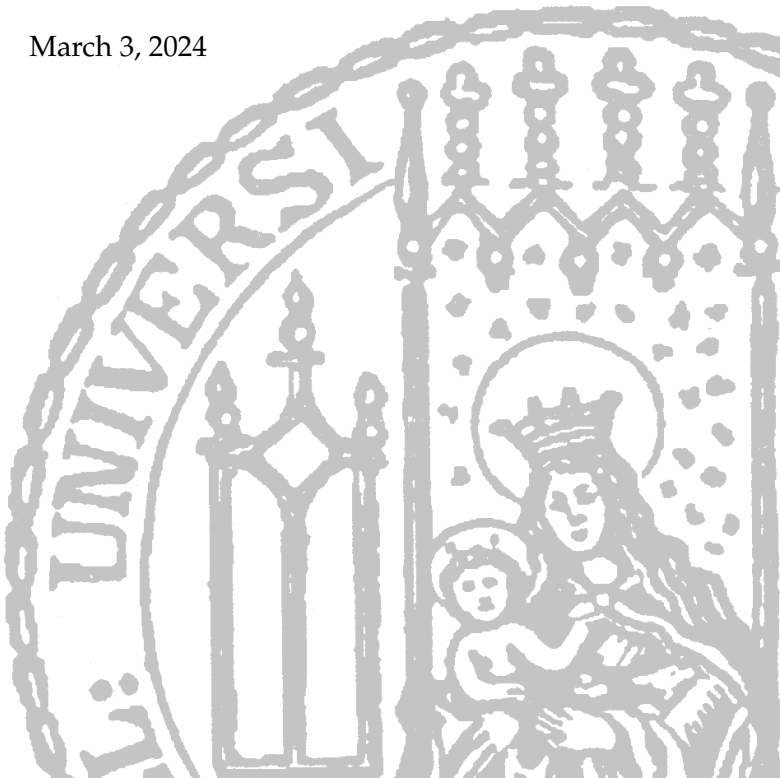
INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

A Library for Unit Verification

Marko Ristic

Bachelor Thesis

Supervisor	Prof. Dr. Dirk Beyer
Mentor	Thomas Lemberger
Submission Date	March 3, 2024



Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, March 3, 2024 Marko Ristic

Abstract

Motivated by the enduring popularity of C and the demand for a reliable verification framework, our approach aims to overcome challenges in readability, scalability, and usability. This thesis addresses the critical challenge of enhancing the reliability and correctness of formal verification. Focused on projects written in the C language and widely-used conventions of the International Competition on Software Verification (SV-COMP), our goal is to develop a dedicated C library that supports users writing unit-test-like verification tasks and a powerful runner to execute them. The runner offers batch execution of tasks and visualisation of the results performed by formal verifiers. By conducting experiments on the coreutils project and applying verification tasks to functions like `cksum`, `cat`, `wc`, and `echo`, we showcase the practical viability of our methodology. The results signify a significant step forward in formal verification, offering broad applicability and serving as a valuable guide for developers and researchers in the field.

Contents

List of Figures

1 Introduction

This thesis centres around formal verification. Formal verification of a source code provides an excellent way to increase the quality of a software project. It achieves this by generating mathematical models of software that are then verified against predefined properties. By proving that software stands by these properties, we can ensure its correctness and reliability.
Specifically, our research formally verifies projects written in C language. To help with it, we aim to develop a C library for software verification. It also involves writing tasks akin to unit tests to verify specific components of a project. Additionally, we will create a runner that would be powerful enough to understand, process and execute these tasks.

1.1 Thesis Goals

Motivation. Despite the enduring popularity of the C language, there remains a significant demand for a reliable verification framework. At the same time, formal verification ensures software correctness, and challenges like readability, scalability, and usability emerge, especially in the context of existing C tools.
Our research targets these challenges, focusing on enhancing usability. While various formal verification tools for C exist, many lack certain features. Frama-C's ACSL [13] provides similar functionalities but requires projects to be initiated with annotations in mind, potentially restricting its use for projects not initially designed for ACSL integration. Comparable approaches, such as the proprietary AWS library [7], often limit accessibility. By creating a dedicated C library for software verification, formulating unit-test-like tasks, and developing a task execution runner, we aim to overcome these challenges and make formal verification in C projects more accessible.

1.2 Overall Approach

Library Methods. There are already conventions that indicate verifiers to do certain things such as `__VERIFIER_nondet_int()`, instructs verifiers to utilise random integers in the verification process or `__VERIFIER_assume(x > 0)` which assumes that `x` is greater than 0. The library is built around these conventions. We wanted to expand these basic conventions to make them easier to use and read.

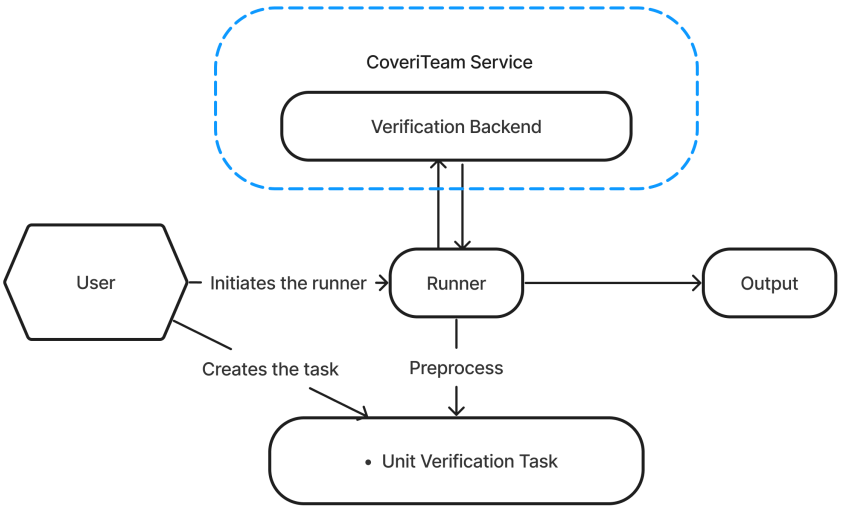


Figure 1.1: Example for an overview of a unit verification pipeline

Additionally, the library provides utility functions for defining conditions on complex data structures. Furthermore, we are introducing new functions to facilitate the definition of properties at the code level for verification.
To illustrate the practical benefits of our library, consider the example in Listing 1. The `vt_oneOfStrings` method builds upon established conventions. Using `__VERIFIER_nondet_int()` allows the selection of a string from a provided array, simplifying the process of generating diverse inputs for verification.

```
1  /* Returns a nondeterministic integer value. */
2  int vt_anyInt() { return __VERIFIER_nondet_int(); }
3
4  /* Selects one of the provided strings using a nondeterministic function.
5     The size parameter represents the size of the string array, and *strings[]
6     is a pointer to the array of strings. */
7  const char *vt_oneOfStrings(const char *strings[], size_t size) {
8    int index = vt_anyInt();
9    vt_assume(index >= 0 && index < size);
10   return strings[index];
11 }
```

Listing 1: Nondeterministic integer and string selection in C

Unit Verification Tasks. With the library methods in place, we get to verification tasks. These tasks consist of verification harnesses that test the function of a project under verification. We can choose any function of a project to verify (hence unit-test-like) and write a harness using our library. This granularity allows for a more accurate examination of each function's behaviour, ensuring alignment with specified requirements and adherence to intended logic. Isolating specific functions simplifies debugging and troubleshooting processes. Utilising verification properties, such as `unreach_call()`, unit verification tasks explicitly define and enforce expected behaviour during function execution. These properties enhance the verification process by setting conditions that functions must meet, contributing to a more robust and predictable system.

```
1  #include <assert.h>
2  #include "convert.h"
3
4  int convert(int temperatureInCelsius) {
5    int inKelvin = temperatureInCelsius + 273;
6    if (inKelvin < 0) {
7      abort();
8    }
9    return inKelvin;
10 }
11
12 #include "convert.h"
13 #include "verlib.h"
14
15 void verifyConvert_invalidInput() {
16   vt_enable_unreach_call();
17   int temperatureInCelsius = vt_anyInt();
18   vt_assume(temperatureInCelsius < -273);
19
20   convert(temperatureInCelsius);
21
22   reach_error();
23 }
```

Listing 2: Example project "convert.c" with and unit-verification and library methods in use

In Listing 2, we observe a verification harness for an example project `convert.c`. We initiate the method with the prefix `verify` to enable our runner to identify the verification task. In line 5, we activate the verification property `unreach_call()`, instructing the verifiers to identify `reach_error()` and flag the task as failed if it encounters this function. Considering the definition of the convert function in the "convert" project, the occurrence of `reach_error()` on line 12 should be prevented, as it should encounter the assertion on line 43 before reaching this point.

Runner. A significant component of our approach involves the implementation of a runner dedicated to unit verification tasks. As mentioned earlier, this runner can identify tasks within a specified directory. Moreover, it preprocesses these tasks using a C compiler, following user-provided instructions for project compilation. For the actual verification process, it uses CBMC [9] and CPAchecker [20] verifiers. The verification backend relies on the CoveritTeam web service to execute the verifications. The provided architecture facilitates the straightforward expansion of both the number of verifiers and the backend service itself. After successful (or failed) verification, the runner can output the result achieved by verification to the user.

Overview. Figure 1.1 illustrates our approach. The process begins with the user generating unit verification tasks using the methods available in our library. Subsequently, the user activates the runner, which identifies tasks, preprocesses them, and engages the verification backend to carry out the verification process. The verification backend yields results, which the runner presents to the user for review.

1.3 Results

To assess the effectiveness of our formal verification approach, we carried out a series of experiments on a real-world C project. Among the many open-source projects, we chose coreutils [15] because it is well-tested and approachable. We functions such as `cksum`, `cat`, `wc` and `echo`. The following subsection shows the verification task for `cksum`.

Verification Tasks for `cksum`

We applied verification tasks to the `cksum` function within the coreutils project. The verification tasks focused on assessing the correctness of the function under various scenarios.

Invalid Input Verification Task

We designed a verification task, shown in Listing 3, to evaluate the behaviour of `cksum` when provided with invalid input.

```
1  #include "cksum.h"
2  #include "verlib.h"
3
4  void verifyCksum_invalidInput() {
5    vt_enable_unreach_call();
6    FILE *emptyStream = fopen("empty.txt", "w");
7    uint_fast32_t *crc_out1 = vt_malloc(sizeof(unsigned int));
8    uintmax_t *length1 = vt_malloc(sizeof(uintmax_t));
9
10   int result1 = cksum_slice8(emptyStream, crc_out1, length1);
11
12   fclose(emptyStream);
13   free(crc_out1);
14   free(length1);
15
16   reach_error();
17 }
```

Listing 3: Invalid Input Verification Task for `cksum`

This verification task assesses how `cksum` handles invalid input scenarios. Specifically, it attempts to compute the checksum for an empty file (`empty.txt`) and ensures that the function reaches the error state (`reach_error()`) if any unexpected behaviour occurs.

1.4 Conclusions

While the findings and methodologies presented in this thesis may not revolutionize the entire software development landscape, they hold substantial promise as a significant step forward in formal verification.

The results that we received from applying the proposed approach to the coreutils project, explicitly targeting functions like `cksum`, `cat`, `wc`, and `echo`, demonstrate the practical viability and potential impact of the developed framework for software

verification. The presented verification tasks, exemplified by the one focused on `cksum` handling invalid input, showcase a robust and granular verification methodology.

The significance of this work extends to its ability to enhance the accessibility and efficiency of formal verification tools in the C language. The challenges tackled in this work, such as readability, scalability, and usability, are widespread in several existing tools. Hence, the C library, unit verification tasks, and task execution runner that were developed could serve as a model for future projects in this field.

Although the research may not have an immediate world-altering impact, it offers a valuable guide for developers and researchers working on formal verification in C projects. The results are broad and can be applied to various situations, providing knowledge and techniques to make the formal verification process more manageable and effective for multiple software projects.

This thesis is a guiding light demonstrating the proposed approach's feasibility and potential success. It is a testament to the ongoing efforts to improve software reliability and correctness through practical and accessible formal verification methods. This marks a significant contribution to the broader software engineering and verification landscape.

2 Related Work

Several research papers and technologies have tackled the topic in one way or another. This section reviews some of the more relevant work in software verification, bounded model-checking and property-based testing.

AWS Library. One notable contribution in this field is the proprietary work conducted at Amazon Web Services (AWS) [7]. Their methodology involves code-level model checking, utilising verification harnesses as unit-test-like tasks to verify specific functions within the source code. While sharing similarities with our approach, which also employs verification harnesses, it is essential to note that the AWS work is confined to projects within the AWS ecosystem and employs the CBMC tool [9]. This restriction prompts consideration of the generalisation and scalability of their approach compared to ours.

ACSL. One alternative to our formal verification approach is found in Frama-C's ASCL (ANSI/ISO C Specification Langage) [13]. ACSL provides similar functionalities but requires projects to be initiated with annotations in mind, potentially restricting its use for projects not initially designed for ACSL integration. This introduces a different set of considerations, emphasising the importance of integration flexibility in verification tools.

Theft. Theft is a property-based testing library for C, designed to generate input for stress-testing code and minimising failures to essential failing inputs [2]. While Theft concentrates solely on property-based testing, our approach enables the utilisation of various verification tools that adhere to SV-COMP [1] rules.

EvoSuite. EvoSuite [14] is another example of a framework that employs a similar approach. It facilitates automatic test case generation for Java classes. The key distinction is that EvoSuite is primarily designed for Java, whereas we operate in the context of C programming.

KLEE. KLEE is a symbolic execution tool capable of automatically generating tests achieving high coverage on complex programs [6]. The critical distinction between our approach and KLEE is that while KLEE emphasises the automatic generation of inputs to stress-test code, our approach involves writing verification harnesses and testing them with verifiers.

3 Background

3.1 Formal Verification

Formal verification ensures that software systems meet their desired specifications (properties) using rigorous mathematical reasoning. Formal verification can prove the correctness of systems for various properties, such as safety, security, or performance [11].

Model Checking. This work focuses on model checking as a formal verification method. Model checking is a systematic procedure that explores a system's possible states and transitions and checks whether they satisfy a given property. Model checking can handle complex systems with many components and interactions. [10]. For this, we use system models that define representations of a system under verification. For instance, consider a command to a model checker like `CHECK( init(main()), LTL(G ! call(reach_error())) )`. The first part of the command sets the initial state and tells the model checker that the system starts at the `main` function. The second part is the property we want to check, which in this example says that the model should never call the function `reach_error()`. Using these commands, we can verify that our system never reaches an error state, even for complex systems with many components and interactions.
In combination with the C programming language, the mentioned command defines properties using Linear Temporal Logic (LTL) [23]. This approach allows us to verify a range of properties, including but not limited to memory safety and buffer overflow.

Model checking has been successfully applied in various domains, including hardware design, protocol verification, and software verification. It has been used to detect errors in real-world systems that would have been difficult to find through traditional testing methods. [12]

3.2 Verifiers

We use tools called model-checkers or "verifiers" to perform model-checking on software systems. The model checker systematically explores all possible states or states reachable from an initial state, checking whether the specified properties hold at each state.

If certain states prove unreachable during this verification or counterexamples are detected, it may indicate potential issues affecting the system's adherence to the specified properties. These verifiers often provide valuable insights through features like witness graphs, illustrating the path that leads to a property violation, and counterexamples, demonstrating specific scenarios where the system deviates from the desired behaviour.
Many model checkers, including tools such as SPIN [18], NuSMV [8], BLAST [16] and PRISM [17], are widely used in the industry.
In the context of this study, we focus on two verifiers, namely C Bounded Model Checker (CBMC) [9] and CPAchecker. These verifiers have been chosen primarily due to their suitability for C projects, as well as their comprehensive documentation and robust performance. [20].

CPAchecker. CPAchecker is a software verification tool rooted in configurable program analysis. This approach facilitates the integration of model checking and program analysis within a single formalism. Upon execution, the CPAchecker conducts a reachability analysis, specifically checking whether a particular state that violates a specified criterion can potentially be reached. [5]

CBMC. CBMC, or the C Bounded Model Checker, is a tool for checking C and C++ programs. It can prove that, for computations of bounded depth, a C program exhibits no memory-safe errors (no buffer overflows, no invalid pointers, etc.), no undefined behaviours, and no failures of assertions in the code. CBMC explores the feasible paths of a program within the specified bounds, systematically analyzing all possible execution paths to find potential issues. When a violation of a safety property is found, CBMC can generate counterexamples, providing details about the specific conditions under which the violation occurs. [19]

3.3 ANTLR

For the recognition of user-written tasks, a C parser is crucial. ANother Tool for Language Recognition (ANTLR) [22] serves this purpose, generating parsers from language grammars.
ANTLR takes a language grammar as input and generates source code files for a parser in the target language. These grammars, exemplified in Listing 4 with a definition of relational expressions in C, serve as the blueprint for the parser. They provide a set of rules that define the syntax of a language.
In the Listing 4, we see a definition of relational expressions in C language. The rule says that the relational expressions start with `shiftExpression` (which is another definition in ANTLR), followed by zero or more groups (denoted by $\star$ sign). Each group consists of relational operators and a `shiftExpression`.

```
1  relationalExpression
2      :   shiftExpression (('<'|'>'|'<='|'>=') shiftExpression)*
3      ;
```

Listing 4: Example of ANTLR grammar for C

In practical terms, when ANTLR processes a user-written source code, it produces parse trees. A parse tree represents a structured data type where each node corresponds to a syntactic construct in a parsed language. This parse tree, in turn, enables structured text traversal and manipulation. [21].
ANTLR utilises a listener interface (or visitor) to conveniently define or add new functionalities without altering the generated source code [21]. This feature is particularly useful when there is a requirement to enhance the capabilities of a parser to address specific needs not covered by the original generated source code.

3.4 CoVeriTeam

A verification backend is essential for conducting verification using selected verifiers to facilitate the verification process. We have opted for the CoVeriTeam Service [4], which simplifies the verification procedure over a web service, eliminating concerns about the installation of tools and verifiers.

For verification, the CoVeriTeam Service relies on CoveriTeam [3]. It considers verification tools as verification actors and treats input and output as verification artifacts. The behaviour of CoveriTeam is specified in a .cvt file. This file accepts input from a YAML file representing our verifier and its options, as illustrated in Listing 5. In this listing, CBMC is defined by actor name, and we have specified options to generate witness graphs. In the archives section, we inform CoveriTeam about the verifier version we wish to use.

```
actor_name: cbmc
toolinfo_module: "https://gitlab.com/sosy-lab/software/benchexec/
-/raw/main/benchexec/tools/cbmc.py"
options: ['--graphml-witness', 'witness.graphml']
archives:
  - version: default
    location: "https://gitlab.com/sosy-lab/sv-comp/archives-2023/
    -/raw/main/2023/cbmc.zip"
```

Listing 5: An example verifier YAML file for CBMC

```
{
    "coveriteam_inputs": {
        "verifier_path": "cbmc.yml",
        "program_path": "test02.c",
        "specification_path": "unreach-call.prp",
        "verifier_version": "default",
        "data_model": "ILP32"
    },
    "cvt_program": "verifier.cvt",
    "working_directory": "coveriteam/examples",
}
```

Listing 6: An example JSON request file for CBMC

CoVeriTeam Service. One of the most straightforward methods to utilise the CoVeriTeam Service involves using HTTP and sending a POST request with a configurable JSON

file. In Listing 6, we present an example JSON request. Within the `coveriteam_inputs` section, we have specified options for our verifier. Initially, we indicate the verifier we want to use, followed by the program under verification, the type of property (specification) employed, the verifier version, and the data model. Additionally, we provide a .cvt file that CoVeriTeam should interpret.
The CoVeriTeam Service receives input through an HTTP POST request, conducts consistency checks on the provided input, constructs the command for CoVeriTeam, downloads the requested verifier, executes the specified program on the server, and returns the output artifacts. [4].

4 Contribution

As part of our contribution, we introduced three entities: a C library for software verification, Unit Verification Tasks, and a runner. The following subsections provide more detailed insights into each of these contributions. The entire library, along with the runner and documentation, is available at GitLab Repository.

4.1 C Library for Software Verification

This section delves into the design and implementation of the C library developed for software verification. The library serves as a foundational component of our formal verification approach, providing essential functionalities that contribute to the accessibility and integration flexibility of the overall solution.

Design

The C library is designed to be simple, versatile, and easily integrated into existing projects. It adheres to established SV-COMP [1] conventions and uses non-deterministic value generation to facilitate incorporation into various software verification tasks.

Library Methods

Nondeterministic Value Generation. The library provides functions such as `vt_anyInt()`, `vt_anyChar()`, and `vt_anyLong()` for generating non-deterministic values for integers, characters, and long integers, respectively. This functionality enhances the diversity of input scenarios during verification tasks.

String Selection with Non-deterministic Index. The `vt_oneOfStrings` function enables the selection of one string from a provided array using a non-deterministic index. This feature supports the creation of varied test cases, contributing to a more comprehensive verification process.

Custom Memory Allocation with Error Handling. The `vt_malloc` function serves as a custom memory allocation wrapper, ensuring successful memory allocation and providing error handling. This feature enhances the robustness of verification tasks by preventing memory-related issues.

Assumption and Assertion Functions. The library includes `vt_assume` and `vt_assumeNotNull` functions, which abort the process if specified conditions are not

met. Additionally, the `vt_assert` function terminates the process if an evaluation is incorrect, aiding in enforcing verification properties.

Distinguishing Features

Integration Flexibility:. The C library is designed to integrate seamlessly into existing C projects, requiring minimal adaptations. Its functions align with established conventions, allowing developers to incorporate formal verification tasks without significantly modifying their codebase.

Usability in Unit Verification Tasks:. The library's functionalities are tailored for unit verification tasks, providing developers with a convenient means of testing specific components of their projects. This granularity enhances the examination of individual functions, aiding in identifying potential issues.

Verification Property Enablement:. The library includes functions, such as `vt_enable_unreach_call`, `vt_enable_valid_memsafety`, and `vt_enable_no_overflow`, allowing users to specify verification properties for the runner. This feature provides flexibility in tailoring the verification process to project-specific requirements.

In summary, the C library for software verification constitutes a practical contribution that significantly enhances the accessibility and adaptability of our formal verification approach. Its functionalities, designed with simplicity and integration in mind, contribute to the effectiveness of unit verification tasks and promote the seamless integration of formal verification into C projects.

4.2 Unit Verification Tasks

We designed unit verification tasks to ensure the correctness of specific functions within our C library. These targeted tests check individual components of a project and their behaviour under different scenarios. Here, we present an illustrative example in the Listing 7 related to the `checkWheels` function. The `car.h` is defined in the Listing 8

Function Being Tested: `checkWheels`

The `checkWheels` function verifies the integrity of a `Car` object's wheels. The function ensures that the number of wheels is precisely four and that each wheel has the same diameter as the first wheel.
In this function, we verify that the `Car` object has exactly four wheels, and each wheel has the same diameter as the first wheel. Any deviation from this expected behaviour results in the abortion of the program, signalling a failure in the wheel-checking logic.

```
1 #include <assert.h>
2 #include <stddef.h>
3 #include "car.h"
```

```
4
5  void checkWheels(Car *car) {
6    size_t numberOfWheels = sizeof(car->wheels) / sizeof(*car->wheels);
7
8    if (numberOfWheels != 4) {
9      abort();
10   }
11
12   int expectedDiameter = car->wheels[0].diameter;
13   for (int i = 1; i < numberOfWheels; i++) {
14     Wheel *wheel = &car->wheels[i];
15     if (wheel->diameter != expectedDiameter) {
16       abort();
17     }
18   }
19 }
```

Listing 7: Function `checkWheels`

```
1  #ifndef CAR_H
2  #define CAR_H
3
4  #include <stddef.h>
5
6  typedef struct {
7    int diameter;
8  } Wheel;
9
10 typedef struct {
11   char color[20];
12   Wheel* wheels;
13 } Car;
14
15 void checkWheels(Car *car);
16
17 #endif // CAR_H
```

Listing 8: Header File: `car.h`

Example Unit Verification Task: `invalidNumberOfWheels`

Consider the scenario where we want to validate the behaviour of the `checkWheels` function in the context of a `Car` object. The task is designed to test the function's response when the number of wheels does not equal the expected value of four. This scenario is captured in the following unit verification task:

```
1  #include "car.h"
2  #include "verlib.h"
3
4  int validDiameter() {
5    int diameter = vt_anyInt();
6    vt_assume(diameter >= 0);
7    return diameter;
8  };
9
10 Wheel* validWheels(int length, int diameter) {
11    Wheel* wheels = (Wheel*) malloc(length * sizeof(Wheel));
12    for (int i = 0; i < length; i++) {
```

```
13      wheels[i].diameter = diameter;
14     }
15     return wheels;
16 }
17
18 char* validColor() {
19   const char *options[] = {"blue", "green"};
20   return vt_oneOfStrings(options, 2);
21 }
22
23 void verifyCheckWheels_invalidNumberOfWheels() {
24   vt_enable_unreach_call();
25   int wheelLength = vt_anyInt();
26   vt_assume(wheelLength > 0 && wheelLength != 4 && wheelLength < 10);
27   int wheelDiameter = validDiameter();
28   Wheel *wheels = validWheels(wheelLength, wheelDiameter);
29   Car car = { .wheels = wheels, .color = validColor() };
30
31   checkWheels(&car);
32
33   // code should not reach this but run into an assertion during
34   // the method call
35   reach_error();
36 }
```

Listing 9: Unit Verification Task for `checkWheels`

In this example, we create a `Car` object with a dynamically allocated array of wheels. The verification task sets up a scenario where the number of wheels is not equal to four and calls the `checkWheels` function. The expectation is that an assertion will be triggered during the function call, indicating a violation of the expected number of wheels.
This example illustrates how unit verification tasks are constructed to systematically test specific functionalities within the C library for software verification.

4.3 Runner

We created a runner to perform these tasks. The design is intended to facilitate further development and expansion with ease. In the following sections, we'll explain the architecture and optimisations.

Parser. The runner employs an ANTLR-based parser for C code to identify tasks efficiently. The parser scans the code for predefined function names within the specified directory, effectively pinpointing tasks. To do this, we enhance the ANTLR-generated parser by extending the listener interface and overriding the `enterFunctionDefinition` method of the ANTLR interface. This approach enables us to traverse the parsed tree and capture all the function names in the C file.

Preprocessing. Preprocessing is a pivotal component in the runner's functionality, essential for generating a single preprocessed file conducive to efficient verification. Verifiers like CBMC and CPAchecker perform best when dealing with a consolidated file rather than multiple ones.

The preprocessing step employs popular C compilers, such as Clang or GCC, leveraging Java's Process Builder to execute the compilation process. Users must provide the command line arguments for the chosen compiler, including those necessary for compiling the project to be verified. The runner then supplements these arguments with additional compilation options, such as library locations, and outputs everything into a single file for further processing. Because the project file to be verified is included in the compilation process, and those can have main functions declared, we need to tell the compiler which main function to use for verification (as it can have conflicts with verifiers). The runner does this by also appending `-Dmain=____main`.
As the last preprocessing step, the runner appends a new main function that calls the verification task. This ensures that only the verification task function will be considered during the verification process.

Verification Backend. To execute the verification process, the runner utilises the CoVeriTeam Service as the verification backend. As detailed in the background section, we initiate an HTTP POST request to the web service, providing a predefined task and selected verification options. These options encompass the verifier tool, verification property, and data model. The resulting output is then organised within the output folder, structured with the following hierarchy:

```
Root Directory
└── Output
    └── CBMC
        └── ProjectX
            ├── Task1
            │   ├── LOG.txt
            │   ├── execution_trace.xml
            │   ├── output.txt
            │   └── witness.graphml
            └── Task2
                ├── LOG.txt
                ├── execution_trace.xml
                ├── output.txt
                └── witness.graphml
```

The verification backend interface is also designed to be decoupled, allowing for the easy implementation of new backends in addition to CoVeriTeam.

Verification View. To showcase the verification output, we designed a user-friendly verification view. This view presents real-time information on the ongoing verification tasks, culminating in the display of the final verification result details at the conclusion. Using the option `debug`, we see more detailed information about the verification process.

```
1 Using verifier: CPACHECKER with data model: ILP32.
2 verifyConvert_invalidInput               SUCCESS  (1/2)
3 verifyCheckWheels_invalidNumberOfWheels  SUCCESS  (2/2)
4 Verification finished successfully. See 'output/cpachecker' for more
5 information.
```

Listing 10: Example output of a runner

In the presented command line output in Listing 10, a verification process is conducted using the CPAchecker verifier with a specified data model (ILP32). The program begins by establishing a temporary directory.
It then explores the directory `./uvl` to identify unit verification tasks, revealing the discovery of two such tasks. The subsequent execution involves verifying a task named `verifyConvert_invalidInput`, completing successfully after 9.0 seconds of processing time (SUCCESS 1/2).
A subsequent task, `verifyCheckWheels_invalidNumberOfWheels`, is then initiated, and its verification process is detailed through debug information. This task concludes successfully after 92.9 seconds of processing time (SUCCESS 2/2).
The verification process as a whole is reported as successful, directing the user to consult the 'output/cpachecker' directory for a more comprehensive overview, as mentioned in the previous subsection.

Command-line Interpreter. We also have a command line interpreter that takes user input and executes wanted behaviour.

```
./start.sh --verify uvl/ --debug -- gcc -I lib/headers uvl/car.c
```

Listing 11: Example command-line options of a runner

As demonstrated in Listing 11, initiating the runner involves executing a shell script named `./start.sh` with specific command-line options. The `-verify` option directs the runner to commence the verification process, specifying the directory `uvl/` as the location to search for tasks.
Additionally, an optional `-debug` flag is available to instruct the runner to generate more detailed output, as illustrated in Listing 10. Including the – option is crucial because it provides the runner with essential information on how to compile the program undergoing verification.

Listing 12 shows us all the available command-line options with detailed descriptions.

```
- `-d, --debug`: Show debug information.
- `-h, --help`: Print available options.
- `-m, --data-model <arg>`: Choose a data model to be used.
Valid choices: [ILP32, LP64]. Default: ILP32.
- `-p, --parser <arg>`: Path to the input file to be parsed.
- `-t, --tool <arg>`: Choose a verifier to be used.
Valid choices: [cpachecker, cbmc]. Default: cpachecker.
- `-v, --verify <arg>`: Path to the directory containing verification
tasks.

Use `--` to pass the following compile commands to the runner:
compiler [gcc, clang..], header files location with '-I', and project
source files.
```

Listing 12: Available runner command-line options

5 Evaluation

To evaluate the efficiency of our software verification approach, we performed verification tasks on a real-world project, specifically the coreutils project [15]. Coreutils was selected for its reputation as a thoroughly tested project with comprehensive documentation. We designed tests for four critical functions within the project: `cat`, `echo`, `wc`, and `cksum`.

5.1 Experimental Setup

The experiments were carried out on a system running Windows 11 with the Windows Subsystem for Linux (WSL) configuration. The hardware setup included an Intel i7 central processing unit (CPU) with 2.30 GHz, and the system used 16 GB of RAM.
The following resource limits were enforced during the experiments:

```
resourcelimits:
  memlimit: "15 GB"
  timelimit: "2 min"
  cpuCores: "2"
```

Experimental Tools

The benchmark tasks used in the experiments are detailed in the provided GitLab repository at the start of Section 4. This repository contains comprehensive information on the benchmark tasks, facilitating the replication of the experiments.

The verification tools employed in this study were CBMC and CPAchecker. The versions of these tools used during the experiments are crucial for result reproducibility. We used the following versions:

- CBMC
  URL: https://gitlab.com/sosy-lab/sv-comp/archives-2023/-/raw/main/2023/cbmc.zip
- CPAchecker
  DOI: 10.5281/zenodo.10203297

The Coveriteam service was utilised to run the experiments and measure data. This service facilitated the execution of the experiments and ensured consistent and

controlled conditions. The complete steps to reproduce the results are documented in the GitLab repository mentioned earlier.

5.2 Experimental Results

Verification Tool	Function Tested	Property Used	Verification Result
CBMC	wc_lines	no_overflow	Unknown
CPAchecker	wc_lines	no_overflow	Unknown
CBMC	hextobin	valid_memsafety	Successful
CPAchecker	hextobin	valid_memsafety	Successful
CBMC	cksum_slice8	enable_unreach_call	Failed
CPAchecker	cksum_slice8	enable_unreach_call	Failed
CBMC	simple_cat	valid_memsafety	Failed
CPAchecker	simple_cat	valid_memsafety	Failed

Table 5.1: Summary of verification outcomes, functions tested, properties used, and verification results

Quantitative Analysis

Table 5.1 summarises the quantitative results of our software verification experiments on the coreutils project using CBMC and CPAchecker. The verification tasks were performed on four functions: `wc_lines`, `hextobin`, `cksum_slice8`, and `simple_cat`. The properties checked included `no_overflow`, `valid_memsafety`, and `enable_unreach_call`. The results are categorised as either "Successful" or "Failed." If the tool has not been able to compute the verification, it returns "Unknown" output.

Qualitative Analysis

In this section, we present code snippets and discuss specific examples to analyse the behaviour of verifiers during the verification tasks qualitatively. Our runner successfully identified and executed all the presented tasks accurately using both mentioned verifiers.

Example: Verification of `hextobin` Function

Consider verifying the `hextobin` function of the `echo.c` using CBMC or CPAchecker with the property `no_overflow`. The following code snippet represents the verification task for the mentioned function:

```
1 // verlib.h
2 #include "verlib.h"
3
4 void verify_hextobin_noOverflow() {
5     // Enable no overflow verification
```

```
6       vt_enable_no_overflow();
7       // Assign non-deterministic value to c
8       char c = vt_anyChar();
9
10      // Call function under verification
11      char result = hextobin(c);
12  }
```

Moreover, the corresponding function being verified:

```
1   // verlib.c
2   static int hextobin(unsigned char c) {
3       switch (c) {
4           default: return c - '0';
5           case 'a': case 'A': return 10;
6           case 'b': case 'B': return 11;
7           case 'c': case 'C': return 12;
8           case 'd': case 'D': return 13;
9           case 'e': case 'E': return 14;
10          case 'f': case 'F': return 15;
11      }
12  }
```

In this example, we enable the `no_overflow` verification property using the `vt_enable_no_overflow()` function provided by the verification library. We then assign a non-deterministic character value to the variable `c` using `vt_anyChar()` and call the `hextobin` function. The verifier will verify whether the `no_overflow` property holds during the execution of this verification task.

The `hextobin` function converts a hexadecimal character `c` to its corresponding decimal value. The switch statement handles various cases, and the function involves simple arithmetic operations.
The `verify_hextobin_noOverflow` function tests the `hextobin` function under the `no_overflow` property. In this context, the `no_overflow` property asserts that the arithmetic operations within `hextobin` do not result in an integer overflow. CBMC will analyse the function and verify whether the specified property holds for all possible inputs.

Unknown Outcome for `wc.c` lines

It is important to note that the verification outcome for the `wc.c` lines task is marked as "Unknown." This is because both CBMC and CPAchecker could not complete the verification process due to the presence of the `_Static_assert` statement within the code. The `_Static_assert` statement introduces static assertions that are checked at compile-time. While this is a valuable mechanism for ensuring certain conditions at compile-time, it can make it challenging for runtime verification tools to analyse the code thoroughly.

5.3 Threats to Validity

External Validity

An external validity threat arises from whether the results obtained from our benchmark tasks can be generalised to real-world programs. Our experiment utilised a set of well-established software verification problems based on the coreutils project. While this collection is recognised and relevant in the verification community, it may not comprehensively represent the diversity of real-world programs. That is why the external validity of our results is limited to the types of programs and verification challenges present in our chosen benchmark.
However, it is essential to note that the coreutils project covers a range of functionalities and is widely used. We aimed to mitigate this threat by selecting a project with a broad scope, but the specific characteristics of other programs may introduce variations in verification behaviour.

Internal Validity

Internal validity concerns the accuracy of our experimental setup and whether the observed results are affected by confounding factors. One potential internal validity threat is related to the presence of the `_Static_assert` statement in the `wc.c` lines verification task. This static assertion is checked at compile-time and can lead to early termination of the verification process by CBMC and CPAchecker. As a result, the verification outcome for this particular task is marked as "Unknown."
While the `_Static_assert` statement ensures certain conditions during compilation, its presence introduces a limitation in the runtime verification process. The inability to complete the verification for this task may affect the overall interpretation of our results.
Additionally, implementing our verification algorithms may contain bugs that could impact the analysis of specific program paths. While we have not observed false proofs during our experiments, the presence of bugs could affect the accuracy of our results. However, the absence of false proofs suggests a reasonable level of confidence in the validity of our outcomes.
In summary, acknowledging these threats to external and internal validity is crucial for interpreting the accuracy of our software verification experiment results.

6 Future Work

In this section, we evaluate potential future research and development of our implementation to optimise and expand upon the work done in the scope of this thesis.

6.1 Optimising Existing Components

The overall architecture of the runner is well-optimised for future development; however, certain areas might benefit from additional optimisation. Specifically, aspects of the error-handling mechanism may need further adjustments, which could not be completed due to time constraints.

6.2 Adding New Components

Verifiers. An effective strategy for enhancing the functionalities of this project is to introduce new components to the existing framework. Currently, the runner supports only two verifiers, as mentioned in the preceding sections. However, incorporating additional verifiers, currently maintained by the CoVeriTeam service, should be relatively straightforward.

Library Expansion. Presently, the library incorporates a limited set of implemented functions. It would be advantageous for future development to extend and introduce additional functionalities as necessary. An easily implementable enhancement would involve expanding the options for verification properties. Currently, the process consists of specifying function names and adding .prp files, but additional options could be incorporated for greater flexibility.

Verification Backend. As previously discussed, the architecture should be sufficiently decoupled to handle the integration of a new verification backend. This could be a locally running system or an entirely separate web service.

Evaluation on more C projects. The current assessment is confined to the coreutils project, potentially needing more representation of the diversity found in real-world C programs. A more comprehensive evaluation, encompassing diverse types of C projects and featuring more complex and insightful verification tasks, could be beneficial.

7 Conclusion

This thesis presents an approach to formal verification of C programs using a dedicated C library, unit verification tasks, and a task execution runner. Our approach aims to overcome the readability, scalability, and usability challenges affecting existing verification tools. We have demonstrated the practical application and effectiveness of our approach by writing verification tasks using our library on the coreutils project, verifying functions such as `cksum`, `cat`, `wc`, and `echo`. Our results show that our runner can be used as a viable alternative to existing formal verification frameworks, as our approach also offers integration flexibility, allowing developers to easily incorporate formal verification into their C projects without significant modifications.
Our contribution is significant for software verification, as it provides a simple, versatile, and accessible solution for ensuring the correctness and reliability of C programs. Our approach can be applied to various domains and scenarios and serve as a valuable guide for developers and researchers working on formal verification in C. Our work also opens up new possibilities for future research and development, such as optimising existing components, adding new features, and expanding the scope of verification tasks and we believe is a step forward in advancing the state-of-the-art in formal verification, and we hope that it will inspire further innovation and improvement in this field.

Bibliography

[1] 12th competition on software verification (sv-comp 2023).

[2] theft: property-based testing for c.

[3] D. Beyer and S. Kanav. Coveriteam: On-demand composition of cooperative verification systems. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 561–579, Cham, 2022. Springer International Publishing.

[4] D. Beyer, S. Kanav, and H. Wachowitz. Coveriteam service: Verification as a service. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 21–25, 2023.

[5] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[7] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Code-level model checking in the software development workflow at amazon web services. *Software: Practice and Experience*, 51(4):772–797, 2021.

[8] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *STTT*, 2:410–425, 03 2000.

[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[10] E. M. Clarke. Model checking. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[11] E. M. Clarke, T. A. Henzinger, and H. Veith. *Introduction to Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018.

[12] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, dec 1996.

[13] Frama-C. ACSL: ANSI/ISO C Specification Langage, ongoing.

[14] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Free Software Foundation. Gnu core utilities, ongoing.

[16] T. A. Henzinger, R. Jhala, and R. Majumdar. The blast software verification system. In P. Godefroid, editor, *Model Checking Software*, pages 25–26, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[17] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[18] G. Holzmann, E. Najm, and A. Serhrouchni. Spin model checking: An introduction. *STTT*, 2:321–327, 03 2000.

[19] D. Kroening and M. Tautschnig. Cbmc – c bounded model checker. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[20] S. Lab. CPAChecker: A Framework for Configurable Software Verification, ongoing.

[21] T. Parr. Antlr repository, ongoing.

[22] T. Parr. Antlr website, ongoing.

[23] K. Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011.