Understanding the landscape of WP by applying it to a C semantic versioner

gab8un

September 2024

Contents

Honor Code

On my honor as a student I have neither given nor received any unauthorized aid on this assignment

1 Introduction

When writing software, it is often important to include, as part of the development lifecycle, some mechanism to provide information on whether or not the code is working as intended. Including such verification techniques helps cover software edge cases, allows for more confidence that the software works as intended after it is modified, and helps reduce the number of bugs in the resulting code. While there are many ways of including such verification processes in software development, this paper will focus on the use of Frama-C (https://frama-c.com/), an "open-source extensible and collaborative platform dedicated to source-code analysis of C software". In particular, it will explore the use of the WP plug-in to dispatch proofs of software specifications written in the ANSI/ISO C Specification Language (ACSL).

The goals of this paper are to document the experience of using WP to provide guarantees of C programs, and in particular the application of WP to a semantic versioning library written in C (https://github.com/h2non/semver.c). The hope is to gain insight into the capabilities and drawbacks of using such a verification system, as well as to provide recommendations both for developers looking to write specifications for their C code, and to developers of the automated provers.

1.1 ACSL, Frama-C, WP, and Automatic provers

The verification techniques described in this paper focus on the use of four main technologies: ACSL, Frama-C, WP, and Automatic provers (e.g. Alt-Ergo & Z3).

The first that is important to understand is the specification language ASCL. A specification language allows for the writing of function contracts which establish preconditions and postconditions of the function. ACSL is a formal language for writing out such function specifications. ACSL also provides many other tools for writing such specifications, such as being able to write inductive and axiomatic definitions to express more complex contracts. The following is the example of a function contract that specifies the pre and post conditions for a function that swaps the values of two locations in memory.

```
/*@
   requires \valid(a);
   requires \valid(b);

   ensures *a == \old(*b) && *b == \old(*a);
*/
```

In this simple example, the pre-condition is that the inputs a and b are pointers to valid memory locations, and the post-condition is that the values pointed to by and b are swapped.

The tool used to actually dispatch proofs for these is the Weakest Precondition (WP) plugin for Frama-C. Frama-C is a platform dedicated to the analysis of C programs, with many different plugins providing static and dy-

namic analyses. WP is a static analysis plugin for Frama-C which aims to prove function contracts using the Weakest Precondition Calculus("Frama-C - Framework for Modular Analysis of C programs", 2024). WP generates "verification conditions" from the C code and ACSL. These verification conditions need to be verified by an external solver in order for the proof to be carried out. This project focuses on automatic SMT solvers, specifically Alt-Ergo and Z3, however there are many more automatic solvers out there. There are also manual solvers or "proof assistants" such as coq, which allows for the proof of more complex properties that the automated solvers may not be able to handle by allowing the developers to write the proofs of the verification conditions(Blanchard, 2020). This paper does not explore the use of any proof assistants.

Sometimes the solvers need additional information about the code in order to dispatch a proof. This is mainly the case when it comes to annotation loops to provide information and with assert statements to assert a property at a certain point in code. The following are examples of providing information so WP can reason about how a loop affects a variable:

```
/*@
  loop invariant 0 <= i <= 10;
  loop assigns i;
  loop variant 10 - i;
*/
for (int i = 0; i < 10; i++) {}
//@assert i == 10;
```

In the above example, the loop invariant provides a condition that must be true before and after every iteration of the loop so that the prover can conclude that if the loop has terminated, the value of i must be 10. As it currently stands, WP with Alt-Ergo as the SMT solver is unable to prove the assert condition without the loop invariant clause.

This paper will not go much further into how all of these tools work together to dispatch proofs, however, a high level overview of how they interact and what tool is responsible for what will help understand the key takeaways from this case study.

2 Motivation

Currently, the leading practice in industry software verification comes in the form of running automated test suites (V Garousi, 2013). These test suites compare the code under test to expected outputs for a given set of inputs. These types of tests have the benefit of being able to verify that the code has the correct behavior for the given set of inputs that it is tested on. These types of tests are also generally good at regression testing, ensuring that when the code is modified or refactored, it does not break any existing implementations.

One of the major pitfalls of automated test suites is that they can only run tests for a limited number of inputs. For example, a developer may write tests that ensure the software produces the expected outputs $\{e_1, e_2, ..., e_n\}$ for inputs $i_1, i_2, ..., i_n$, however, there is no guarantee that the software will produce

correct output e_{n+1} for i_{n+1}. The only way that automated tests can provide the guarantee that the software will produce the correct output for any given input, is if the test suite covers the entire input space. Oftentimes this is infeasible, for example a function that has a singular signed, 32 bit integer input has 2^{32} different possible inputs to test, and some input types, like arrays or strings can have an unbounded number of inputs (or the number can differ based on the hardware).

As a contrast for automated test suites, formal verification methods can verify that a program works for every input within the input space. The input space is defined in ASCL by the pre-conditions, and when a proof is being conducted, WP will ensure that the pre-conditions are met. If the pre-conditions are met, and the function contract is proven, then we can ensure the correctness of the post condition.

While this is a very clear advantage that formal proof has over automated testing, gaining the benefit of formal proofs is far from trivial. Even with the use of automated provers discussed in this paper, there is significant effort that goes into creating the function contracts. This study aims to gain a better understanding of that effort, as well as provide recommendations for how it can be done better and more efficiently. The hope is that by lowering the cost of implementing formal verification methods, a higher quality of software can be created at lower costs.

For this project, I also wanted to see how these techniques can be applied to a more real world problem. The case for this report is applying WP to a Semantic Versioning Library in C. This project was chosen because it provides practical value, as semantic versioning is something that is widely used in the software industry. The project also seemed to be of reasonable complexity, not too trivial as to not be able to gain any insight, and not too complex so that I could conduct a reasonable analysis on the project in the time given.

3 Experience

When starting to write out program specifications to be proved by WP, the first step was learning and becoming familiar with ASCL. As mentioned previously, ASCL is a specification language for C which allows developers to write out function contracts which can later be proved. ASCL draws on a lot from first order logic in order to write the specifications. For some background information on my experience prior to taking on this project, I worked on this during my 6th semester of my CS undergraduate degree. At that time, I had already taken a few introductory courses in discrete math, which covered first order logic, as well as formal program proofs. This leads to the first point that learning ASCL has a potential learning curve.

Learning ASCL is, at its core, is learning another programming (or rather specification) language. If you have never been exposed to another specification language, this can be quite a daunting task. Due to formal verification techniques (and the use of Frama-C) being somewhat niche, there

were also not many resources available to learn this language. One of the main resources I leveraged was the WP tutorial by Allan Blanchard (https://github.com/AllanBlanchard/tutoriel_wp). While very useful, this tutorial only covered the basics of using ASCL with WP, and often did not go into detail on how to use it in more complex scenarios.

Along with learning the syntax and capabilities of both ASCL and WP, there is the added difficulty of thinking in the way of specification. As programmers, we often think of writing out our code imperatively, writing out how the code should accomplish something rather than writing out what it should accomplish. Even with a full understanding of ASCL syntax, there is still a lot of mental overhead when attempting to write out a function specification in a logically sound way.

One good example of the difference between declarative and imperative thinking is sorting a list. When writing out the code to sort a list, we may write something like the following bubble sort implementation:

```
void bubble_sort(int a[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (a[j] > a[j+1]) {
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

If we consider how this affects the array, we can conclude that the code is sorting the array (progressively moving larger values towards the end of the array), however, there is nothing in this code that actually says that the array is sorted. For the same function, the specification may look something like

```
/*@ requires \valid(a + (0..n-1));

requires n >=0;

ensures \forall integer i, j; 0 <= i <= j < n ==> a[i] <= a[j];

Assigns a[0..n-1];
*/
```

This example demonstrates two things, the first, as mentioned, is that this code writes out what it means for a list to be sorted. That is, for any pair of integers i, j, if i ¡= j then a[i] should be less than or equal to a[j]. This example also demonstrates how easy it is for specifications to be too weak. It is true that our example code could be proved to meet this specification, however, it is also true that an incorrect* program could also be proved to meet this specification. For example, the function:

```
void sort(int* a, int n) {
    for (int i = 0; i < n; i++) {
```

```
        a[i] = 0;
    }
}
```

If we consider how this function acts in relation to the specification, it is true that for all pairs i, j that a[i] ¡= a[j]. This indicates that our specification is imprecise, and not actually proving what we wish to prove. A more constrained function contract may look something like:

```
/*@ requires \valid(a + (0..n-1));

requires n >=0;

ensures \forall integer i, j; 0 <= i <= j < n ==> a[i] <= a[j];
ensures \forall integer i; 0 <= i < n ==>
    \exists integer j; 0 <= j < n && a[i] == \old(a[j]);

assigns a[0..n-1];
*/
```

Which provides finer constraints on the problem (if a number existed in the input array it should still exist in the output array), meaning that the previous example would no longer be proven. However, this still does not ensure that function preserves the number of occurrences of each number in the array. For example a function that transformed an input array of [3, 2, 1, 2, 4] to [1, 1, 2, 3, 4] could still potentially be proven by this specification as all numbers are in non-decreasing order and all the numbers in the first array exist in the second. Allan Blanchard provides an example of an inductive definition for a permutation of an array, which we can use to ensure that the output array is a permutation of the input array providing a complete specification for sorting (Blanchard, 2020).

```
/*@
    predicate swap_in_array{L1,L2}(
        int* a, integer b, integer e, integer i, integer j
    ) =
        b <= i < e && b <= j < e &&
        \at(a[i], L1) == \at(a[j], L2) &&
        \at(a[j], L1) == \at(a[i], L2) &&
        \forall integer k; b <= k < e && k != j && k != i ==>
            \at(a[k], L1) == \at(a[k], L2);

    inductive permutation{L1,L2}(int* a, integer b, integer e){
    case reflexive{L1}:
        \forall int* a, integer b,e ; permutation{L1,L1}(a, b, e);
    case swap{L1,L2}:
        \forall int* a, integer b,e,i,j ;
            swap_in_array{L1,L2}(a,b,e,i,j)
                ==> permutation{L1,L2}(a, b, e);
    case transitive{L1,L2,L3}:
        \forall int* a, integer b,e ;
            permutation{L1,L2}(a, b, e) && permutation{L2,L3}(a, b,
     e)
                ==> permutation{L1,L3}(a, b, e);
    }
*/
```

There are a few main takeaways from this example. The first is that the difficulty from writing good specifications comes from two different angles: being able to express the requirements within the specification language, and ensuring that the specification accurately reflects what you are trying to prove. In this example, it is rather easy to overlook the complexities of writing a complete specification, and accept that the naive implementation is acceptable. Furthermore, there is no real indication that something is wrong when you have an incomplete specification. If the specification or code is incorrect, then there will be a signal that something is wrong from Frama-C failing to prove the contract. When a specification is imprecise, it is likely that the contract will be proved, but that proof does not provide much value.

Similarly, with this example, a code snipped that is relatively simple, ends up having a rather complex function contract. Even after determining that the previous contracts were incomplete, it is far from trivial to correct that issue. Writing the permutation definition requires understanding defining recursive predicates in ASCL, as well as using inductive reasoning to construct the predicates in a way that accurately reflects the goals.

4 Examples

This section will examine examples related specifically to the semver project in which I attempted to apply WP to verify the correctness of the software. While working through writing contracts for the project, I found that despite attempting to scope out a project with reasonable complexity, it was still too difficult to write suitable contracts for every single function within the project. For this reason, I extracted certain functions to examine individually to examine how WP might be applied to parts of the project as opposed to the project as a whole. The code can be found at https://github.com/GarrettBurroughs/semver-wp. The main file is in semver.c, and the analyzed snippets can be seen in the isolated folder.

4.1 Version Bumping

The project had various functions to bump versions within a struct. The code for these was exceptionally trivial, as were the function contracts. Here is the example of one such function

```
typedef struct semver_version_s {
  int major;
  int minor;
  int patch;
  char * metadata;
  char * prerelease;
} semver_t;

/**
 * Version bump helpers
 */
```

```
12
13 /*@
14     requires \valid(x);
15     requires x->major >= 0;
16     requires x->minor >= 0;
17     requires x->patch >= 0;
18     ensures x->major == \old(x->major) + 1;
19   */
20 void
21 semver_bump (semver_t *x) {
22   x->major++;
23 }
```

This is an example where the specification follows the implementation pretty closely, and both are similarly trivial. The importance of writing a contract for a function like this is to allow WP to reason about it if it is used within another funciton. This contract also provides additional information about the precondition of the function, that isn't apparent in just the implementation, and can constrain the use of the function. While the function might "work" if a version number is negative, adding the positive version invariant might allow us to catch a bug where version numbers end up being negative. This also demonstrates how ASCL is able to support structure datatypes.

4.2 Contains

The project defined various helper methods, one of which was a contains method to determine if an array contained an element. This is a good example of code where the specification varies from the implementation.

```
1    #include <stddef.h>
2
3  /*@
4     requires \valid_read(matrix + (0 .. len - 1));
5     assigns \nothing;
6     behavior in:
7         assumes ! \forall size_t i ; 0 <= i < len ==> matrix[i] !=
      c;
8         ensures \result == 1;
9     behavior out:
10        assumes \forall size_t i ; 0 <= i < len ==> matrix[i] != c;
11        ensures \result == 0;
12    disjoint behaviors;
13    complete behaviors;
14   */
15 static int contains(const char c, const char *matrix, size_t len) {
16   /*@
17         loop invariant 0 <= x <= len;
18         loop invariant \forall size_t i; 0 <= i < x ==> matrix[i]
      != c;
19         loop assigns x;
20         loop variant len - x;
21    */
22   for (size_t x = 0; x < len; x++)
23     if (matrix[x] == c)
24       return 1;
```

```
  return 0;
}
```

This example shows how despite the function having a relatively simple implementation, the function contract and annotations needed to prove the function can still be quite lengthy. Similarly, there was another interesting result of writing this function contract. While initially writing it, I expressed the first condition as:

```
behavior in:
    assumes \exists size_t i ; 0 <= i < len ==> matrix[i] == c;
    ensures \result == 1;
```

Despite this being logically equivalent to the above statement, the version of WP I used failed to dispatch a proof for this statement. This demonstrates that depending on how the contracts are expressed can influence WPs ability to dispatch a proof.

4.3 Strcut

The following example is an unsuccessful attempt to write a contract for a function, and some pitfalls that come along with it.

```
#include<stddef.h>
#include "../framac/string.h"

static const int MAX_SAFE_INT = (int) -1 >> 1;
/*
 * Remove [begin:len-begin] from str by moving len data from begin+
    len to begin.
 * If len is negative cut out to the end of the string.
 */



/*@
    requires \valid(str + (0..strlen(str)));
    requires valid_string_s: valid_read_string(str);
    requires 0 < begin < strlen(str);
    assigns \nothing;
    behavior out_of_bounds:
        assumes strlen(str) < 0 || strlen(str) > MAX_SAFE_INT;
        ensures \result == -1;
    behavior negative:
        assumes len < 0 && 0 < strlen(str) < MAX_SAFE_INT;
        ensures \result == (strlen(str) - begin + 1);
    behavior past_str:
        assumes len >= 0 && (begin + len) > strlen(str) &&
            0 < strlen(str) < MAX_SAFE_INT;
        ensures \result == strlen(str) - begin;
    behavior regular:
        assumes len >= 0 && (begin + len) <= strlen(str) &&
            0 < strlen(str) < MAX_SAFE_INT;
        ensures \result == len;

    disjoint behaviors;
```

```
    complete behaviors;
*/
static int strcut (char *str, int begin, int len) {
  size_t l;
  l = strlen(str);

  if((int)l < 0 || (int)l > MAX_SAFE_INT) return -1;

  len = l - begin + 1;
  if (begin + len > (int)l) len = l - begin;
  memmove(str + begin, str + begin + len, l - len + 1 - begin);

  return len;
}
```

When asking WP to discharge proofs for the above statements, it concludes that all are valid, however, while true, it does not actually mean that the contract is proving what we expect the function to do. While a subtle error, the line:

```
static const int MAX_SAFE_INT = (int) -1 >> 1;
```

is actually supposed to be:

```
static const int MAX_SAFE_INT = (unsigned int) -1 >> 1;
```

Without properly typecasting -1 to an unsigned int, MAX_SAFE_INT is set to -1 instead of the properly computed max safe integer value. This leads to all of the pre-conditions to be always false, and so therefore since it is impossible to achieve the precondition, any statement that follows can be assumed to be true. This lead me to thinking that the function contract was properly written without having any signal that something was wrong.

This contract also demonstrates a function that relies on an external function. Luckily, Frama-C provides function contracts for some of the C standard library, however it is also important that the pre-conditions of that function are satisfied in order to discharge a proof.

4.4 String Helpers

The following code samples are axiomatic definitions that were written to aid in the proof of other functions.

```
/*@
     axiomatic IntToAscii {
        logic char* itoa{L}(integer n);

        axiom valid_char{L}: \forall integer i ; 0 <= i < 10 ==>
            \valid(itoa(i));

        axiom to_ascii{L}: \forall integer i ; 0 <= i < 10 ==>
            *itoa(i) == i - 48;
     }
*/

/*@
```

```
   predicate str_concat(char* result, char* a, char* b) =
    \forall size_t i ; 0 <= i < strlen(result) ==> i < strlen(a) ?
    result[i] == a[i] : result[i] == b[i - strlen(a)];
  */

/*@
   predicate str_concat_char(char* result, char* a, char b, char* c
    ) =
    \forall size_t i ; 0 <= i < strlen(result) ==>
        i < strlen(a) ? result[i] == a[i] :
            i == strlen(a) ?
                result[i] == b : result[i] == c[(i - strlen(a) + 1)
    ];
  */
```

One approach that was taken when writing these contracts and definitions was to provde a function contract for the concat_num function that is assumed to be true. Because it is just a function header, Frama-C will take the contract at face value without attempting a proof. This can be a useful way to segment sections of the program proof by assuming everything up to a certain point is true and then going from there.

The other definitions shown here are examples of predicates that can be useful in writing other function contracts. I found that strings were particularly difficult to deal with, due to operations being less well defined for strings as opposed to something like an integer. Writing out predicates for these operations allowed for the function contracts written to be more concise.

4.5 comparison

The following is a contract for a function that compares two semantic versions.

```
/*@
    behavior major_gt:
        assumes x.major > y.major;
        ensures \result == 1;

    behavior major_lt:
        assumes x.major < y.major;
        ensures \result == -1;

    behavior minor_gt:
        assumes x.major == y.major && x.minor > y.minor;
        ensures \result == 1;

    behavior minor_lt:
        assumes x.major == y.major && x.minor < y.minor;
        ensures \result == -1;

    behavior patch_gt:
        assumes x.major == y.major && x.minor == y.minor && x.patch
     > y.patch;
        ensures \result == 1;

    behavior patch_lt:
```

```
23          assumes x.major == y.major && x.minor == y.minor && x.patch
        < y.patch;
24          ensures \result == -1;
25
26      behavior eq:
27          assumes x.major == y.major && x.minor == y.minor && x.patch
        == y.patch;
28          ensures \result == 0;
29
30      disjoint behaviors;
31      complete behaviors;
32  */
33  int
34  semver_compare_version (semver_t x, semver_t y) {
35    int res;
36
37    if ((res = binary_comparison(x.major, y.major)) == 0) {
38      if ((res = binary_comparison(x.minor, y.minor)) == 0) {
39        return binary_comparison(x.patch, y.patch);
40      }
41    }
42
43    return res;
44  }
```

This example shows the potential verbosity of some function contracts. Despite the actual implementation of the function being rather straight forward, the contract has to reason about a large number of scenarios in order to provide a useful specification for the function. This is potentially a limitation of the specification itself, and there could potentially be better ways, such as nested behaviors to allow for more concise contracts.

5 Lessons Learned and Recommendations

This section will outline the main takeaways I have gained from working through this project as well as documenting some suggestions I have for improving the development experience when it comes to writing and proving function specifications. Some of the suggestions are things that can be taken on by the developer while writing their code and specifications, while others are suggestions for tools and functionality that don't exist but would be nice to have.

5.1 Writing code with verification in mind

One thing to note about this project is that I had taken someone else's existing code and attempted to add function specifications on top of it. One of the main issues I ran into was that this code was not written with verification in mind. There were some very long functions with many different branches, which would require very extensive definitions to provide any useful information about the function. I often found myself splitting up the written code to make writing the contracts easier.

While splitting up functions is generally advisable when it comes to good software development practices, this can be especially crucial when writing function contracts. The shorter the function, the less has to be reasoned about when writing the function contract.

5.2 Axiom and predicate packages

Similar to the function contracts in frama-C for C standard library functions, it would be useful if packages that contained common predicates and axioms would be useful. Having a more complex set of tools at your disposal while writing function contracts would allow developers to focus on the higher level requirements of the contracts without having to reason through more of the lower level details of the contract. One good example of such a predicate would be the permutation predicate documented in the Experience section. Such a predicate is commonly used whenever dealing with in-place array mutations to ensure that the contents of the array has not changed.

5.3 Better learning support

Through my experience there are not many resources available to learn about ASCL. Additionally, it seems like most of the examples given in the resources that were available were solution based examples. In these examples, a capability of ASCL was introduced, and then ways to apply it were shown, rather than taking the approach of being given a problem, and then the tools to solve it. Because of this, I found it was much more difficult to apply ASCL function contracts to a real world system as opposed to a set of functions that were designed to have contracts written for them.

5.4 Macro System

As seen in the semver_compare_version example, sometimes the function contracts end up being very verbose. There are also many scenarios where there is a fair amount of code shared between function contracts. One thing I think would make writing function contracts easier is to have some ability to pragmatically generate them through some sort of macro system.

5.5 Contract Auditing

As mentioned many times throughout this paper, one of the larger difficulties faced when writing the function contracts was reasoning through whether or not the function contract expressed what I wanted it to. A tool to help point out common issues that arise when writing function contracts such as providing impossible pre-conditions would be very useful.

Another useful tool would be something that generates input/output pairs that satisfy the pre and post conditions of the function. While this is not garunteed to catch any issues, it is possilbe that it could potentially catch issues

like with the sorting specification where [3, 2, 1, 2, 4] to [1, 1,2, 3, 4] is a valid input/output pair based on the specification. This could provide a brief sanity check of the specifications.

5.6 Language Tooling

With modern programming languages, there are many tools that go along with them. One of the two major ones that help with developer productivity are syntax highlighting and intellisense (smart autocomplete, error checking, etc). As of writing this, I am not aware of any tools that provide either for ACSL.

5.7 Functions as predicates

Consider the scenario where you have written the function contract for a sorting function, similar to the one that was talked about in the Motivation section. Now imagine there was a second function which used the sorting function

```
1 void foo(int *arr, int n) {
2     sort(arr, n);
3 }
```

The function contract for this function would have to be essentially the same as the sorting function. It would be useful to be able to re-use the conditions established in the sorting function and apply them to this one. From a developer standpoint, this can be improved upon by writing the conditions of your functions as predicates that can be used across definitions, however it would be nice if those predicates were provided by ASCL for free if the contract already existed. Ther were many times throughout this project where I essentially had to re-write the contract of a function that another function called.

6 Future Work

This paper has a relatively narrow scope, and there is still much more to be learned about formal verification tools. One idea for improvement is to do a case study across different verification tools. Other such tools and specification languages such as JML for Java (https://www.cs.ucf.edu/~leavens/JML/index.shtml) verus for Rust (https://github.com/verus-lang/verus) and the dafny programming and specification language (https://dafny.org/). It would be interesting to see how the experience and capabilities vary across the different tools that currently exist.

Another consideration for future work is to take the approach of writing a software system from the ground up with verification in mind, as opposed to taking an existing project and attempting to retroactively add verification. Contrasted with this experience, it could provide insight into how to better write verification oriented software.

7 Conclusion

This paper has explored the application of the Frama-C WP plugin to formally verify a C-based semantic versioning library. Through this case study, I have demonstrated both the strengths and limitations of formal verification techniques. While WP provides a powerful tool for ensuring software correctness across all defined, the effort required to write accurate and meaningful function contracts is not something that can be ignored. The learning curve associated with ACSL, combined with the ability to write well defined function contracts for complex software results in a large amount of work that needs to get done just in the verification process.

As it stands now, it seems like the development of these tools is mainly focused on what is technically possible, while developer productivity is not at the forefront. The prospect of having such powerful verification tools is exciting, and there are many things that can be done both on the sides of the users and the creators of these tools.

There are clear opportunities for improvement, both in the development of formal verification tools and in the learning resources available to developers. Enhancing support through more robust libraries of axioms and predicates, better educational resources, and automated contract auditing would make the process more accessible. Additionally, improved language tooling, such as syntax highlighting and macros for contract generation, would streamline the verification workflow and reduce the cognitive overhead for developers. The continued development of these tools and continues assessment of their usefulness has the potential to lead towards formal methods being integrated more effectively into real-world software development.

References

Blanchard, A. (2020). *A gentle introduction to c code verification using the frama-c platform*. Zeste De Savior.

Frama-C - Framework for Modular Analysis of C programs. (2024, September). https://frama-c.com

V Garousi, J. Z. (2013). A survey of software testing practices in canada. *Journal of Systems and Software*.