate_condition_and_vars()` The main method that builds Z3 logical formulas:

1. Traverses the Rust `syn` AST of the condition.
2. Calls `generate_z3_ast()` to translate the Rust `syn` AST into a `z3` crate AST.
3. Manages Z3 variables with `get_or_create_var()` to ensure consistent mapping.
4. Handles implications and logical nesting, post-processing them for correctness.

`generate_z3_ast()` This method translates individual AST nodes into Z3 formulas:

- **Literals:** Converts integers and booleans into Z3 constants.
- **Binary Operations:** Handles logical (AND, OR, IMPLIES) and arithmetic operations.
- **Unary Operations:** Processes negations and other unary expressions.
- **Path Expressions:** Retrieves or creates variables from identifiers.
- **Parentheses:** Recursively processes inner expressions.

`get_or_create_var()` A utility method to manage Z3 variables:

- Checks if a variable exists in the variable map.
- If not, creates a new Z3 variable and adds it to the map.

Visual Representation of the `z3_parser` Submodule

The Figure 5.7 illustrates the high-level workflow of the core methods in the `z3_parser` submodule.

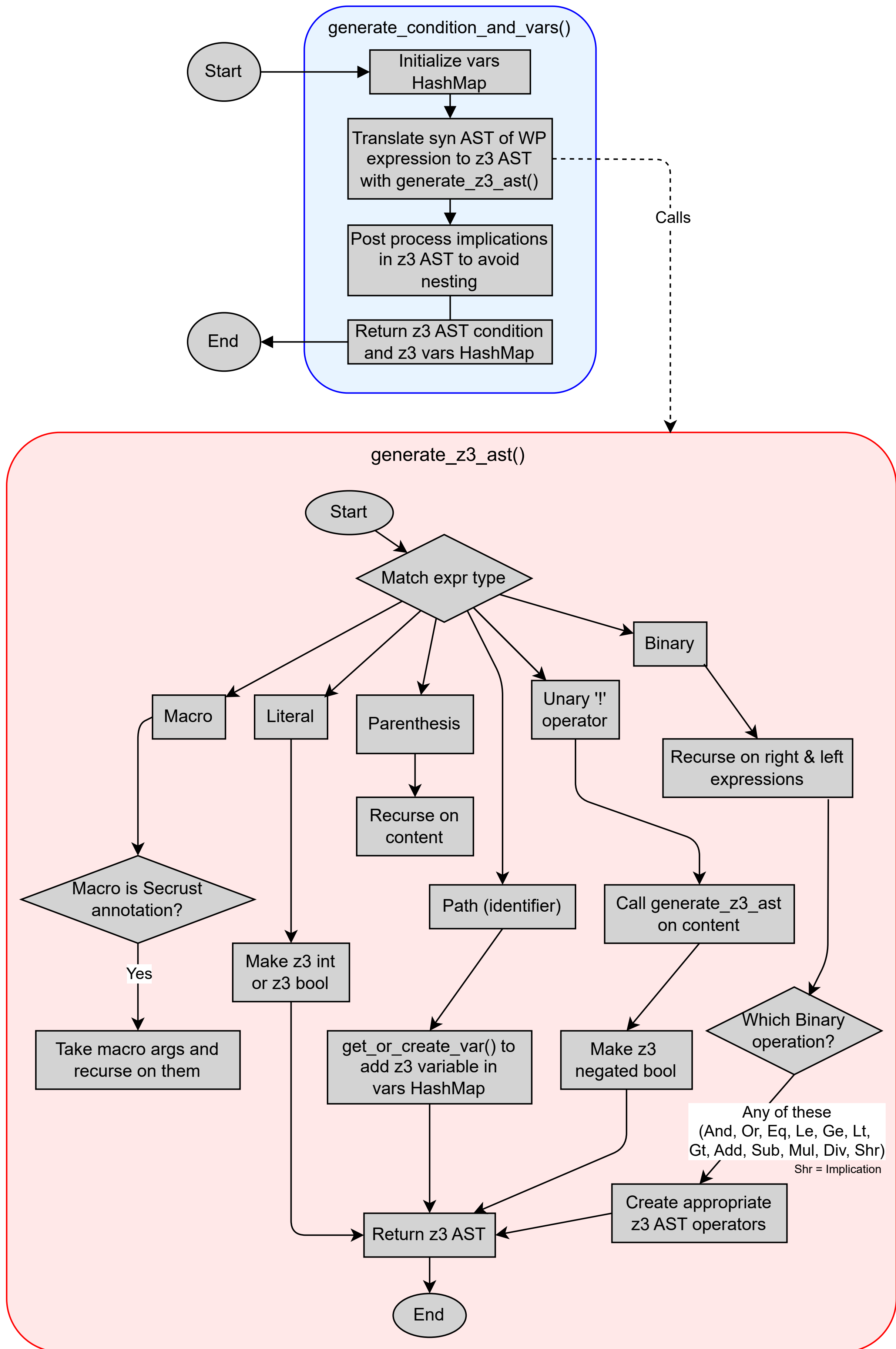


Figure 5.7: High-level workflow of the core methods of the `z3_parser` submodule.

Chapter 6

Validation

Our validation strategy focuses on two key aspects:

- **Condition Verification:** We test if the logical conditions generated for annotated Rust methods are indeed verified as valid. When conditions are not valid, we verify if the system provides counterexamples to demonstrate the failure. This process ensures that our verification system correctly distinguishes valid logical conditions from invalid ones.
- **Programmatic Construction of Conditions:** We validate the programmatic construction of logical conditions on multiple arithmetic examples. This validation step ensures that the system can automatically generate and verify conditions for Rust code that respects our syntax limitations.

6.1 Condition Verification

The Secrust tool outputs the results of verifying logical conditions in Rust methods annotated with preconditions, postconditions, and invariants. The primary goal of this validation process is to ensure that the conditions are properly verified and that counterexamples are detected when they are not satisfied. Below, we detail the key components of this output, using two examples to highlight the verification process and possible outcomes.

6.1.1 Successful Verification

The following output is produced when the verification of the `sum_first_n` method in Code Listing 4.1 succeeds.

```
1  Running Secrust verification on file: "src/main.rs"
2  Generate DOT graph: true
3  file path: "src/main.rs"
4  AST successfully parsed for file "src/main.rs"
5  ---------
6  Final implication for Path 1: pre ! (n >= 0) >> invariant !
      ((1) <= n + 1 && (0) == ((1) - 1) * (1) / 2)
7
8  Generated Z3 Condition:
9  (=> (>= n 0) (and (<= 1 (+ n 1)) (= 0 (div (* (- 1 1) 1) 2)))
      )
10
11 Condition is valid (unsatisfiable when negated).
```

```
12
13 Verification completed for "pre ! (n >= 0) >> invariant !
      ((1) <= n + 1 && (0) == ((1) - 1) * (1) / 2)"
14 ---------
15
16 ---------
17 Final implication for Path 2: invariant ! (i <= n + 1 && sum
      == (i - 1) * i / 2) >> ! (i <= n) >> post ! (sum == n * (n
       + 1) / 2)
18
19 Generated Z3 Condition:
20 (=> (and (<= i (+ n 1)) (= sum (div (* (- i 1) i) 2)))
21     (=> (not (<= i n)) (= sum (div (* n (+ n 1)) 2))))
22
23 Condition is valid (unsatisfiable when negated).
24
25 Verification completed for "invariant ! (i <= n + 1 && sum ==
       (i - 1) * i / 2) >> ! (i <= n) >> post ! (sum == n * (n +
       1) / 2)"
26 ---------
27
28 ---------
29 Final implication for Path 3: invariant ! (i <= n + 1 && sum
      == (i - 1) * i / 2) >> (i <= n) >> invariant ! ((i + 1) <=
       n + 1 && (sum + i) == ((i + 1) - 1) * (i + 1) / 2)
30
31 Generated Z3 Condition:
32 (=> (and (<= i (+ n 1)) (= sum (div (* (- i 1) i) 2)))
33     (=> (<= i n)
34         (and (<= (+ i 1) (+ n 1))
35              (= (+ sum i) (div (* (- (+ i 1) 1) (+ i 1)) 2)))
                  ))
36
37 Condition is valid (unsatisfiable when negated).
38
39 Verification completed for "invariant ! (i <= n + 1 && sum ==
       (i - 1) * i / 2) >> (i <= n) >> invariant ! ((i + 1) <= n
       + 1 && (sum + i) == ((i + 1) - 1) * (i + 1) / 2)"
40 ---------
41
42 DOT graph saved as: "src/graphs/main/main.dot"
43 Verification completed successfully.
```

Code Listing 6.1: Example of a successful verification for the `sum_first_n` method

Explanation of the output:

- **File and Arguments:** The file being analyzed is `src/main.rs`. The tool is called with the argument `--dot`, meaning it will generate a DOT file representing the CFG.
- **Verification Results:** The final implications for Paths 1, 2, and 3 are displayed. For each path, the logical implication is transformed into the z3 compatible format and the SMT-LIB representation is printed before we verify its validity. In this case,

all paths are verified as `valid (unsatisfiable when negated)`, meaning the logic holds for all possible executions.

6.1.2 Counterexample Found

In this scenario, we intentionally modify the invariant for the `sum_first_n` method from `i <= n + 1` to `i <= n`. This change introduces a logical error, and the Secrust verifier produces a counterexample.

```
---------
Final implication for Path 3: invariant ! (i <= n && sum == (
   i - 1) * i / 2) >> (i <= n) >> invariant ! ((i + 1) <= n
   && (sum + i) == ((i + 1) - 1) * (i + 1) / 2)

Generated Z3 Condition:
(=> (and (<= i n) (= sum (div (* (- i 1) i) 2)))
    (=> (<= i n)
        (and (<= (+ i 1) n) (= (+ sum i) (div (* (- (+ i 1)
            1) (+ i 1)) 2)))))

Condition is not valid (counterexample found).

Counterexample model assignments:
sum = 1
i = 2
n = 2

Verification completed for "invariant ! (i <= n && sum == (i
   - 1) * i / 2) >> (i <= n) >> invariant ! ((i + 1) <= n &&
   (sum + i) == ((i + 1) - 1) * (i + 1) / 2)"
---------


DOT graph saved as: "src/graphs/main/main.dot"
Verification completed successfully.
```

Code Listing 6.2: Example of a failed verification for the `sum_first_n` method

Explanation of the output:

- **Counterexample Found:** The Secrust verifier identifies that the condition in Path 3 is `not valid`, meaning the logical implication does not hold for all possible executions. This failure occurs because the incorrect invariant `i <= n` is too restrictive and is violated during the loop execution.
- **Why it Fails:** The invariant `i <= n` is too restrictive, as i is incremented within the loop, and it can exceed n during certain iterations. For example, with `n = 2` and `i = 2`, incrementing i produces `i = 3`, which violates the invariant `i <= 2`. This is confirmed by the Z3 solver, which produces a counterexample where `sum = 1`, `i = 2`, and `n = 2`, causing the condition `i + 1 <= n` to fail since `3 <= 2` is false. This demonstrates that a more general invariant like `i <= n + 1` is required.
- **Graph Output:** As with the previous case, a DOT file is generated to visualize the control flow graph, which helps understand how control flows and where the error may have occurred.

6.2 Programmatic Condition Validation for Arithmetic Methods

First we will manually derive the WP formula for the basic path 3 of both `sum_of_odds` and `sum_of_squares` and then compare with the output of Secrust.
Path 3 is the path we take inside the while loop when the condition of the loop is true. It is a similar path to the path 3 in figure 4.2

```
fn sum_of_odds(n: i32) -> i32 {
    pre!(n > 0);
    let mut sum = 0;
    let mut i = 1;
    let mut count = 0;
    invariant!(count <= n && sum == count * count && i == 2 *
        count + 1);
    while count < n {
        sum = sum + i;
        i = i + 2; // Move to the next odd number
        count = count + 1;
    }
    post!(sum == n * n); // Sum of first n odd numbers is n^2
    return sum;
}
```

Code Listing 6.3: Code for `sum_of_odds`

Weakest Precondition for the basic path 3 of `sum_of_odds`

1. **Invariant:**

$$count \leq n \quad \&\& \quad sum == count * count \quad \&\& \quad i == 2 * count + 1$$

2. **Operation:** `count = count + 1`

Updated Invariant:

$$(count + 1) \leq n \quad \&\& \quad sum == (count + 1) * (count + 1)$$
$$\&\& \quad i == 2 * (count + 1) + 1$$

3. **Operation:** `i = i + 2`

Updated Invariant:

$$(count + 1) \leq n \quad \&\& \quad sum == (count + 1) * (count + 1)$$
$$\&\& \quad (i + 2) == 2 * (count + 1) + 1$$

4. **Operation:** `sum = sum + i`

Updated Invariant:

$$(count + 1) \leq n \quad \&\& \quad (sum + i) == (count + 1) * (count + 1)$$
$$\&\& \quad (i + 2) == 2 * (count + 1) + 1$$

5. **Condition Assumed:** `count < n`

Updated Invariant:

$$(count < n) \Rightarrow (count + 1) \leq n \quad \&\& \quad (sum + i) == (count + 1) * (count + 1)$$

$$\&\& \quad (i+2) == 2 * (count+1)+1$$

6. Invariant Asserted:
Final WP Formula:

$$(count \leq n \quad \&\& \quad sum == count * count \quad \&\& \quad i == 2 * count + 1) \Rightarrow$$

$$(count < n) \Rightarrow (count+1) \leq n \quad \&\& \quad (sum+i) == (count+1) * (count+1)$$
$$\&\& \quad (i+2) == 2 * (count+1)+1$$

Which matches the final output result given by Secrust:

```
invariant!(count <= n && sum == count * count && i == 2 *
    count + 1) >> (count < n) >> invariant!((count + 1) <= n
    && (sum + i) == (count + 1) * (count + 1) && (i + 2) == 2
    * (count + 1) + 1)
```

Code Listing 6.4: Output WP formula for path 3 of sum_of_odds

Now we do the same process for `sum_of_squares`

```
fn sum_of_squares(n: i32) -> i32 {
    pre!(n >= 0);
    let mut sum = 0;
    let mut i = 1;
    invariant!(i <= n + 1 && sum == (i - 1) * i * (2 * i - 1)
        / 6); // Formula for sum of squares up to (i-1)
    while i <= n {
        sum = sum + i * i;
        i = i + 1;
    }
    post!(sum == n * (n + 1) * (2 * n + 1) / 6); // Formula
       for sum of squares up to n
    return sum;
}
```

Code Listing 6.5: Code for `sum_of_squares`

Weakest Precondition for the basic path 3 of`sum_of_squares`

1. Invariant:

$$i \leq n+1 \quad \&\& \quad sum == (i-1) * i * (2 * i - 1)/6$$

2. Operation: $i = i + 1$
Updated Invariant:

$$(i+1) \leq n+1 \quad \&\& \quad sum == ((i+1)-1) * (i+1) * (2 * (i+1) - 1)/6$$

3. Operation: $sum = sum + i * i$
Updated Invariant:

$$(i+1) \leq n+1 \quad \&\& \quad (sum + i * i) == ((i+1)-1) * (i+1) * (2 * (i+1) - 1)/6$$

4. Condition Assumed: $i \leq n$
Updated Invariant:

$$(i \leq n) \Rightarrow (i+1) \leq n+1 \quad \&\& \quad (sum+i*i) == ((i+1)-1)*(i+1)*(2*(i+1)-1)/6$$

5. Invariant Asserted: Final WP Formula:

$$(i \leq n+1 \quad \&\& \quad sum == (i-1)*i*(2*i-1)/6) \Rightarrow$$

$$(i \leq n) \Rightarrow (i+1) \leq n+1 \quad \&\& \quad (sum+i*i) == ((i+1)-1)*(i+1)*(2*(i+1)-1)/6$$

Which matches the final output result given by Secrust:

```
invariant!(i <= n + 1 && sum == (i - 1) * i * (2 * i - 1) /
    6) >> (i <= n) >> invariant!((i + 1) <= n + 1 && (sum + i
    * i) == ((i + 1) - 1) * (i + 1) * (2 * (i + 1) - 1) / 6)
```

Code Listing 6.6: Output WP formula for path 3 of sum_of_squares

Now that we know that Secrust properly derived the WP formula, we have to check if it can properly translate the Rust format into SMT-LIB.

SMT-LIB Output for Invariant Paths

For the methods `sum_of_odds` and `sum_of_squares`, we look at the implication generated by the code and the SMT-LIB outputed by the `z3` crate and see that both are equivalent.

Example Path for `sum_of_odds`: For Path 3, the final implication is generated as follows:

```
Generated Z3 Condition:
(=> (and (<= count n) (= sum (* count count)) (= i (+ (* 2 count
    ) 1)))
     (=> (< count n)
          (and (<= (+ count 1) n)
               (= (+ sum i) (* (+ count 1) (+ count 1)))
               (= (+ i 2) (+ (* 2 (+ count 1)) 1)))))
```

Code Listing 6.7: SMT-LIB format for `sum_of_odds`

Example Path for `sum_of_squares`: For Path 3, the final implication is:

```
Generated Z3 Condition:
(=> (and (<= i (+ n 1)) (= sum (div (* (- i 1) i (- (* 2 i) 1))
    6)))
     (=> (<= i n)
          (and (<= (+ i 1) (+ n 1))
               (= (+ sum (* i i))
                  (div (* (- (+ i 1) 1) (+ i 1) (- (* 2 (+ i 1))
                      1)) 6))))))
```

Code Listing 6.8: SMT-LIB format for `sum_of_squares`

We can see that both SMT-LIB representations correspond with the actual WP derived by Secrust.

This validation illustrates how Secrust accurately translates the program's logic into verifiable conditions.

Chapter 7

Use of AI

Artificial Intelligence (AI), particularly Large Language Models (LLMs), have been utilized to assist in the development of this thesis, especially for tasks such as LaTeX formatting and English grammar corrections. Beyond these, AI contributed to high-level reasoning, architectural considerations, and code structuring. However, every AI-generated suggestion has been carefully evaluated, given the complexity of the project and the potential of current AI tools to produce inaccuracies or "hallucinations" in their responses.

The effectiveness of AI tools in the context of this thesis lies in their ability to propose alternative approaches and identify potential areas for improvement. While these suggestions were not always ideal, they often provided new perspectives on the challenges encountered.

7.1 AI-Driven Support

AI assisted in the following ways:

- LaTeX formatting for academic writing, reducing the time spent on document structuring.
- Helped with the visual formatting of DOT graph nodes and updating DOT files for visual representation in this paper.
- Helped with converting between formats, such as DOT to XML, to facilitate the editing and visualization of graphs.
- Providing suggestions to restructure Rust code for clarity and maintainability.
- Debugging syntax and logic errors, offering detailed explanations and alternative approaches to resolve issues.
- Helped with the use of specific libraries and platform configurations across multiple operating systems, minimizing setup errors and reducing setup time for tools like Z3 and C compilers.

7.2 Personalized AI

Wanting a specialized assistant for our specific problem, we created a custom GPT tailored to our project. A custom GPT is a personalized version of ChatGPT, which allows users to adapt the model to specific contexts by incorporating their own data.

To create the custom GPT, we downloaded our private chat history and filtered the data to retain only relevant discussions, particularly those involving Rust-related questions.

By incorporating these chat logs and project materials into its knowledge base, we enabled the assistant to align its suggestions with the way we code, reason, and write about the project.

Although this approach did not entirely prevent the tool from generating hallucinated responses, it proved valuable due to its ability to reference the "knowledge" we provided. This allowed it to offer suggestions in line with our specific subject rather than providing general or unhelpful answers.

7.3 Reflection on AI Usage

AI was useful in this thesis for tasks such as checking text, providing suggestions, and formatting content. It proved helpful in suggesting alternative approaches to solving project-related problems, debugging issues, and navigating specific libraries when documentation was insufficient

However, AI had difficulty reasoning about complex ideas related to the project, and its suggestions often required careful evaluation.

Despite these challenges, AI contributed by simplifying certain tasks and assisting in refining ideas.

Chapter 8

Conclusion

This thesis explores the integration of formal verification for annotated Rust code by analyzing source code. As seen in Chapter 3 popular Rust verification projects that analyze execution paths for formal verification use the Rust Mid-level Intermediate Representation (MIR) generated by the compiler to take advantage of control flows. As the core focus of this thesis is to analyze Rust source code directly, we needed a way to process annotated Rust code and generate control flow graphs without relying on the compiler's MIR representation. While this approach introduced an additional layer of complexity, it offers a notable advantage: it keeps the formal verification process closely aligned with the original source code written by developers, enabling them to better understand how annotations and proofs relate to their code without the need to navigate through intermediate abstractions.

8.1 Key Contributions

1. **Control Flow Graph Generation and Analysis:** Because no existing tools or crates in Rust were found to perform this task at the time of writing this thesis, we developed a module to analyze parsed Rust source code and generate a Control Flow Graph (CFG) for each annotated method. This module constructs the overall control flow graph and identifies all basic paths within the analyzed method. Additionally, it provides the capability to generate a DOT-format representation of the control flow, enabling developers to visually inspect the code's structure using graph visualization softwares.

2. **Weakest Precondition Calculus & Rust source code:** This thesis implements Weakest Precondition (WP) calculus for verifying logical correctness in Rust programs.

   A macro-based system was developed to annotate Rust code with preconditions, invariants, and postconditions. These annotations are processed using the methodology outlined in Chapter 2, which involves analyzing each basic path of the annotated methods to derive the weakest precondition.

3. **Integration with Z3 SMT Solver:** Secrust integrates the Z3 SMT solver to enable the formal verification of Rust programs. The Z3 solver evaluates logical conditions derived from the annotated Rust code.

   The project translates the conditions to be verified for correctness into a format supported by the `z3` Rust crate, which is then passed to Z3 for verification. The system will then determine whether the derived conditions are satisfiable or if potential violations exist.

4. **Developer-Friendly Integration** By packaging our verification framework as a Rust crate and leveraging procedural macros for annotation, we stay as close as possible to Rust's syntax and ecosystem. This approach minimizes overhead and simplifies integration into existing Rust projects, allowing developers to easily annotate and verify methods on the go.

   Additionally, because we operate directly on the source code rather than on the Rust compiler's MIR, the solver (Z3) returns counterexamples that refer to the *original* variables in the Rust code. This makes counterexamples far more understandable and actionable for developers who are not experts in verification or compiler internals. In other words, developers can see exactly which source variables triggered the issue, rather than needing to interpret compiler-specific Intermediate Representation variable names.

   As a result, our framework is easier to use and requires less specialized knowledge, allowing developers to focus on verifying their code without worrying about low-level compiler details.

5. **Modular Design for Extensibility:** The framework is built with a modular architecture, making it easier to add new features or abstractions without affecting existing functionality.

8.2 Challenges and Lessons Learned

The approach taken in this thesis — sticking as closely as possible to Rust source code — introduced significant challenges in interpreting and processing the code. These challenges and the corresponding lessons learned are detailed below.

8.2.1 Interpreting Rust Source Code

- **CFG Generation:** Due to the absence of tools capable of automatically generating a CFG from Rust source code, we developed a dedicated module to handle this task. This module interprets the Rust Abstract Syntax Tree (AST) generated by the `syn` crate to construct the CFG. This means that we ultimately need to interpret the entire Rust syntax, including how the analyzed code updates variables appearing in annotated assertions. This process adds significant complexity, as it demands a thorough understanding of how each operation in the source code modifies variables. At present, the implementation is limited to supporting simple Rust constructs, as demonstrated in the examples provided in Chapter 6, and does not yet handle more advanced syntax.

- **Assertion Variable Updates:** Beyond CFG generation, we had to manually interpret the AST structures to track how variables in the annotated assertions are updated in the source code. This required creating logic to identify and keep track of these updates along the CFG paths.

- **Branching Factor of Conditions:** Each conditional statement in the code introduces new branches in the CFG, leading to an exponential growth in the number of execution paths as the number of conditions increases. Specifically, for N independent binary conditions, the number of paths can grow as 2^N. This branching factor adds significant complexity to the verification process, as every branch must be analyzed to ensure logical correctness. However, this challenge could be somewhat mitigated by verifying smaller functions (submethods) individually, thereby reducing the scope

of analysis at any given time. Note that our current tool does not yet support function-call semantics. Future work includes extending our framework to handle function calls properly, which would then allow developers to verify submethods —and subsequently the parent methods— incrementally.

8.2.2 Integrating the Z3 Solver

- **Translation to Z3 AST:** The integration of the Z3 solver was made easier by the availability of an existing crate linking Rust to Z3. However, translating the conditions from the `syn` AST into the Z3 AST format posed a challenge. This process involved interpreting the Rust AST and creating the corresponding `z3` crate AST node structures.

8.2.3 Lessons Learned

- **Complexity of Source Code Analysis:** As described above, generating CFGs and analyzing Rust source code adds complexity, especially in understanding how variables are modified during execution and updating them in assertions. However, this approach also gives us more freedom and flexibility to decide how things are interpreted and verified. By working directly with the source code, the verification process stays closely connected to the original code, making it easier to understand and customize.

- **Control Over Verification:** By directly analyzing Rust source code instead of relying on the compiler's MIR representation, we avoid issues caused by compiler optimizations or changes to the MIR language. This approach gives us more freedom to decide how abstractions are handled and conditions are represented in the Z3 SMT-LIB format, allowing for greater flexibility in verification. However, this also means that if the compiler introduces an error during optimization, it would not be detected by our framework, as we reason solely on the source code and not on the optimized MIR.

- **Benefits of Modularization:** The challenges encountered during the development of the CFG builder, particularly when extending the code after its initial implementation, highlighted the need for a modular architecture. The code became so convoluted that it was necessary to refactor it into modular components, allowing for easier additions to the Rust syntax handled by the CFG builder and other related modules. Modularization simplifies maintenance and ensures that future adaptations and extensions can be implemented with minimal disruption to the existing codebase.

8.3 Final Thoughts

Through this thesis, we highlight the benefits of focusing formal verification on the actual Rust source code. This approach allows developers to verify their code as they write it, using simple commands, while maintaining a clear connection between the verification process and the code itself. The project code is open-source and available at https://github.com/vasilevlaicu/secrust.

Chapter 9

Future Work

This thesis lays a groundwork for the formal verification of annotated Rust source code, but there is room for improvements and extensions.

9.1 Extending Rust Syntax Support

We could start by handling the following cases as part of our journey toward supporting the full Rust syntax:

- **Function Calls:** To extend the framework, we need to implement support for verifying function calls. This could be achieved using uninterpreted functions in Z3, allowing the framework to symbolically represent function calls. This extension would also help address part of the branching issue discussed in Section 8.2.1.
- **Improved Logical Implication Representation:** The tool currently uses the `>>` (right bitshift) operator to represent logical implications. While this leverages Rust's existing syntax, it introduces limitations, as `>>` cannot simultaneously represent both logical implication and actual bitshift operations. Future work could explore introducing a custom syntax, such as `=>`, to represent implications. However, this would require modifying the parsing behavior of the `syn` crate or adopting alternative parsing frameworks, as `=>` is not valid Rust syntax except in specific cases (such as match arms and closures). We aim to avoid solutions such as a macro representation (such as `imply!()`), as these could make conditions less readable for users.
- **Enums and Structs:** Enums and structs could be modeled using Z3's algebraic datatypes. For example, Rust `enum` types could be mapped to Z3 Scalars, and structs could be represented as Z3 Records.
- **Closures:** Closures are anonymous functions in Rust that can capture variables from their surrounding environment and manipulate them. For example, closures are often used in higher-order functions like `filter` and `map` to process collections:

```rust
    let numbers = vec![1, 2, 3, 4];
    let transformed: Vec<_> = numbers
        .iter()
        .filter(|&x| x % 2 == 0) // Keep only even
           numbers
        .map(|x| x * 3)           // Multiply each by 3
        .collect();
    println!("{:?}", transformed); // Outputs: [6, 12]
```

Here, the closure `|&x| x % 2 == 0` filters even numbers, while `|x| x * 3` transforms each remaining element by multiplying it by 3. A submodule for closures would handle their unique characteristics, making it easier to extend support for related Rust features like `map` and `filter`. The difficulty would mostly come from tracking how closures modify variables present in the assertions.

- **Dynamic Structures:** Dynamic structures such as hash maps and vectors could be modeled using Z3's arrays or uninterpreted functions. These abstractions would allow the tool to represent dynamic collections symbolically. For example, hash maps could be represented using Z3 functions, where keys map to values:

  ```
          (declare-fun hashmap (Key) Value)
  ```

  Similarly, vectors could be represented as Z3 arrays.

9.2 Industry Case: MultiversX Smart Contracts

Smart contracts are a real-world use case that aligns with the capabilities of Secrust and would require just a few updates to verify certain types of contracts.

MultiversX [17] is a blockchain that operates on a proof-of-stake consensus mechanism and supports smart contracts written in any language as long as they compile to Web Assembly. These smart contracts enable developers to implement decentralized applications for various use cases, such as token staking, exchanges, and crowdfunding. The MultiversX Virtual Machine executes these contracts and is complemented by a Rust Software Development Kit [18] (SDK), which provides tools for development and testing. While the SDK includes infrastructure for simulating blockchain interactions and unit testing, it currently lacks integrated formal verification capabilities.

Formal verification tools like Secrust could enhance the reliability of MultiversX smart contracts, ensuring logical correctness for critical applications in finance and beyond. This need was communicated to us by a Smart Contract audit team within the MultiversX ecosystem, who mentioned that a tool for annotating Rust code to formally verify contracts was something they were actively looking for but had not yet found a suitable solution.

- **Arithmetic Operations:** In MultiversX smart contracts, `BigUint` is used to represent financial amounts. `BigUint` is an unsigned integer type that supports operations on large values and can be converted to and from `u64` and other types. While `BigUint` has the capability to handle values larger than `u64`, its size can vary depending on the application.

  To model `BigUint` in Z3, we use the `Int`, which is based on the SMT-LIB `Ints` theory [19]. This theory defines integers as having arbitrary precision, making them well-suited for handling very large values often required in smart contracts. However, since `BigUint` is an unsigned integer type, we define the variable and add a constraint to ensure that it is always non-negative:

```
        (declare-fun biguint_var () Int)
        (assert (>= biguint_var 0))
```

This ensures that the variable `biguint_var` is explicitly declared as an integer and constrained to be non-negative, accurately modeling the unsigned nature of `BigUint`.

However, smart contracts also frequently rely on `u64` values, especially for financial computations such as token balances and staking amounts, which typically fit within the `u64` range. For these cases, Z3's bit-vectors provide a more accurate representation. Bit-vectors are fixed-width binary representations that model the behavior of types like `u64`.

- **Storage Mappers:** Storage mappers are a fundamental feature of blockchain virtual machines, used by smart contracts to store and manage data on the blockchain. For example, `Single-Value Mappers` are commonly employed to map a value, such as a deposited amount, to an interacting address. These can be effectively modeled in Z3 using uninterpreted functions, where a function maps an address to a stored value:

```
    (declare-fun deposit (Address) Int)
```

Updates to storage mappers can be represented by introducing new function states to reflect changes:

```
    (assert (= (deposit_new caller) (+ (deposit_old caller)
       payment)))
```

To track previous values, we propose introducing an `old!()` procedural macro that can be used within the existing assertion macros.

By addressing these areas, we show that Secrust can grow into a useful tool for verifying Rust programs, whether for general use or specific industries.

Bibliography

[1] S. Overflow, "Stack overflow developer survey 2024," 2024, accessed: November 29, 2024. [Online]. Available: https://survey.stackoverflow.co/2024/

[2] W. Bugden and A. D. Alahmar, "Rust: The programming language for safety and performance," *CoRR*, vol. abs/2206.05503, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2206.05503

[3] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[4] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 136:1–136:27, 2020. [Online]. Available: https://doi.org/10.1145/3428204

[5] E. W. Dijkstra, *A Discipline of Programming.* Prentice-Hall, 1976. [Online]. Available: https://www.worldcat.org/oclc/01958445

[6] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, "Verifying dynamic trait objects in rust," in *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022.* IEEE, 2022, pp. 321–330. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP55303.2022.9794041

[7] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: A bounded verifier for rust (N)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 75–80. [Online]. Available: https://doi.org/10.1109/ASE.2015.77

[8] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 66:1–66:34, 2018. [Online]. Available: https://doi.org/10.1145/3158154

[9] R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *J. Funct. Program.*, vol. 28, p. e20, 2018. [Online]. Available: https://doi.org/10.1017/S0956796818000151

[10] INRIA, "The rocq prover, previously coq," 2024, accessed: 2024-11-13. [Online]. Available: https://coq.inria.fr/

[11] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, “Mirchecker: Detecting bugs in rust programs via static analysis,” in *CCS ’21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 2183–2196. [Online]. Available: https://doi.org/10.1145/3460120.3484541

[12] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, “The prusti project: Formal verification for rust,” in *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, ser. Lecture Notes in Computer Science, J. V. Deshmukh, K. Havelund, and I. Perez, Eds., vol. 13260. Springer, 2022, pp. 88–108. [Online]. Available: https://doi.org/10.1007/978-3-031-06773-0_5

[13] P. Müller, M. Schwerhoff, and A. J. Summers, “Viper: A verification infrastructure for permission-based reasoning,” in *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, ser. Lecture Notes in Computer Science, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer, 2016, pp. 41–62. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2

[14] X. Denis, J.-H. Jourdan, and C. Marché, “The CREUSOT Environment for the Deductive Verification of Rust Programs,” Inria Saclay - Île de France, Research Report RR-9448, Dec. 2021. [Online]. Available: https://inria.hal.science/hal-03526634

[15] J. Filliâtre and A. Paskevich, “Why3 - where programs meet provers,” in *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. Felleisen and P. Gardner, Eds., vol. 7792. Springer, 2013, pp. 125–128. [Online]. Available: https://doi.org/10.1007/978-3-642-37036-6_8

[16] C. W. Barrett, A. Stump, and C. Tinelli, “The smt-lib standard version 2.0,” 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:7943149

[17] MultiversX, “Multiversx rust sdk documentation,” 2024, accessed: December 20, 2024. [Online]. Available: https://docs.multiversx.com/sdk-and-tools/sdk-rust

[18] ——, “Multiversx documentation,” 2024, accessed: December 20, 2024. [Online]. Available: https://docs.multiversx.com/

[19] S.-L. Initiative, “Theories: Ints,” 2017, accessed: January 3, 2025. [Online]. Available: https://smt-lib.org/theories-Ints.shtml

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**