Automatic precondition generation for VerCors

JORT MOL LOUS, University of Twente, The Netherlands

ABSTRACT

Program verification is a field of computer science, which focuses on determining if a program functions correctly. This is done by defining pre- and postconditions and possibly other related specifications. However, the writing of these specifications is still very much a human task. One of the tools that suffers from this burden is the verification tool *VerCors*. To reduce this burden, algorithms have been proposed for generating preconditions based on postconditions. In this paper, we look at multiple currently available tools that use such algorithms to generate preconditions. We compare their use cases with the scenarios in which *VerCors* is usually used, looking at their overlap. We look further at a specific tool that shows higher potential, *PGen*, and see the value of integrating it within VerCors. This is done by testing on multiple self-made examples and a set of real-world examples, that are used to test *VerCors*. Using the results we determine if this tool should be implemented within *VerCors*

Additional Key Words and Phrases: VerCors, Specifications, Automation, Precondition Inference

1 INTRODUCTION

VerCors is a tool for program verification tool that is developed by the FMT group within the University of Twente. This tool focuses on deductive verification for parallel and concurrent programs. For such a verification tool to be able to analyze the correctness of a program, it needs specifications to be written for it. These specifications are written per function in a program and can be categorized into two subcategories: Those that specify how a program should behave, and those that help the tool with proving this behavior [10]. The general forms of this first category are pre- and postconditions. Preconditions specify the expected input, e.g. by limiting values to be in a specific range, and postconditions specify the expected output (given the expected input) of a function, e.g. checking if the output values are within the same range. An example of a specification in the second category is that of loop invariants, which specify conditions that should be true before and after each iteration of a loop. These extra specifications allow a verification tool to verify the iterations of a loop instead of the entire loop at once, and without them, verification of a program may become infeasible.

However, the writing of these specifications is one of the bottlenecks for verification [4]. In response to this issue, research is being done into automatically generating preconditions. This research focuses specifically on what is called the maximal precondition, this is the logically weakest precondition, so any weaker precondition would allow inputs that violate the postcondition. Recently there were some innovations within the field of precondition generation [23, 24]. While this research does not focus on generating preconditions for code, it uses C-like code to show how their research works. A precondition generator for code, however, is quite interesting for use within *VerCors*. As such, there will be a look at the use cases of these tools. Aiming to answer the question of how these tools can be used to generate annotations for code that is verified by VerCors.

In this research, there will be a brief look at currently available precondition generators and cover their suitability for use together with *VerCors* in section 4. Then the workings of *PGen* will be analyzed, to see how it can be used within VerCors. This will be done by running extensive tests from both real-world examples from *VerCors*[11] as well as self-made examples. This analysis also hopes to find the limitations of *PGen*. The methodology of these tests will be covered in section 5, while the results of these tests will be covered in section 6. Finally, in section 7, a final analysis of the use within VerCors will be covered.

2 RELATED WORK

Much of the work of automatic precondition generation is based on the field of Satisfiability Modulo Theories. This is a field focused on finding whether certain formulas are satisfiable. This field includes checking if programs successfully execute, and thus also includes the field of generating preconditions. An initiative to create a common format for it is that of SMT-LIB [3]. This format has been adapted by one of the most popular SMT solvers Z3[7], and many programs that use z3 will also translate their functions into this format. Some other programs in the field of program verification are those of [9] and [8]. For background on the work that has been done for translating programs into a format for which they can be verified [2] can be read. We briefly cover the concept of deductive verification, but for a further background of this deductive verification [14] offers a concise history. We describe specification as a bottleneck, as it is a problem encountered within *VerCors*. For some research into why this is a problem for the field of program verification, [4] has done research identifying the writing of specifications to be a bottleneck in program verifications. Java is a language that is barely touched by this research, despite being one of the biggest programming languages. If the reader is interested in more information about specification inference tools for Java, [18] covers way more than is covered in this paper. They also cover the strengths and weaknesses of these tools. Some work done related to precondition generation, but not covered in this paper can be found in [1, 6, 13].

3 PROBLEM STATEMENT

While the *VerCors* tool is built to verify the correctness of programs, it has to verify the entire function in one go. In practice, this often means that a function will have to be split into smaller verification tasks to test the functioning of each part separately. This, however, results in a large amount of annotations having to be written for these functions. To reduce the burden of writing these annotations, automation can be used. As there is no such automation in place

TScIT 41, July 5, 2024, Enschede, The Netherlands

Table 1. Tools and their results

Tool	Method	Result
MaxPraNQ [21, 23]	Convert C code using SeaHorn[12] and FreqHorn[8]	Manual rewrite is too intensive, SeaHorn transformation violates the given grammar
PreQSyn [22, 24]	Same as MaxPraNQ	Same as MaxPrANQ
cccheck [5]	Write code in C# and use visual studio	Only works with visual studio and C#, so almost no overlap with the use cases of VerCors
PIE [19]	Write code in C++ and use the C++ implementation of PIE	The C++ implementation is not properly archived and documented
QPR Verify [16, 17]	Download the artifact and run on VerCors examples	Running the given example does not work
PGen [20]	Download the binary and run on VerCors examples	Manages to generate correct preconditions

currently, the goal of this research is to find a way in which such automation could be implemented into *VerCors*. This resulted in the following research question:

In what way can current precondition generation tools be applied to generate annotations for code that is typically verified by *VerCors*?

This will be answered by the following subquestions:

SQ1: Which of the currently available precondition generators can generate preconditions for code?

SQ2: For which kind of programs can these tools generate preconditions?

SQ3: How can the output of these tools be used to generate VerCors annotations?

To answer the first subquestion, multiple tools were (briefly) investigated for their appropriateness for generating preconditions. A quick overview of the results can be found in table 2.

3.1 PreQSyn and MaxPrANQ

The first tools that were investigated were the tools *PreQSyn* [24] and *MaxPrANQ* [23]. These are the most recent tools to generate preconditions. Since they are quite recent they were extensively investigated for their use.
These tools take as input a program in the SMT-LIB2 [3] format. This is a format commonly used by SMT solvers. These solvers are commonly used to prove the correctness of programs in tools like *VerCors* (which uses the SMT solver Z3). This means that for it to verify the code, that code will first need to be translated into SMT-LIB2. There are already existing tools for that purpose, for example for C there is the tool *SeaHorn* [12] and for Java, there is the tool *Jayhorn* [15]. Both of these tools have to goal of verifying the runtime correctness of programs, and they apply certain transformations to the programs to be able to more optimally verify this.

MaxPrANQ and *PreQSyn* only work on a certain subset of programs. This is a subset called "linear array programs". This subset consists of programs that can read and write to variables and the elements of arrays. These arrays can only be accessed in a for loop for which in iteration *i* you can access element *i* of the array.

At first, Jayhorn was used to translate Java code to the SMT-LIB2 format. However, it was quickly found that as this tool is mostly for reasoning about runtime correctness, many variables are added to model the Java runtime. On top of that, the tool reasons from the main method, which makes it less appropriate for verifying independent functions with undefined variables, which is the main type of function that preconditions are defined for. For example to model the content of a function *test*, one would have to call the function in main, one way of doing this can be seen in example 1. When translating this code, *Jayhorn* adds about 50 variables in the translation. This makes the program too complicated to be verified by *MaxPrANQ* and *PreQSyn*. Taking this into account, Jayhorn did not seem to be appropriate for generating preconditions for Java code.

```
1  class AddAssignJava {
2
3    void test() {
4        int x = 5;
5        x += 2;
6    }
7
8    public static void main(String[] args) {
9    AddAssignJava a = new AddAssignJava();
10   a.test();
11   }
12 }
```

Code example 1. a simple Java program

After that SeaHorn was tried to translate C code to the SMT-LIB2 format. One of the examples that were tested for this can be found in example 2. This code was adapted from one of the examples of [23] to work together with *SeaHorn*. *SeaHorn* does not have any strict grammar, so no large changes have been made. Some smaller changes were made, such as defining the length of the array in line 2 and defining the array as a global variable. Further the examples were changed to use the *sassert* function of *Seahorn*, which allows it to more effectively reason about assertions and postconditions.
Compared to Jayhorn, *SeaHorn* manages to encode C code at runtime with fewer variables. The transformations of *SeaHorn* however, do not preserve the grammar of the linear array program that is needed. *SeaHorn* encodes the access of (integer) arrays by accessing them with j + 4*i. This can be seen in how lines 9-10 of example 2 are

encoded, which can be seen in example 3. Example 4 shows the same lines after being further transformed by *FreqHorn*[8], to show it in a more readable format.

```
#include "seahorn/seahorn.h"
#define N 1000

int a[N];

int main()
{
  int i;
  for(i=0;i<N;i++)
    a[i]=2*a[i];

  for(i=1;i<N;i++)
    sassert(a[i] >= 2*i);
}
```

Code example 2. a C program doubling the content of an array

```
(= main@%_2_0 (+ @a_0 (* 0 4000) (* main@
  %.0.i1_0 4)))
...
(= main@%_3_0 (select main@%shadow.mem.0.0
  _0 main@%_2_0))
(= main@%_4_0 (* main@%_3_0 2))
...
(store main@%shadow.mem.0.0_0 main@%_2_0
  main@%_4_0))
```

Code example 3. A fragment of the same program after being transformed through seahorn

```
(let ((a!1 (* 2 (select _FH_2 (+ _FH_0 (* 4
  _FH_1))))))
...
(store _FH_2 (+ _FH_0 (* 4 _FH_1)) a!1)))
```

Code example 4. A fragment of the same program after further being transformed through freqhorn

This transformation of *SeaHorn* replicates the way C stores and accesses arrays in memory, however, this makes a program that would be within the class of linear array programs be outside of the class when translated. A translation without encoding the runtime behavior of the programming language is possible, but the main interest in verifying programs seems to be accounting for runtime behavior. Naturally, the runtime of most languages does not fit within the grammar of linear array programs, so there does not seem to be a tool whose main purpose is translating code to SMT-LIB2 format.

This makes the use of *MaxPrANQ* and *PreQSyn* very limited for working on code. The authors of *MaxPrANQ* and *PreQSyn* were contacted about a potential translation, and they suggested that a transformation respecting the grammar must be done manually. As such these tools cannot be implemented within VerCors to automatically generate preconditions based on the code.

3.2 The search for another tool

As the tools originally proposed were found to not be appropriate, another tool would need to be found to continue the research. As *VerCors* mainly works on C, Java, and OpenCL, the new tool would have to work on one of those languages or a language similar to it.

3.2.1 cccheck. One of the tools we looked at was *cccheck* [5]. This is a tool for generating necessary preconditions for C# (.NET) code. This is implemented in CodeContracts, an extension for Visual Studio. However, no implementation outside of Visual Studio was given. As Visual Studio is not used together with VerCors, this tool was thus not considered further.

3.2.2 PIE. Another tool that was looked at was that of PIE (Precondition Inference Engine). This tool aims to generate preconditions for C++ and OCaml programs. The project has been further continued by a project called LoopInvGen [19], this project however works on a format called SyGuS so it has a similar problem to that of *MaxPrANQ* and *PreQSyn*. The PIE version has been archived, but only the OCaml version seems to have been kept. Since OCaml is not supported by *VerCors* and C++ is barely supported, this tool was also deemed inappropriate.

3.2.3 QPR Verify. The work of [16] implements a precondition generator in the tool *QPR Verify* [17]. This approach tries to generate preconditions based on bounded model checker theories. Following the instructions to run the precondition-learner functionality of *QPR Verify* with the provided command and examples gives an error. As following the instructions gave an error this tool was dropped right after that.

3.3 PGen

Finally the tool *PGen* [20] was tried. This is a tool that generates preconditions for C code. It does this by trying to find all the inputs for which the program violates the postcondition. To find these inputs it estimates both the inputs for which the program satisfies the postcondition and those for which it violates the postcondition. It will then try to refine both sets until there is no overlap between the sets. At that point, the set of inputs that satisfy the postcondition should be the precondition.
Unlike the other tools, this tool was quite easy to set up. This tool also seemed more promising because it showed its ability to generate quantified preconditions within the benchmarks. This tool almost immediately worked on example C programs, generating valid preconditions. As this was the easiest-to-use tool, this tool was chosen to be further investigated.

4 METHODOLOGY

To check for which kind of programs *PGen* can generate preconditions, it was run on multiple tests. These tests are separated into two different categories. The first category is that of self-written tests, these tests are mostly array programs that test simple functions that you might see in bigger programs. The second category is that of code examples from VerCors. These tests were taken from the public

repository of tests made for testing VerCors [11]. The tests are categorized into different categories based on programming concepts. For the selection of these tests, finding tests that contained arrays was prioritized. So, tests were taken from concepts that commonly use arrays. This was to validate the patterns found by real-world examples. The rest of the tests were taken from the 'C' category of tests, as these covered some C- and general programming concepts that were not covered by the self-made tests.

4.1 Writing the tests

For the self-written tests the basic example that was started with was the following:

```
1  void foo(int ar[],int len){
2    int i;
3    int k;
4    for(i=0;i < len;i++){
5        ar[i]=0;
6    }
7    for(k=_ZERO; k < len; k++) {
8      if(ar[k] != 0) goto ERROR_1;
9    }
10   goto end;
11   ERROR_1:;
12   end:;
13 }
```

Code example 5. a program checking if array elements are zero

The code in example 5 initializes an array with all values being zero in a loop at lines 4-6.
Lines 7-9 model the postcondition:

```
ensures(\forall int k; 0 <= k && k < len; ar[k] == 0);
```

Since *PGen* reasons about the correctness of a program by checking if it can reach an error state, for it to be able to do so an error state needs to exist. This is the flag ERROR_1 at line 11. To encode the postcondition the inverse of the postcondition is taken, which is

```
\exists int k; 0 <= k && k < len; ar[k] != 0
```

When that is true the program is sent to the ERROR position, this is done in line 8. Since the error state is a flag, an extra flag is needed to skip over the error state for successful executions. This is the flag *end* at line 12, and the goto at line 10 skips the error state.
To model the range 0-*len*, the variable k is initialised as _ZERO. This is because *PGen* reasoning works better on abstract ranges. The range of the initialization loop, however, is modeled from 0 - *len*. Encoding the initialization and the postcondition differently has shown to give better results, this will be further elaborated on in the results.

These changes were also applied to the other examples, including those taken from VerCors. Apart from adjusting the programs to fit these traits, these programs also needed another change to be made for the program to work. The syntax requirement for *Pgen* specifically requires all variables that are going to be used in a function to be declared at the start of a function. If that is not the case it will give a syntax error. This is also needed for for loops, which means that:

```
for (int i = 0; i < len; i++)
```

would need to be changed to:

```
int i;
for (i = 0; i < len; i++)
```

Due to the late discovery of this fact, some postconditions were written in the functionally identical while notation, this has not shown to change performance.

The *zero_array* test was taken as a basis to test multiple options and multiple notations. As such many tests follow the same format as this test with (slightly) changed initialization and postcondition. For instance, example 6 shows a variation that tests multiple postconditions.

```
1  void foo(int ar[], int len) {
2      int i = 0;
3      int k = _ZERO;
4      while (i < len) {
5          ar[i]++;
6          i++;
7      }
8      while (k < len) {
9          if (ar[k] % 2 != 0 || ar[k] % 3 != 0)
       goto ERROR_1;
10         k++;
11     }
12     goto end;
13     ERROR_1:;
14     end:;
15 }
```

Code example 6. a program testing multiple post conditions

To test different notations, many of the examples also have a variant with only the postcondition loop and a variant using loop invariants. The examples with only postconditions were created by removing everything except the postcondition from a program. As can be seen in example 7, which is missing lines 2 and 4-6 from example 5. This made it possible to check if the first loop was taken into account when generating the precondition.

```
1  void foo(int ar[],int len){
2    int k;
3    for (k = _ZERO; k < len; k++) {
4          if(ar[k] != 0) goto ERROR_1;
5    }
6    goto end;
7    ERROR_1:;
8    end:;
9  }
```

Code example 7. a version of example 1 with the initialization removed

The loop invariant variation, however, was used as a performance comparison. In theory, one loop should be easier to reason about than two loops that depend on each other. These programs thus aim to check if this theory impacts *PGen*. These variations were

made by putting the postcondition check inside the first loop in a program. In example 8 this is done by combining lines 4-7 and 7-9 from example 5 into lines 3-6. This means that the postcondition is translated into a loop invariant of ar[k] == 0.

```
1  void foo(int ar[], int len){
2    int k;
3    for(k=_ZERO;k < len;k++){
4         ar[k]=0;
5         if (ar[k] != 0) goto ERROR_1;
6    }
7   goto end;
8   ERROR_1:;
9   end:;
10 }
```

Code example 8. example 1 with only the postcondition

These variants can be quite similar, but for reproducibility, they are listed separately.

4.2 Running the tests

To run these programs the following command was used:

```
time ~/artifact2/tools/P-Gen/p-gen --reach --mainproc foo --file ./zero_array.c --stringabs --expheap --beyondwp --precond --noinline --tprover z3 --dontabsarray
```

Here, *reach* indicates that it will reason by trying to reach the error state, no way for the tool to generate preconditions was found with this option disabled. *mainproc* gives the name of the function to be analyzed, for all self-made tests this program was named foo for simplicity. *stringabs* tells the program to treat strings as arrays, this was not necessary for any self-made tests, but no performance drop was found by enabling it. The *expheap* option tells the program to encode the heap as an array. Similarly to the *stringabs* option, enabling this option did not seem to affect the programs for which it was not needed. The *beyondwp* option enables a more advanced refinement option, this option allows the program to find quantified preconditions. The *precond* option tells the program to generate a precondition, disabling this option only tells the route to the error state instead of generating preconditions. The *noinline* option tells the program to not change the variable names, this makes the generated preconditions understandable. The *tprover* option tells the program which SMT solver to use, in this case, Z3 is used, which was also suggested as the solver that worked the best with *PGen*. Version 2.2 was used as it was used in combination with *PGen* in [22]. The latest version was also tried, but it is not backward compatible with version 2.2, and *PGen* uses options that have been removed and/or renamed. Finally the *dontabsarray* option tells the program to not abstract arrays, all programs were run with this option both on and off because this option gave quite varying results. The total program was run with the Linux time command, to measure the relative time it took for programs to execute.

The outputs of running these tests were collected in one place in combination with the time it took to run them.

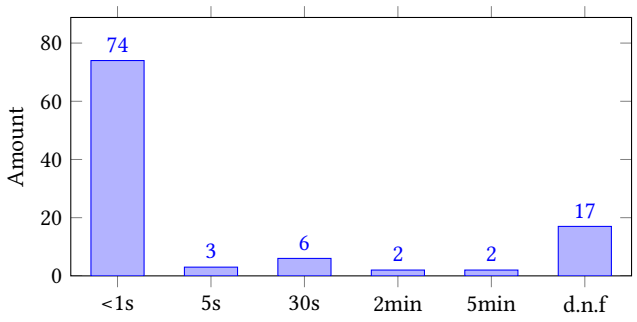

Fig. 1. Execution time for programs

5 RESULTS

5.1 SQ2

We look at the results from multiple examples to answer the question of which kind of programs *PGen* works. The results of running the tests can be found in table 2. This table shows how many of which tests gave which result. Since the generated preconditions could be quite long, the results were described in multiple categories. These categories are: contains the precondition, incorrect precondition, lacking a quantified precondition, too slow to finish, and intermediate precondition. Lacking a quantified precondition refers to programs that expect a quantified precondition, but for which it only reasons about a limited amount of array elements (such as the first three elements). Intermediate precondition is used to refer to a precondition that ignored a certain part of the program, and is thus true somewhere in the code but not at the start. For a program to be considered too slow it would have to run for more than 5 minutes, since in general it could be determined at that point if it was at the point of terminating or not. The time for Z3 to timeout was set to around 5 minutes, so when Z3 timed out the program was considered too slow. Since z3 is called multiple times per iteration, it was unlikely that the program would terminate after one Z3 timeout. A representation of the time taken for programs to terminate is given in figure 1. Since performance can differ these were rounded with as much as a 50% error margin to the nearest number in the graph. Two programs managed to terminate right after the soft time limit of 5 minutes was passed, as these tests managed to finish before I was able to cancel them.

As can be seen in table 2 36/104 of the tests found correct preconditions. There are also 12 tests which find intermediate preconditions. These preconditions are for programs with two loops, for which it only finds a precondition that is true before the second loop. As intermediate checks are used to assist in verification these can also be considered useful outputs.
None of the programs with two loops managed to reason about both loops. This is partially due to choosing to have one loop as a range from 0 to *len*, while the postcondition was from *_ZERO* to *len*. However, programs where both loops had a range from 0 to len gave no quantified postcondition, while still only reasoning about the second array. On the other hand, changing both arrays to have a range from *_ZERO* to *len* ends in non-termination. Combining

Table 2. Test results

Category	Total	found precondition	too slow to finish	no quantified	incorrect precondition	intermediate precondition
All tests	104	36	17	17	22	12
dontabsarray enabled	52	19	14	6	9	4
dontabsarray disabled	52	17	3	11	13	8

Table 3. Variant results

Category	Total	found precondition	too slow to finish	no quantified	incorrect precondition	intermediate precondition
No modification	18	2	4	1	3	8
loop invariant variation	18	12	2	1	3	0
Only postcondition loop	18	9	4	1	4	0

arrays into one, such as with the loop invariant variations, does allow for reasoning over the content of multiple arrays.

There are 17 programs that took too long to terminate, this seems to be because the program does not stop after finding one quantified precondition. When running with the *dontabsarray* option, the program will try to fill in multiple values. This means that the program can spend a lot of time testing different values. The programs that failed to terminate can be split into two categories. The first was the category of postconditions which checked if the array values were bigger or smaller than a certain value. Unlike equals, smaller than and bigger than have a range of correct values, this range means that the program can try a lot of values without discovering that it should terminate. The other category of programs that took too long too terminate were the programs that relied on multiple postconditions. These programs similarly struggled because they did not realize they could have a lot of values to check, e.g. they would have to determine that all values in an array were not one of two values. When running the program without the *dontabsarray* option, the first category successfully managed to find preconditions, however the second category only found preconditions for one of the postconditions.

The programs that did not give a quantified precondition were all quite different. If a feature was not supported, a program likely ended up in this category. This includes features like floating points, array comparison, functions and nested arrays.

For the incorrect preconditions, the previously mentioned category of programs with multiple postconditions also manages to result in incorrect preconditions, being the only programs for which having *dontabsarray* enabled resulted in an incorrect precondition instead of non-termination. The difference with the programs that did not terminate, is that these programs have postconditions that complement each other. These consist of programs that check if a number is divisible by both 2 and 3 for example. The other programs in this example result in preconditions where the preconditions where less strict than the expected precondition, such as with conditional values.

As mentioned in section 4, for certain programs there were multiple variations created. The differences between the variations van be found in table 3. As can be seen in the table, in general, the programs that use loop invariants have the best general results. In none of the examples where *PGen* fails to find the correct precondition for the invariant version, did it manage to find a correct precondition the other variants of the program. Most notably it allows for programs to account for the changes done in the initialization phase.

5.2 Unexpected behaviour

Some unexpected behavior was also found. One of the oddest behaviors was found. This program increments an array and checks if the value is equal to five after incrementing. This program fails to terminate with the option *dontabsarray* enabled. This is odd because it does terminate for the value four, and also for i + 1. So it can reason about undefined variables and lower values, but it struggles when the value is higher than five. This discovery was relatively late and the impact on real programs could not be fully researched.

Another somewhat more predictable behavior is the difference between having the option *dontabsarray* enabled and disabled. When finding the correct postcondition, having the option disabled always results in finding the quantified condition negated. While for all quantified conditions found while the option was enabled (which were also all correct), *PGen* always returned the condition without the negation. This can be explained by the fact that having the option disabled means it will always exit earlier when having to reason about arrays, and thus the program was still in the process of proving. The proof is done by proving the inverse is unreachable, so it makes sense for the inverse to be returned. The more unexpected part of this behavior is the fact that if the program terminates early with the option enabled, which it sometimes does, the result is not negated.

Further, the attempts at encoding existential (there exists) postconditions were disappointing. Replicating the encoding given in the benchmarks resulted in a program that did not give a quantified precondition. Using another implementation made the program either not terminate or give a non-quantified precondition. Further examples from VerCors were run, from these examples it was found that floats were not properly supported by PGen. When their values are integers they can reason about them, however, it will model any

other value as a new variable. For example, for the code of example 9 it generates the precondition

```
:
    ABS_STR_2- test3( 3 , ABS_STR_3 ) == 0
    && ABS_STR_5- ABS_STR_4< 0
```

```
1  int test3(int x, int y){
2      return x % y;
3  }
4
5  int main(){
6      float a = 0.510000;
7      if (1.0 != test3(3, 2.0)) goto ERROR_1;
8      if (3.0<=2.0) goto ERROR_1;
9      goto end;
10     ERROR_1:;
11     end:;
12     return 0;
13 }
```

Code example 9. A program checking floating-point operators

Some function calls outside of the program were found to be modeled similarly, with the code in example 10 having the *ceil_f32* function being replaced by a variable called *temp_var*. This mean the generated precondition is:

```
temp\_var\_1- 1 == 0
```

```
1  int main(){
2      float a = 0.510000;
3      int b = (int)(ceil_f32((a + -0.500000)));
4      if (b != 1) goto ERROR_1;
5      goto end;
6      ERROR_1:;
7      end:;
8      return 0;
9  }
```

Code example 10. A program checking the functionality of the ceil function

However, many of the run examples also tested simple concepts, with similar concepts. This meant that the examples did not extend the coverage that much.

5.3 SQ3

To look at how the output of *PGen* can be used to generate *VerCors* annotations, we take a further look at the output. An output given by this program can look like this:

```
(len- _ZERO- 1 < 0 || FORALL( UNI_1>= _ZERO, UNI_1<= len- 1 , ar[ UNI_1] + 1 % 2 == 0  ) )&&  (len- _ZERO- 1 < 0 || len- _ZERO- 2 < 0 )
```

This is an example of an output of the category containing the correct precondition. The precondition of *len- _ZERO- 1 < 0 || FORALL(UNI_1>= _ZERO, UNI_1<= len- 1 , ar[UNI_1] + 1 % 2 == 0)* is correctly identified by the program, however, it adds an extra condition that makes the total condition incorrect. When given as output one can see that the second part of the condition makes this condition trivial and can reason that the first part of the condition is the real condition. This means that the first part of the condition can be translated into a precondition in VerCors, but the second part should be ignored. No preconditions were generated that were not supported by VerCors and the syntax is very similar to that of VerCors. As such all of the generated preconditions are viable to be translated to VerCors preconditions without much trouble. For the examples from VerCors, there was a surprising lack of preconditions specified in the examples taken. A common precondition was that of disallowing zero for division, this was not found by *PGen*. The other examples were quite simple and able to be verified such as an increment function with

```
requires y>=0;
ensures  \result > 0;
```

as their contract. This means that the examples that were compared had quite a good success rate, but the samples were not diverse enough to allow for an analysis.

6 CONCLUSIONS

To answer the question of if *PGen* should and can be implemented in VerCors multiple factors should be accounted for.
First, the ease of translation should be considered. Since the tool works on C code little changes have to be made to the program. Having all variables be declared at the top of the program is inconvenient, but an easy transformation to perform. Encoding a postcondition separately is done easily by adding any variables at the top, and putting an if condition at the end that sends the program to the error state if the inverse of the postcondition is true. For a quantified postcondition, the variable that will be used to loop over the array needs to be undefined for the program to return a quantified precondition. The inverse of a universal postcondition can easily be modeled by sending the program to an error state when a single element of the array violates the requirement. To model the inverse of the existential postcondition, having a loop that continues while the element does not satisfy the condition can be used. When the end of the array is reached, it can then be sent to an error state. Secondly, the translation of the results should be considered. As mentioned in the results, many of the generated conditions contain both the correct precondition and an incorrect part of the precondition. When looking at automatically converting these postconditions one could try exclusively looking at taking the part of conditions with quantified preconditions. However, these conditions can contain multiple quantified postconditions. While taking the first quantified condition seen seems to generally give a correct precondition, it cannot be considered reliable enough for an automatic translation to be able to be considered. While a human would be able to have the same problem, this technique is easy enough for a human to execute when given the raw output of the tool.

Most importantly, the grammar of the languages for which this tool works should be considered. One important distinction to get out of the way is that this tool does not work well when defining multiple postconditions that should be true at the same time, these

have not given any valuable results in testing. For types other than ints, *PGen* tends to replace them with a variable, seeing the value replaced by a variable, it also tends to combine a concrete value and a newly made variable into one variable. This means that for floating points and similar non-integer types that rely on mathematic operations, *PGen* is not very appropriate. For types like string and char, this replacement does not affect the program as much. In a similar vein, functions are also replaced by variables. While not excluded, bigger than and smaller than only work with the *dontabsarray* option disabled, as such these programs are likely to give less qualitative results. For arrays, *PGen* only manages to find correct quantified preconditions when there is one loop. Further nested arrays are not supported.

The mentioned translations are quite easy to implement, so implementing *PGen* is a low investment. However, the value of the results can also be quite low. Many test programs took a long time to terminate, and slowing down work to wait for this tool to generate a precondition can decrease efficiency. On top of that multiple postconditions at the same time cannot be modeled, which makes the tool almost unusable in a situation where you have multiple postconditions. Running the tool multiple times for each postcondition can give a precondition for each postcondition, and taking the intersection of these preconditions could theoretically result in a useful precondition. This however is still quite a burden on the user and does not reduce the burden of writing specifications. Another negative, but slightly less impactful is the inability of the program to reason over multiple loops. It may only reason over one loop at a time. This however can be rectified by separating the program into smaller tasks and verifying each task separately, for example using the precondition of the next loop as a postcondition for an earlier loop.

In conclusion *Pgen* is quite easy to use, and also quite easy to create a simple implementation for. The benefit of this implementation is however doubtful. Since the implementation is so simple, however, it is also possible to try it first and determine if it improves productivity from person to person. What has been found as a great advantage is having a frame of reference. Having a wrong output might be able to allow one to reason to the correct precondition, while with no output one would have to reason from scratch.

7 LIMITATIONS

An obvious limitation was time. Not only is around 6 weeks not much time for research, by having to pivot the research halfway through those 6 weeks gave me about 3 weeks of actually being able to work with the new tool. While having a similar purpose, my first 3 weeks did not help me in getting more efficient in testing, apart from the fact that I already had some tests ready. The tools were too different for the effort to be able to be translated into the new tool. Further, this time constraint also made me think more limited in possibilities and made me make some unfounded assumptions just to be able to go further with testing. An example of such an assumption was assuming that for loops did not work because the standard notation for for loops gave a syntax error. The issue turned out to be the fact that variables were not allowed to be declared inside a for loop. This combination made me able to try significantly fewer benchmarks than I wanted to try, and probably a smaller variety than I wanted to.

On the other hand, I feel like I limited myself too much when looking for a new tool to continue the research with, I chose an option for which I could get results, but it might have been more interesting to change to research to compare the usefulness of different tools as I feel like the field is large and I did not give enough tools a try. Something that did not help in this fact is the difficulty of finding a tool based on the research. A decent subset of research I could find did not have a public implementation and was thus not considered.

Somewhat implied by the previous paragraph was the limitation of creativity. The limited grammar of the original tools made me mostly think of tests within that grammar. However, *PGen* did not mention these limitations anywhere, as such some interesting edge cases were only thought of after the point where testing was determined to be finished. This allowed for some testing but no extensive testing, which means I might have missed some interesting use cases. This problem was exacerbated by the limited benchmarks and documentation of PGen, which lacked some examples for which it failed to find a precondition. While I think the research covers a lot of common patterns you might want to look at, I think the benchmarks are the largest point of improvement to be made if the research is repeated.

8 FUTURE WORK

One interesting research could be translating code to SMT-LIB2 format in a way that retains the linear array program grammar, to optimally make use of the more recent tools of [23, 24]. This is the approach that created tools like *SeaHorn* and *JayHorn* in the first place. However, this is unlikely to be useful outside of the purpose of researching these specific tools, as both *SeaHorn* and *JayHorn* were made with the purpose of verifying and not with the purpose of translation. It might thus be more interesting to look into the different ways in which SMT-LIB2 format is used and see if there could be a way of encoding programs that is multi-purpose, instead of optimizing the translation for a specific tool.

Apart from that, there is an interest in looking at other precondition generators which might have been missed by this research. As the benchmarks that were used are quite easily reproducible in other languages, the research could look at a less limited subset of languages than those that are verified by VerCors. There might be a more powerful tool for a language that was not investigated, for which the translation from VerCors is easy.

ACKNOWLEDGMENTS

I would like to thank Marieke Huisman and Alexander Stekelenburg for supervising this project

REFERENCES

[1] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages (POPL '16). Association for Computing Machinery, New York, NY, USA, 789–801. https://doi.org/10.1145/2837614.2837628

[2] Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi, and Maurizio Proietti. 2022. Analysis and Transformation of Constrained Horn Clauses for Program Verification. *Theory and Practice of Logic Programming* 22, 6 (Nov. 2022), 974–1042. https://doi.org/10.1017/S1471068421000211

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[4] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. 2012. Lessons Learned From Microkernel Verification — Specification is the New Bottleneck. *Electronic Proceedings in Theoretical Computer Science* 102 (Nov. 2012), 18–32. https://doi.org/10.4204/EPTCS.102.4

[5] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, Berlin, Heidelberg, 128–148. https://doi.org/10.1007/978-3-642-35873-9_10

[6] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *Verification, Model Checking, and Abstract Interpretation*, Ranjit Jhala and David Schmidt (Eds.). Springer, Berlin, Heidelberg, 150–168. https://doi.org/10.1007/978-3-642-18275-4_12

[7] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *2008 Tools and Algorithms for Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 337–340. https://www.microsoft.com/en-us/research/publication/z3-an-efficient-smt-solver/

[8] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2019. Quantified Invariants via Syntax-Guided Synthesis. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 259–277. https://doi.org/10.1007/978-3-030-25540-4_14

[9] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*, José Nuno Oliveira and Pamela Zave (Eds.). Springer, Berlin, Heidelberg, 500–517. https://doi.org/10.1007/3-540-45251-6_29

[10] University of Twente FMT group. n.d.. Tutorial VerCors Tool FMT UTwente. https://utwente.nl/vercors

[11] FMT group, University of Twente. n.d.. vercors/examples at dev utwente-fmt/vercors. https://github.com/utwente-fmt/vercors/tree/dev/examples

[12] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[13] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2018. Quantifiers on Demand. In *Automated Technology for Verification and Analysis*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 248–266. https://doi.org/10.1007/978-3-030-01090-4_15

[14] Reiner Hähnle and Marieke Huisman. 2019. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In *Computing and Software Science: State of the Art and Perspectives*, Bernhard Steffen and Gerhard Woeginger (Eds.). Springer International Publishing, Cham, 345–373. https://doi.org/10.1007/978-3-319-91908-9_18

[15] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 352–358. https://doi.org/10.1007/978-3-319-41528-4_19

[16] Marko Kleine Büning, Johannes Meuer, and Carsten Sinz. 2022. Refined Modularization for Bounded Model Checking Through Precondition Generation. In *Formal Methods and Software Engineering*, Adrian Riesco and Min Zhang (Eds.). Springer International Publishing, Cham, 209–226. https://doi.org/10.1007/978-3-031-17244-1_13

[17] Marko Kleine Büning, Carsten Sinz, and David Faragó. 2020. QPR Verify: A Static Analysis Tool for Embedded Software Based on Bounded Model Checking. In *Software Verification*, Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel (Eds.). Springer International Publishing, Cham, 21–32. https://doi.org/10.1007/978-3-030-63618-0_2

[18] Sophie Lathouwers and Marieke Huisman. 2024. Survey of annotation generators for deductive verifiers. *Journal of Systems and Software* 211 (May 2024), 111972. https://doi.org/10.1016/j.jss.2024.111972

[19] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (June 2016), 42–56. https://doi.org/10.1145/2980983.2908099

[20] Mohamed Nassim Seghir and Daniel Kroening. 2013. Counterexample-Guided Precondition Inference. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, Berlin, Heidelberg, 451–471. https://doi.org/10.1007/978-3-642-37036-6_25

[21] Sumanth Prabhu, Deepak D'Souza, Supratik Chakraborty, R Venkatesh, and Grigory Fedyukovich. 2024. Artifact for the Paper Titled "Weakest Precondition Inference for Non-Deterministic Linear Array Programs" To Appear in TACAS 2024. https://doi.org/10.6084/m9.figshare.24923517.v1

[22] Sumanth Prabhu, Grigory Fedyukovich, and Deepak D'Souza. 2024. Artifact for the paper titled "Maximal Quantified Precondition Synthesis for Linear Array Loops" to appear in ESOP 2024. https://doi.org/10.6084/m9.figshare.24945996.v1

[23] S. Sumanth Prabhu, Deepak D'Souza, Supratik Chakraborty, R. Venkatesh, and Grigory Fedyukovich. 2024. Weakest Precondition Inference for Non-Deterministic Linear Array Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 175–195. https://doi.org/10.1007/978-3-031-57249-4_9

[24] S. Sumanth Prabhu, Grigory Fedyukovich, and Deepak D'Souza. 2024. Maximal Quantified Precondition Synthesis for Linear Array Loops. In *Programming Languages and Systems*, Stephanie Weirich (Ed.). Springer Nature Switzerland, Cham, 245–274. https://doi.org/10.1007/978-3-031-57267-8_10