autoSOLLVEvv

STPM15-125

An automated tool to make SOLLVE Validation and Verification test suite more accessible for AD teams on exascale and pre-exascale systems.

Report

WBS Element: 2.3.2.11

ECP: SOLLVE

Nikhil Rao

MS, Cybersecurity (Expected Spring 2023)

University of Delaware

Dr. Sunita Chandrasekaran

Brookhaven National Laboratory/University of Delaware

Abstract

The OpenMP language evolves with the release of each new specification. The SOLLVE Validation and Verification suite aims to find faults in the implementation progress of various compiler vendors, especially on the newer OpenMP features that are being used by application developers.

The autoSOLLVEvv project aims to bridge the gap between the SOLLVE V&V test suite and the Application Developers/Vendors/Facilities by making the test suite more accessible by building an interactive and easy-to-use command line interface tool. This CLI tool seeks to inform its users of the compiler/runtime failures that have already been declared as bugs by the test suite.

Introduction

This document highlights the deployment of the tool, its usage, and the approaches that were taken to arrive at this stage of the software development life cycle. The following are detailed in all of the approaches:-

- Introduction to the method
- Overview of the database structure with the changes made for each iteration
- The outline of the program logic with a flowchart and a demonstration of the output
- Obstacles faced while building the approach
- Limitations of the approach

Deployment

The tool, autoSOLLVEvv, has been currently deployed on OLCF's pre-exascale system Crusher. Following are some of the steps employed for this deployment.

The forthcoming sections will narrate the various steps that took to put this tool in place. Section 1.1 discusses the installation of our tool. Section 1.2 narrates how a user can use this tool with optional parameters. Section 1.3 describes a few limitations of this deployment.

1 Crusher

1.1 Installation

The installation media is located in the UMS012 space on Crusher at "/sw/crusher/ums/ums012/SOLLVE/autosollvevv/".

The installation requires Python3 and Pip3 dependencies.

pip3 install -e /sw/crusher/ums/ums012/SOLLVE/autosollvevv/

mv .local/bin/ autosollvevv .

The -e installation option is for --editable, by which the user wouldn't have to install newer versions as the current version will automatically be updated without the need for reinstallation.

```
[nikhilrao@login2.crusher ~]$ pip3 install -e /sw/crusher/ums/ums012/SOLLVE/autosollvevv
Defaulting to user installation because normal site-packages is not writeable
Obtaining file:///sw/crusher/ums/ums012/SOLLVE/autosollvevv
Installing collected packages: autosollvevv
  Running setup.py develop for autosollvevv
Successfully installed autosollvevv
[nikhilrao@login2.crusher ~]$ mv .local/bin/autosollvevv .
[nikhilrao@login2.crusher ~]$ ls
autosollvevv  test.c
```

Figure 1: Installation Steps on Crusher.

1.2 Usage

A file argument is a minimum requirement for the tool to run. The file extensions are limited to C, C++, and Fortran, and other file types will be rejected.

```
python3 autosollvevv <file>
```

The optional arguments include

- -c – This is used to specify a compiler that the user is interested in.
  The current implementation on Crusher only includes LLVM, RocM, CCE.
- -cv – This is used to specify a compiler and its version.
  The current implementation of the tool on Crusher includes the following:-
  - LLVM – llvm_14,llvm_15,llvm_16
  - RocM – rocm_4.5,rocm_5.0,rocm_5.2
  - CCE – cce_14.0.0,cce_14.0.1
- -omp – This is used to specify an OpenMP version
  The current implementation of the tool on Crusher only includes versions .4.5,5.0,5.1
- -h – This is a help command.

When the parameters are input correctly the output of the program looks like the following image.

In Figure 2, a C file is provided, and the tool is informed that the user is interested in OpenMP version 4.5 and is looking into multiple versions of LLVM.

The output of the tool gives the line number, where the pragma is found, and then lists the tests with the compiler result as well as the runtime result of the test.

```
[nikhilrao@login2.crusher ~]$ python3 autosollvevv test.c -omp 4.5 -c llvm

--------------------------------------------------------------------
Line number: 1
Line: #pragma omp target map(tofrom:errors) defaultmap (present:scalar)

OMP 4.5 tests
Test Name: test_target_defaultmap.c OpenMP Version: 4.5
Related Pragma: #pragma omp target defaultmap
Compiler Name: llvm_14_0_0 Compiler Result: PASS Runtime Result: PASS
Compiler Name: llvm_15_0_0 Compiler Result: PASS Runtime Result: PASS
Compiler Name: llvm_16_0_0 Compiler Result: PASS Runtime Result: PASS


Test Name: test_task_target.c OpenMP Version: 4.5
Related Pragma: #pragma omp target map
Compiler Name: llvm_14_0_0 Compiler Result: PASS Runtime Result: PASS
Compiler Name: llvm_15_0_0 Compiler Result: PASS Runtime Result: PASS
Compiler Name: llvm_16_0_0 Compiler Result: PASS Runtime Result: PASS

--------------------------------------------------------------------
```

Figure 2: Usage demonstration on Crusher.

`

When incorrect parameters are set the tool informs the user of the error faced and what are the valid parameters for the program as seen in Figure 3.

```
[nikhilrao@login2.crusher ~]$ python3 autosollvevv test.c -omp 4.5 -c gcc
usage: autosollvevv [-h] [-c {llvm,rocm,cce}]
                    [-cv {llvm_14,llvm_15,llvm_16,rocm_4.5,rocm_5.0,rocm_5.2,cce_14.0.0,cce_14.0.1}]
                    [-omp {4.5,5.0,5.1,5.2}]
                    file
autosollvevv: error: argument -c/--compiler: invalid choice: 'gcc' (choose from 'llvm', 'rocm', 'cce')
```

Figure 3: Incorrect input demonstration.

The user of the tool can also make use of the '-h/--help' help command to understand in detail how to use the CLI tool and the optional parameters as seen in Figure 4.

```
[nikhilrao@login2.crusher ~]$ python3 autosollvevv -h
usage: autosollvevv [-h] [-c {llvm,rocm,cce}]
                    [-cv {llvm_14,llvm_15,llvm_16,rocm_4.5,rocm_5.0,rocm_5.2,cce_14.0.0,cce_14.0.1}]
                    [-omp {4.5,5.0,5.1,5.2}]
                    file

positional arguments:
  file                  This is to input the program file

optional arguments:
  -h, --help            show this help message and exit
  -c {llvm,rocm,cce}, --compiler {llvm,rocm,cce}
                        This is to specify the compiler to show all versions
  -cv {llvm_14,llvm_15,llvm_16,rocm_4.5,rocm_5.0,rocm_5.2,cce_14.0.0,cce_14.0.1}, --compilerversion {llvm_14,llvm_15,llvm_16,rocm_4.5,rocm_5.0,rocm
_5.2,cce_14.0.0,cce_14.0.1}
                        This is to specify the compiler with it's version
  -omp {4.5,5.0,5.1,5.2}, --openmp {4.5,5.0,5.1,5.2}
                        This is to specify the OpenMP Version
```

Figure 4: --help command.

1.3 Limitations

- Only a directives database is provided. This tool won't be able to verify a pragmas validity.
- Unlike on a local system the tooling cannot be called globally by the user on Crusher
    - That's the reason we move the executable of the tool to an addressable directory

Approaches

This section will elaborate on 3 approaches to build the autoSOLLVEvv tool. The approaches were built evaluated and updated before trying on the next approach for the tool. The current version of the tool that is deployed on OLCF's Crusher is based on Approach 3.

- Approach 1
- Approach 2
- Approach 3

Approach 1

1.1 Overview

The goal of what the program should achieve is to be something that understands the contents of an OpenMP pragma and then relates it to a test that has been conducted by the OpenMP V&V test suite.

1.2 Database

For approach #1 the most ideal way I could think of at the time to achieve the goal was to use JSON databases for the following

- All Directives (directives.json) [This database would contain the list of all directives with the clauses that can be used with them and their compatible OMP version.] Manual
- All Clauses (clasues.json) [This database would contain the list of all compilers with the options that can be used with them and their compatible OMP version.] Manual
- All Options (options.json) [This database would contain the list of all options with the modifiers that can be used with them and their compatible OMP version.] Manual
- All Modifiers (modifers.json) [This database would contain the list of all their compatible OMP version.]
- All Tests (test.json) [This database would contain the directive, clauses, options, and modifiers used in a test with the test names.] Manual
- Results (results.json) [This database would contain the result of whether the test has failed or passes along with other details.] Automated

OpenMP 5.1 was taken as a starting point as the number of tests that were written for it was smaller than the other OMP versions.

- Directives JSON database was created with this structure.

```
{
  "directive": "atomic",
  "implementation":"",
  "updation":"",
  "clauses":
    {
      "0":"capture",
      "1":"compare",
      "2":"hint",
      "3":"fail",
      "4":"weak"
    }
},
```


Figure 5: Approach 1 directives.json.

- Clauses JSON database was created with this structure.


```
{
  "clause":"defaultmap",
  "implementation":"",
  "updation":"",
  "options":
    {
      "0":"alloc",
      "1":"to",
      "2":"from",
      "3":"tofrom",
      "4":"firstprivate",
      "5":"none",
      "6":"default",
      "7":"present"
    }
},
```


Figure 6: Approach 1 clauses.json.

- Options JSON database was created with this structure.

```
{
  "option":"tofrom",
  "implementation":"",
  "updation":"",
  "modifiers":
    {
      "0":"errors",
      "1":"aggregate",
      "2":"pointer"
    }
},
```

Figure 7: Approach 1 options.json.

- Modifiers JSON database wasn't created as its goal was for OMP version control. As OMP 5.1 was the only version that these databases were created for, this database would be redundant.

- Test JSON database was created with multiple versions of structures.
    - version 1 [Naming scheme was incorrect.]

```
{
  "construct": "target",
  "directive": "NULL",
  "clause": "defaultmap",
  "conditioner":"present",
  "modifier":"NULL",
  "name": "test_target_defaultmap_present.c"
},
```

Figure 8: Approach 1 test.json version 1.

    - version 2 [Corrected Naming scheme, reduced the size of each block of data.]

```
{
  "directive": "target",
  "clause": "defaultmap",
  "conditioner":"present",
  "name": "test_target_defaultmap_present.c"
},
```

Figure 9: Approach 1 test.json version 2.

1.3 Program

1.3.1.1 Type A – Input pragma statement.

This program will take input of an OpenMP pragma from the user and will provide the user with the result of a test that's related to the pragma.

- Input: omp pragma statement.
- Output: Result of test related to the user input that has passed or failed.

1.3.1.2 Program Flowchart

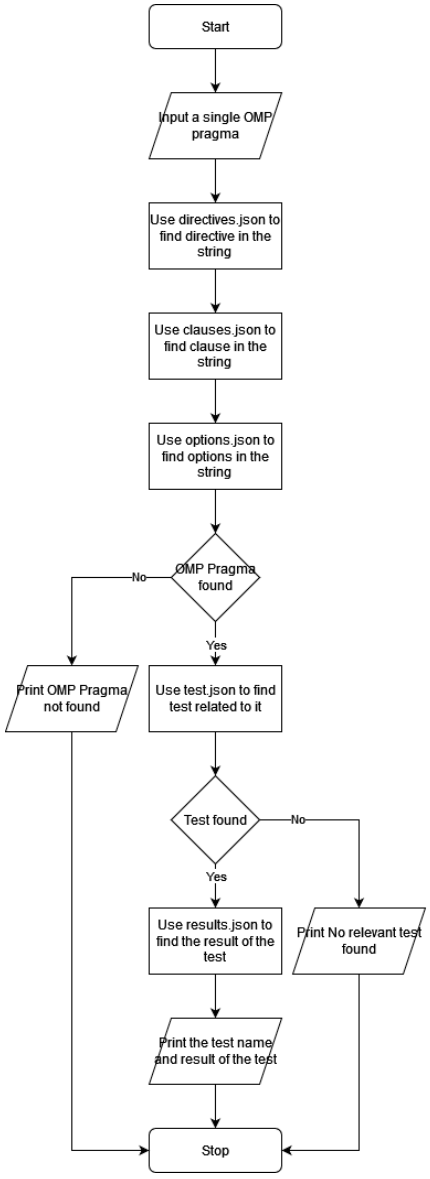


Figure 10: Approach 1 Type A process flow.

1.3.1.3 Program Output

```
nikhil@ubuntu:~/Desktop/SOLLVE$ python3 SOLLVE_auto_final_show.py
#pragma omp target update to
['#pragma', 'omp', 'target', 'update', 'to']

Construct: target update to
Test Name: test_target_update_to_present.c
Compiler Name: clang 14.0.0 (https://github.com/llvm/llvm-project.git 36892727e4f19a60778e371d78f8fb09d8122c85) Result: PASS
Compiler Name: gcc 11.1.0 Result: FAIL
Compiler Name: nvc version unknown Result: FAIL
```

Figure 11: Screenshot of the Output of Approach 1 Type A where the user inputs an entire pragma

1.3.2 Type B – User selects from a list.

This program presents directives, clauses and options to the user and then gives the results of the test of the corresponding result. This Program was created just as an exploratory exercise as it's easier for a user to divide up the OMP pragma than for a program to do it.

- Input: User choice.
- Output: Test result.

1.3.2.1 Program Flowchart

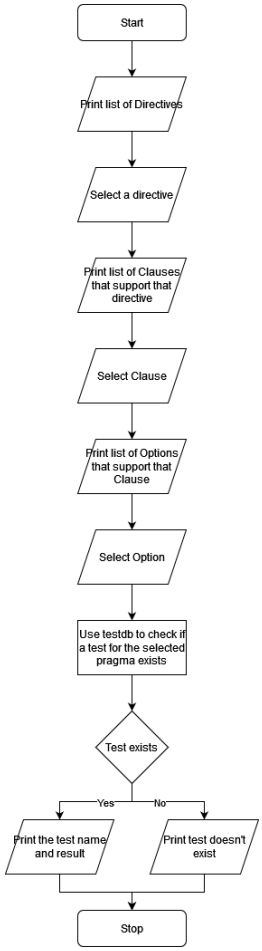


Figure 12: Approach 1 Type B process flow.

1.3.2.2 Program Output

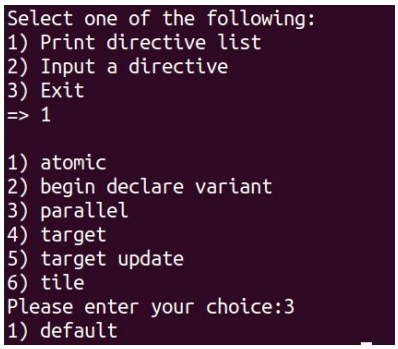


Figure 13: Screenshot of the Output of Approach 1 Type B where the user selects to create the pragma.

1.4 Coding Obstacles

- Writing a program that understood what an OMP pragma is and its structure.
- Creating the JSON files for the OMP spec.
- Linking the test name to an OMP pragma.

1.5 Program Limitations

- Excess manual labor for building the databases.
- Only finds tests that exactly fit the omp pragma.
- Overly complicated JSON database structure.
- Inputting a single pragma is impractical for a user.

Approach 2

2.1 Overview

After a thorough discussion with SOLLVE collaborators, the following conclusions were made:-

- The best course of action was decided to be to use the OpenMP JSON spec from OpenMP's GitHub.
- It was also decided to drop the alternative type. While it does serve a purpose, it isn't in the scope of the result.
- Looking into CLANG AST to gather information about the OMP calls to know what pragmas were being used.

2.2 Databases

2.2.1 Changes Made

- Removed directives JSON database.

- Removed clauses JSON database.

- Removed options JSON database.

- Dropped program alternative version.

+ Modified tests JSON database.

version 3 [Added points for multiple options and modifiers under clauses.]

```
{
  "directive": "target",
  "clause": "defaultmap",
  "conditioner":
    {
      "0":"present"
    },
  "name": "test_target_defaultmap_present.c"
},
```

Figure 14: Approach 2 test.json version 3.

+ Added OMP 5.2 Specification JSON files.

The confusing structure of JSON files is used. An example of a single directive JSON file is listed below.

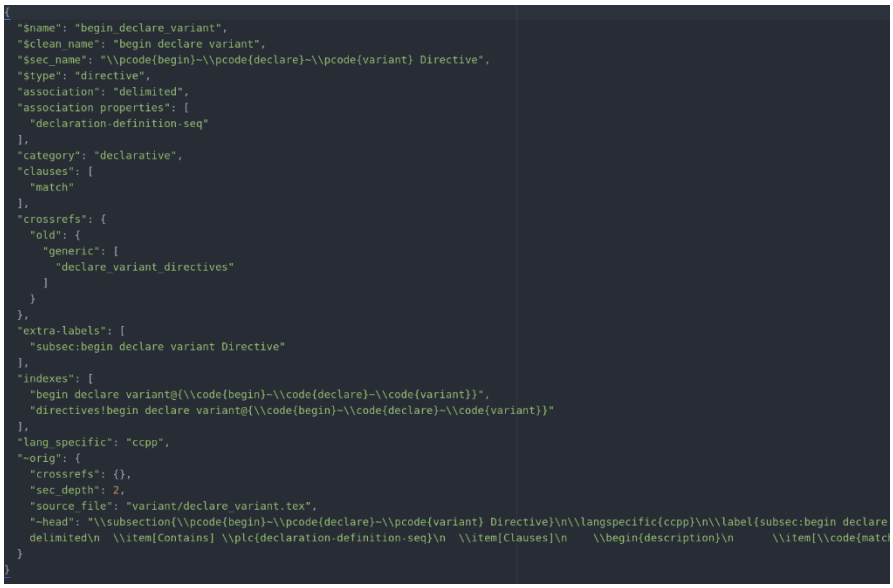


Figure 15: OMP 5.2 specification JSON snippet.

2.3 Program

2.3.1 Changes Made

- Removed support for older JSON files.

+ Rewrote how the program understands an OMP pragma using the OMP JSON spec.

+ Added support for tests with multiple options and modifiers.

+ Added support for OMP env calls.

2.3.2 Overview

This program will take input of an OMP pragma from the user and will provide the user with the result of tests that are related to the pragma.

- Input: omp pragma statement.
- Output: Results of tests related to the input that have passed or failed.

2.3.3 Program Flowchart

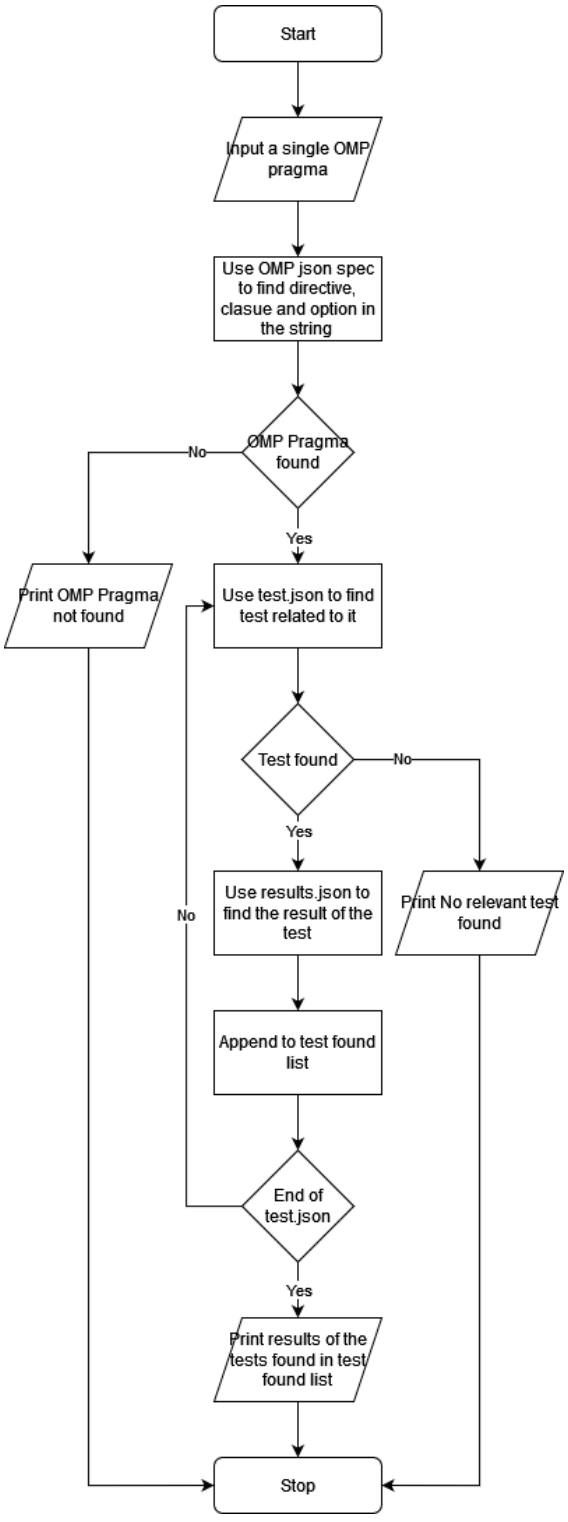


Figure 16: Approach 2 process flow.

2.4 Coding Obstacles

- Rewriting a program that understood what an OMP pragma is and its structure using the OMP specification JSON.
- CLANG AST was investigated as a means of reading user data but directly reading a user code file was much easier as we're already parsing single OMP strings.
- Creating a function to mix and match the clauses and their options to find all tests that relate to the given pragma.

2.5 Program Limitations

- OMP JSON spec is not written straightforwardly.
    - Missing Directives, Clauses.
    - A large number of JSON files must be parsed though as each directive and clause has a JSON file.
- Overly complicated JSON database structure.
- Inputting a single pragma is impractical for a user.
- Still can't read a code file and pick out pragmas.
- Each version of OMP will have its list of a hundred or so JSON spec files with slightly different structures. Incorporating all will be a waste of space and time.

Approach 3

3.1 Overview

After taking into consideration all the limitations and issues with Version #2. The most straightforward solution was to drop the OMP JSON spec and generate only the absolute minimum databases for the program to function.

3.2 Databases

3.2.1 Changes Made

- Removed OMP specification JSON files.

+ Created new directives database with this structure. Made in excel and exported to JSON.


```
{
  "Directive": "target teams distribute simd",
  "4.5": 1,
  "5.0": 1,
  "5.1": 1
},
{
  "Directive": "target teams loop",
  "4.5": 0,
  "5.0": 1,
  "5.1": 1
},
{
  "Directive": "target update",
  "4.5": 1,
  "5.0": 1,
  "5.1": 1
},
```


Figure 17: Approach 3 directives.json.

+ Created a new test database with this structure. Made in excel and exported to JSON.


```
{
  "OMP Version": 5.1,
  "Test Name": "test_omp_display_env.c",
  "Pragma": "test omp_display_env()"
},
{
  "OMP Version": 5.1,
  "Test Name": "test_target_defaultmap_present_scalar.c",
  "Pragma": "#pragma omp target defaultmap(scalar)"
},
{
  "OMP Version": 5.1,
  "Test Name": "test_target_defaultmap_present.c",
  "Pragma": "#pragma omp target defaultmap(present)"
},
{
  "OMP Version": 5.1,
  "Test Name": "test_target_has_device_addr.c",
  "Pragma": "#pragma omp target map"
},
```


Figure 18: Approach 3 test.json.

3.3

3.4 Program

3.4.1 Version 1.1

3.4.1.1 Changes Made

- Removed support for OMP specification JSON files.

+ Split the single code block into multiple functions.

+ Added support for new directives database.

+ Added logic of what qualifies as a clause.

+ Added logic of what qualifies as a conditioner (Option + Modifier/s).

+ Modified support for tests with multiple options and modifiers.

+ Added support for OMP runtime library calls.

+ Added support to input files and parse through them.

+ Reduced dependencies by using native python libraries as alternatives.

+ Rewrote how test.json using a function.

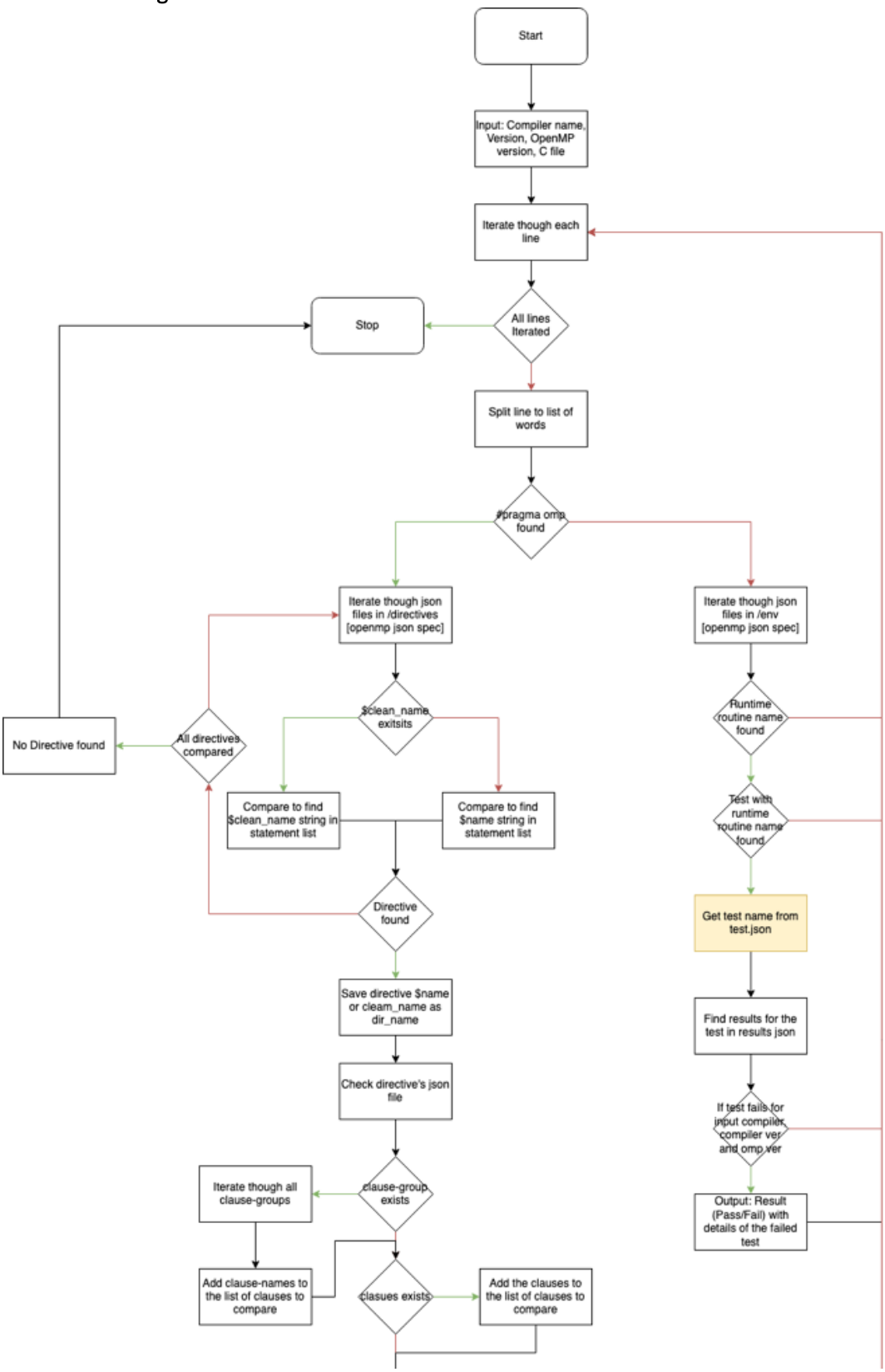
+ Added support for feedback to gain statistical knowledge of what's being tested is being triggered often.

3.4.1.2 Overview

This program will take an input of a c/cpp file from the user and will provide the user with the result of tests that are related to each pragma that the program finds in the code file provided.

- Input: Code file.
- Output: Results of tests that have failed for the OMP pragmas in the code file.

3.4.1.3 Program Flowchart



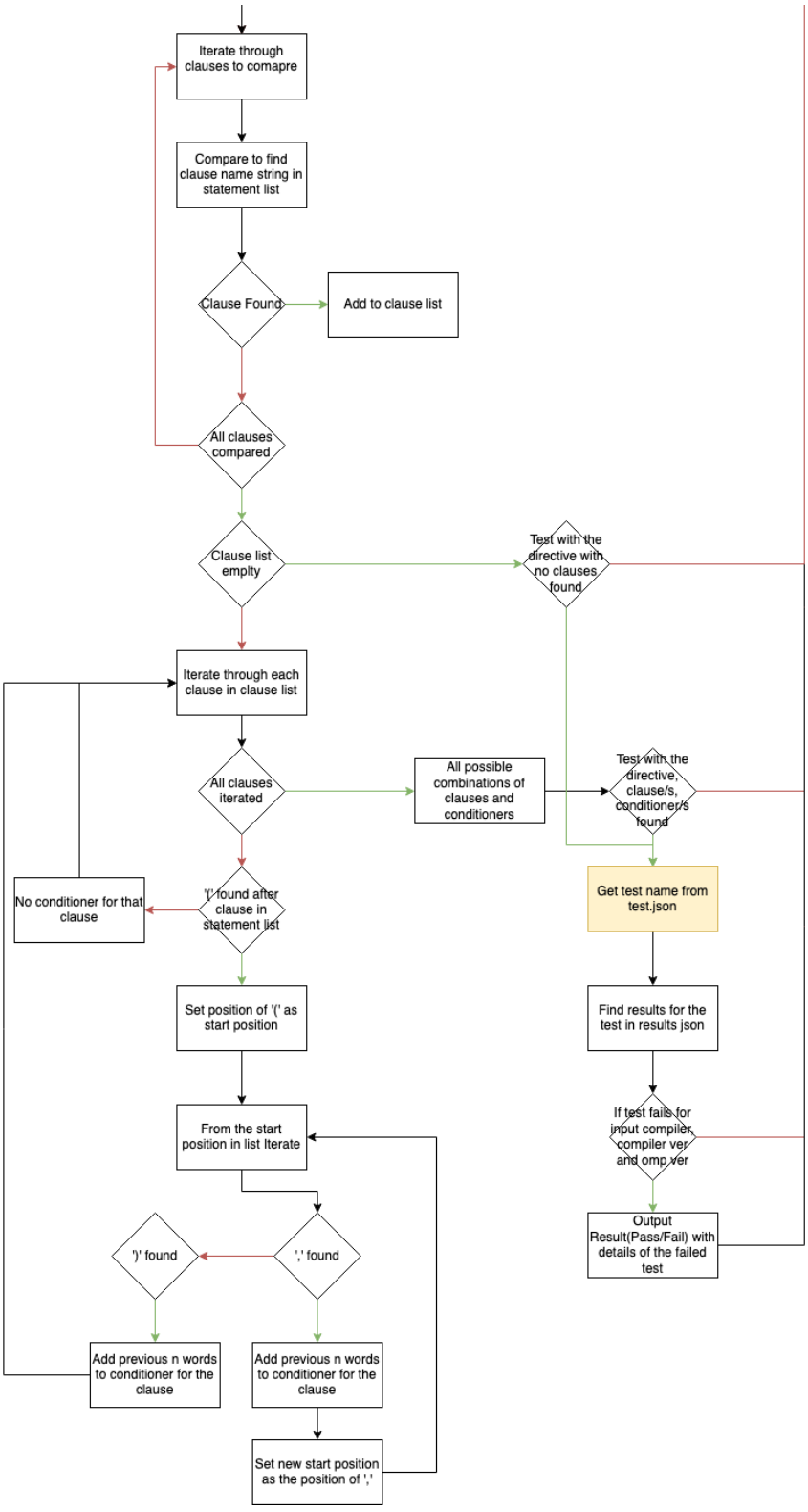


Figure 19: Approach 3 Version 1.1 process flow.

3.4.1.4 Program Output

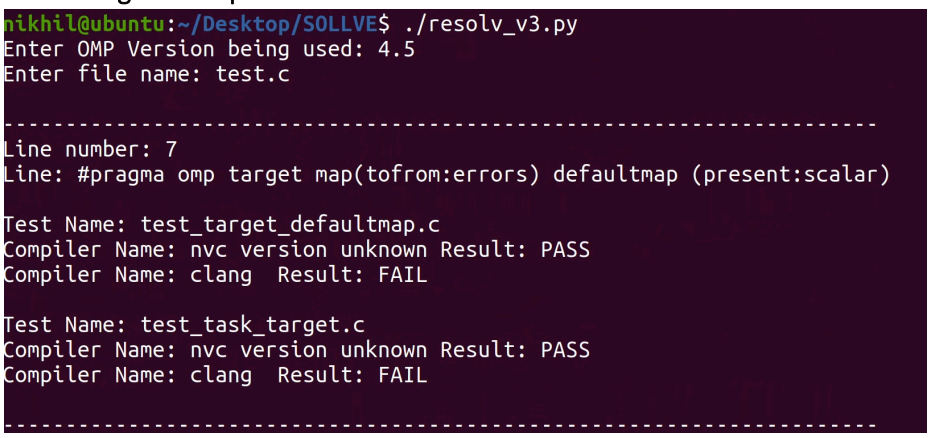


Figure 20: Screenshot of the output of Approach 3 Version 1.1.

3.4.2 Version 1.2 [UNDER DEVELOPMENT]

3.4.2.1 Changes Made

+ Added wrapper to turn it into a CLI tool.

+ Added arguments for the tool.

+ Added support for multiple conditions.

+ Added help argument.

+ Added better interface.

+ Added Fortran support.

3.4.2.2 Overview

This program was built on top of version 1.1 and is a python command line tool that can be installed using pip. It will take multiple arguments including a c/cpp/Fortran file, OpenMP version, compiler, and compiler version from the user and will provide the user with the result of tests relevant to the arguments provided that are related to each pragma that the program finds in the code file provided.

- Input: Code file, -c, -cv, -omp, -h.
- Output: Results of tests that have failed for the OMP pragmas in the code file.

3.4.2.3 Program Flowchart

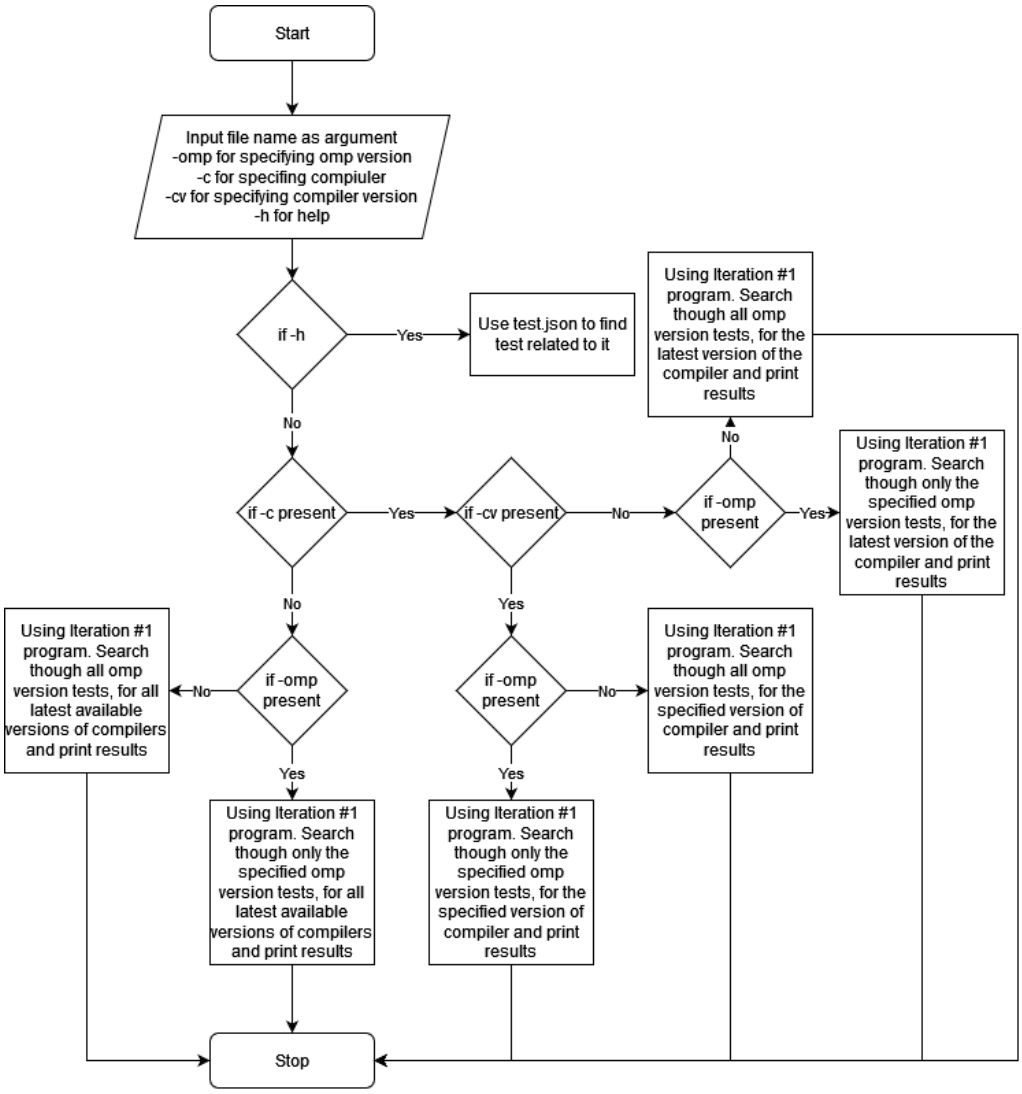


Figure 21: Approach 3 Version 1.2 process flow.

3.4.2.4 Program Output

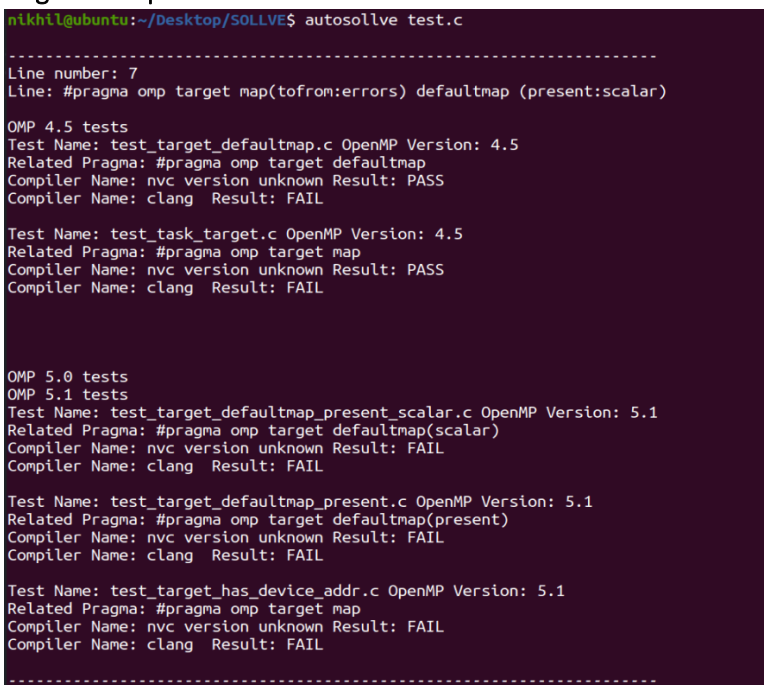


Figure 22: Screenshot of the output of Approach 3 Version 1.2.

3.5 Coding Obstacles

- Rewriting a program that understood what an OMP pragma is by using syntax as the primary guide.
- Packaging the script and dependencies into a single package.

3.6 Program Limitations

- Only a directives database is provided. This program won't be able to verify a pragmas validity.
- The feedback system will be dependent on the architecture of the system it's being deployed on.