Center for Embedded and Cyber-Physical Systems
University of California, Irvine

A Tool to Flatten Multi-File SystemC Models for the RISC compiler

Yutong Wang, Rainer Dömer

Technical Report CECS TR 21-01
March 1, 2021

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

yutongw5@uci.edu
http://www.cecs.uci.edu

A Tool to Flatten Multi-File SystemC Models for the RISC compiler

Yutong Wang, Rainer Dömer

Technical Report CECS TR 21-01
March 1, 2021

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

yutongw5@uci.edu
http://www.cecs.uci.edu

Abstract

*The Recoding Infrastructure for SystemC (RISC) uses a dedicated SystemC compiler for aggressive yet standard compliant parallel simulation capable of out-of-order execution on many-core platforms. One of the known shortcomings of the current RISC compiler version 0.6.2 is handling multiple source files. While having separated source files is fairly standard practice, when modeling complex embedded systems, the compiler needs to combine all source files in the correct order to function properly. In this report, we introduce a tool **flatten** which solves the multiple source file problem by examining the inclusion hierarchy of the source files and combining all sources into one flattened file. This report describes the development stages, the usage as well as the future development goal of **flatten**. This report also includes several representative examples and shows how **flatten** handles them.*

Contents

A Tool to Flatten Multi-File SystemC Models for the RISC compiler

Y. Wang, R. Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
yutongw5@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

The Recoding Infrastructure for SystemC (RISC) uses a dedicated SystemC compiler for aggressive yet standard compliant parallel simulation capable of out-of-order execution on many-core platforms. One of the known shortcomings of the current RISC compiler version 0.6.2 is handling multiple source files. While having separated source files is fairly standard practice, when modeling complex embedded systems, the compiler needs to combine all source files in the correct order to function properly. In this report, we introduce a tool ***flatten*** *which solves the multiple source file problem by examining the inclusion hierarchy of the source files and combining all sources into one flattened file. This report describes the development stages, the usage as well as the future development goal of* ***flatten****. This report also includes several representative examples and shows how* ***flatten*** *handles them.*

1 Introduction

Flatten is developed using Python3 [1] as one of the pre-built tools of the RISC [2][3] version 0.6.3 release. **Flatten** is designed to solve one of the known limitations of RISC compiler: it does not work well with separate source files. Prior to the release of the script, RISC users need to manually combine their source files in the correct order. While it is easy to do for simple SystemC designs, large designs with numbers of source files and complex file hierarchies are a burden for system designers to flatten manually. Although Python3 is needed in the user's system, **flatten** is overall easy to setup and use. Installing the RISC compiler will automatically configure this tool, the user can then simply call **flatten** command to execute the script.

1.1 Problem definition

As mentioned in the abstract, RISC uses a custom SystemC [4] compiler for aggressive yet standard compliant parallel simulation on multi-core systems to speed up otherwise slow SystemC compilation. One of the known limitations that **flatten** is aiming to bypass is introduced in the version 0.4.0 technical report of RISC (see section 3.4 for more detailed explanation) [5]. The RISC shortcoming is related to source code instrumentation: the RISC compiler only works if it has access to the entire source code of the design model. While RISC compiler can deal with smaller SystemC projects which usually have simpler file hierarchies and small number of libraries and source files, it has difficulty to compile complex SystemC models.

1.2 Related work

RISC compiler version 0.5.0 [6] introduced a different solution than **flatten**, which is Partial Segment Graph (PSG) [7]. PSG is implemented to solve the aforementioned limitation on multi-file inputs and hierarchical file structures by representing the behavior model of each separate translation unit (multiple source files). PSG can then be combined to construct a complete Segment Graph (SG) for the input model. For detailed information on how RISC uses SG to construct the model, please refer to the RISC 0.5.0 report [6]. With the help of PSG, RISC compiler is able to properly compile the model without the need to manually edit each source file. However, in order to deal with uncertainty in the abstract syntax tree (AST), PSGs are separated into three types and they are constructed by the Intellectual Property (IP) provider [6]. In the case of missing PSG support, the RISC compiler faces the same limitation introduced in version 0.4.0..

Using PSG is rather complex and cumbersome to handle for the designer. **Flatten** on the other hand, offers a simpler, easy-to-use solution that supports a range of SystemC file extensions and different file hierarchies.

2 Features

Flatten has many built-in features including support for multiple SystemC file extensions, automatic main file detection and detecting of file hierarchy. More detail can be found later in this report. A list of arguments is also included in this report as well as examples that demonstrate all features. The current version of **flatten** is capable of combining SystemC source files in the correct order with simple commands. The output file can be directly compiled with regular g++ compiler or RISC, assuming source files have no compatibility issue with each compiler.

2.1 Compatibility with multiple SystemC file extension

SystemC source files have many file extensions. Suffixes ".cc", ".cpp", ".h", ".hpp", and ".C" are all supported by a SystemC compiler and therefore **flatten** needs to be compatible with all of the aforementioned file extensions. To achieve that, **flatten** does not use hard-coded file extensions to detect files. Instead, it will read the user specified input file and use the file's inclusion statements to determine the correct file extension for the source files. An example with multiple file extensions is shown below in Figure 1.

```
[user]:> flatten -i testfile1.cc
User input name for main, the path is:  /testfolder/testfile1.cc
---DONE FLATTENING---
[user]:> flatten -i testfile1.C
User input name for main, the path is:  /testfolder/testfile1.C
---DONE FLATTENING---
[user]:> flatten -i testfile1.cpp
User input name for main, the path is:  /testfolder/testfile1.cpp
---DONE FLATTENING---
```

Figure 1: **Flatten**0 can take multiple SystemC suffixes

2.2 Separation of library inclusion and user file inclusion

When **flatten** combines multiple source files, it will scan for preprocessor included directives and then compare the inclusion statement with the collected file information to determine whether the included header file or source file is a standard library or a user file. In the case of a C/C++ standard library file, all of such

inclusion statements from different files will be moved to the top of the output file. In the case of user files, such inclusion statements will be taken out.
An example with different inclusion statements is shown below in Figure 2.

```
//testfile1.cc
#include <iostream> \\standard library
#include <testfile2.cc>
#include <anyuserheader.h>
[code in testfile1.cc]

//output.cc
#include <iostream>\\standard library
[code in anyuserheader.h]
[code in testfile2.cc]
[code in testfile1.cc]
```

Figure 2: **Flatten** will take out user file inclusion and keep system library at top

2.3 Detail display and debug mode

Detail display and debug mode were added in case of the output file is not compilable or incorrectly flattened, **flatten** uses arguments *-v* and *-d* to play detailed and debug information respectively. At its current development stage, the *-v* option displays a list of included system libraries, a list of included SystemC header files and a list of SystemC implementation files. The *-d* option displays internal data structure used to store file hierarchical information, internal flags and path variables in addition to what *-v* displays. While the debug mode will give more information to help identify problems, turning it on sometimes displays a large amount of text for a large project. Therefore it is recommended to use the graphic option plus detail display mode (*-g* and *-v*) to view a more intuitive representation for the inclusion hierarchy of the user's project.
An example with debug mode on is shown below in Figure 3.

```
[user]:> flatten -i testfile1.cc -v
User input name for main, the path is:  /testfolder/testfile1.cc
SYSTEM_INCLUSION:{ #include <systeminclusion>}
H_FILES:'headerfiles1.h','headerfiles2.h'
C_FILES:'testfile1.cc','testfile2.cc'
---DONE FLATTENING---

[user]:> flatten -i testfile1.cc -d
User input name for main, the path is:  /testfolder/testfile1.cc
FOUND USER INPUT!
HAS_MAIN = True
PATH2MAIN = /testfolder/testfile1.cc/
PATH2FOLDER = /testfolder/
SYSTEM_INCLUSION:{ #include <systeminclusion>}
H_FILES:'headerfiles1.h','headerfiles2.h'
C_FILES:'testfile1.cc','testfile2.cc'
[internal data structure]
testfile1.cc level 0 Parent:  None
testfile2.cc level 1 Parent:  testfile1.cc
...
---DONE FLATTENING---
```

Figure 3: **Flatten** script with detail display mode on and debug mode on

2.4 Draw inclusion hierarchy

Flatten will read all files under the user current path using the Python3's OS library, then build a list of each file and path to get to them. From each file and their inclusions, **flatten** can then build a directed graph which the nodes are files and edges are the inclusion relationship. From there, **flatten** will run a modified version of depth first search algorithm and generate a tree of files. Then the script reverse traverses the tree to print out the files in the correct order. By using the *-g* argument, the script will display the depth and inclusion information of the files. An example with the graph option on is shown below in Figure 4.

```
[user]:> flatten -i testfile1.cc -g
User input name for main, the path is:  /testfolder/testfile1.cc
---DONE FLATTENING---
-------GRAPH-------
NOTICE: ALL the C files are still printed in the correct order
though they are not shown in the graph

|--------testfile1.cc
    |--------testfile1.h
        |--------testfile2.h
        |--------testfile3.h
             |--------testfile4.h
             |--------testfile5.h
```

Figure 4: **Flatten** script with *-g* displays the file hierarchy in shell

3 Command Line Options

Flatten is a tool designed to generate a compilable file for the RISC compiler. It takes multiple source files from the user based on the inclusion in each file and outputs a single combined source file in the correct order.

Flatten includes 7 command line options to provide necessary functionalities and to offer better user experience. These command line options include a help page, specification of input and output file names, showing detailed information, displaying debug information and graph for the inclusion hierarchy, and option to execute an auto-compile command after flatten, which is currently under development.
Detailed explanation and example is shown below.

3.1 Available Arguments

1. **No arguments**: If no argument is provided, the script will scan each file and sub-directory in the current path for a SystemC sc_main function to identify the main file. If the main file is found, then the script would execute without any arguments automatically. When no main file is found, the script would display the following message and then asks for a manual input.

   ```
   ERROR: input file not found, exiting....
   ```

   In the case of finding multiple sc_main function, which is possible since one large project might contain multiple independent models, the script would display the following message to notify the user to switch to manual input instead.

   ```
   ----Found multiple files containting name main or Main,
   exiting....Please use manual input----
   ```

2. **Display help using *-h* argument**: user can add this argument to see a full list of available arguments and a line of brief explanation for each argument. The script will only display help messages upon seeing this argument and ignores the rest of the input. This is implemented to protect user from unintentional use and accidentally overwrite important files. The help message is shown below and is also included in the Appendix A at the end of this report.
   An example is shown below:

   ```
   >flatten -h
   User Help:
   use -v argument to show included files
   use -g argument to show graph(tree) of file dependency
   use -x argument to auto compile the flattened file using RISC
   use -i [input] argument to indicate the main file, no need to input
   multiple files
   use -o [output] argument to change the name of output file, if not
   default to Main_flat.cpp
   use -d argument to enter debug mode
   ```

3. **Display detailed information using *-v* argument**: user can add this argument to see detailed information after flatten is executed. Detailed information includes: a list of system inclusions, a list of included SystemC header files and a list of included SystemC implementation files. This argument is useful for users who want to check whether all necessary files are successfully included and it is best paired with the *-g* option to get a clear understanding of the file inclusions.
   An example is shown below:

```
>flatten -v
SYSTEM_INCLUSION: [systemincludeions]
H_FILES: [headerfiles]
C_FILES: [implementation files]
```

4. **Display a text based graph using *-g* argument**: user can add this argument to see a tree of included files using characters. The graph contains information including the root of the graph (usually the main file), the depth information of the included file, and parent information of each included file. The depth information indicates the distance from the root file to that file and parent information indicates where that file is referred from. In the case of a large and complex SystemC model, the tree could get very large and it is better to pair this argument with the *-v* argument to make sure that the necessary files are included and flattened correctly.
   An example is shown below:

```
>flatten -g
-----GRAPH-----
|--------Main
    |--------first_level_file
             |--------second_level_files
             |--------second_level_files
             |--------second_level_files
             |--------second_level_files
                      |--------third_level_files
                      |--------third_level_files
```

5. **Use argument *-x* to compile flattened file**: user can use this argument to compile and execute the flattened file immediately after flatten. Since the RISC compiler may not support all SystemC and custom libraries and user might also add custom flags while compiling, this argument is currently disabled in the script and is part of the future development plan of **flatten**.

6. **Specify input file name using *-i [input]***: user can use this argument to indicate the name of the main file. This argument is not required to use the script but it is recommended, since the script could detect multiple main files by accident or detect no main files at all. Notice that to use this argument the user needs to input the complete name of the file. In addition, it is not required to input the absolute path to the file.
   An example is given below:

```
>flatten -i custom_name.cc
-----DONE FLATTENING-----
```

7. **Specify output file name using *-o [output]***: user can use this argument to indicate the name of the output file. This argument is not required to use the script but it is recommended since the script by default write to *Main_flat* with the same file extension as the main file, it is possible to overwrite other documents.
   An example is given below:

```
>flatten -o custom_name.cc
-----DONE FLATTENING-----
```

8. **Display debug information using *-d* argument**: user can use this argument to see debug information. Similar to the *-v* argument for displaying detailed information, *-d* provides even more detailed information. In addition to a list of system inclusions, a list of included SystemC header files and a list of included SystemC implementation files, adding the debug argument will also make the script display internal flags, path variables, and the data structure used to store the graph. Notice that when the SystemC model is large and contains many source files, turning on debug mode might result in large amount of text, using the *-v* argument might be the better choice.
   An example is given below:

```
>flatten -d
HAS_MIN_FLAG = [T/F]
PATH_VARIABLES = [path_to_main/folder]
SYSTEM_INCLUSION: [systemincludeions]
H_FILES: [headerfiles]
C_FILES: [implementation files]
INTERNAL_DATA: [print_doubly_linked_list]
```

4 Experiments and Results

In Section 2 of this report, examples used are very ideal and standard, using such test bench and is not enough to cover many real-life scenarios. Therefore, this section will show few experiments on SystemC models from both industry and academia. These models are fair representations of real-life SystemC models even though some of the detailed information are hidden due to privacy restrictions.

4.1 Table of Examples

Table 1 below contains 8 examples from both industry and academia. These examples covers a broad range of scenarios with different number of files, inclusion depths, libraries used and file extensions. Although **flatten** supports many forms of input, it still has trouble handling certain type of projects. In addition, some examples can be flattened and compiled with RISC or g++ compiler successfully while others need to be modified in order to compile by either compiler. Details of each example are shown in Subsection 4.2.

List of Examples							
#	Example	Source	Number of Files	Flatten Result	Compile with g++	Compile with RISC	Comments
1	ProdsCons	Industry1	13	Success	No	No	Missing library on host system
2	SysC	Industry1	1	Success	Yes	No, needs more modifications	RISC error: cannot handle certain module
3	TBM	Industry1	Unknown	N/A	N/A	N/A	Provided files are encrypted
4	SystemC Tester	Industry2	11	Success	Yes	No, needs more modification	RISC error: segmentation fault
5	Jpeg_encoder	UCI	22	Success	Yes	Yes	No additional modification needed
6	canny	UCI	43	Success	Yes	Yes*	RISC executable failed to run correctly
7	skunk	UCI	5	Fail	N/A	N/A	Project uses python scripts to generate source code
8	png_encoder	UCI	14	Success	Yes*	No	minor modification: delete 2 lines to make g++ work

Table 1: Application examples and experimental results

4.2 Examples in Detail

1. **ProdsCons from Industry1**: Shell output is shown in Figure 5 below. The output file has been verified to contain all 13 files in 3 separate folders (source and header folder) and the reason g++ and risc compiler fail is due to one library file missing from the host machine. In other words, **flatten** did work properly and the output file should compile in machines with necessary packages installed. This project uses standard file extensions ".cpp" and ".h" which are supported by the tool. **Flatten** is also able handle multiple folder for libraries in different paths.

```
[user]:> flatten -i cfm_prod_cons_top.cpp -o test.cpp -v -g
User input name for main, the path is:  /ProdCons_forUCI/Prod
Cons_forUCI/risc _test/risc_test/gen-sysc/prod_cons/src/cfm_prod_
cons_top.cpp
SYSTEM_INCLUSION: {'#include "dt/cft_defchar.h\,'#include
"cofluent.h"}
H_FILES: ['cft_defchar.h','cfm_prod_cons_global.h',
'cfm_consumer.h', 'cfm_producer.h', 'cfm_prod_cons_user.h',
'cfm_prod_cons.h', 'cfm_prod_cons_top.h' ]
C_FILES: ['cfm_consumer.cpp', 'cfm_producer.cpp','cfm_prod_cons_use
r.cpp', 'cfm_prod_cons_user.cpp', 'cfm_prod_cons.cpp',
'cfm_prod_cons_top.cpp']
---DONE FLATTENING---
---------GRAPH----------
NOTICE: ALL the C files are still printed in the correct order
though they are not shown in the graph
|--------cfm_prod_cons_top.cpp
    |--------cfm_prod_cons_top.h
        |--------cfm_prod_cons_user.h
            |--------cfm_prod_cons_global.h
        |--------cfm_prod_cons.h
            |--------cfm_consumer.h
                |--------cft_defchar.h
            |--------cfm_producer.h
                |--------cft_defchar.h
            |--------cft_defchar.h
```

Figure 5: **Flatten** script output on Industry1 ProdsCons example with *-v* and *-g* flag

2. **SysC from Industry1**: Shell output is shown in Figure 6 below. Since the input file has only 1 file and it uses supported file extension, it is easy to verify the output. This project is an unusual scenario since it has only one source file, but **flatten** can handle this type of input with no issue. While the flattened output can be compiled with g++, RISC does not support one of the internal modules. Which will require manual modifications to work around RISC compiler limitations.

```
[user]:> flatten -i prodcons.cpp -o test.cpp -v -g
User input name for main, the path is:  SysC_forUCI/sc_prod_cons/
src/prodcons.cpp
SYSTEM_INCLUSION: {' #include <systemc.h>'}
H_FILES: []
C_FILES: ['prodcons.cpp']
---DONE FLATTENING---
---------GRAPH----------
NOTICE: ALL the C files are still printed in the correct order
though they are not shown in the graph
|--------prodcons.cpp
```

Figure 6: **Flatten** script output on Industry1 SysC example with *-v* and *-g* flag

3. **TBM from Industry1**: We were not able to test this example because the file archive is encrypted.

4. **SystemC Tester from Industry2**: The shell output is shown in Figure 7. The output file has been verified to contain all SystemC header files and implementation files in the correct order. In this example, the naming scheme for SystemC source files are fairly standard (".cpp" and ".hpp") and the inclusion hierarchy is also simple. While the g++ compiler is able to compile the flattened source file with no issue, modifications on the included libraries are needed for RISC compiler to work properly.

```
[user]:> flatten -i Main.cpp -o Main_flat.cpp -v -g
User input name for main, the path is:  /SystemC_Tester
/source/Main.cpp
SYSTEM_INCLUSION: {'#include "tlm.h"','#include
"tlm_utils/simple_initiator_socket.h"', '#include
"tlm_utils/simple_target_socket.h"','#include <stdint.h>'}
H_FILES: ['AddressMap.hpp', 'SimpleMem.hpp', 'SimpleTimer.hpp',
'SimpleTop.hpp', 'Interconnect.hpp']
C_FILES: ['SimpleMem.cpp', 'SimpleTimer.cpp', 'SimpleTop.cpp',
'Interconnect.cpp', 'Main.cpp']
---DONE FLATTENING---
---------GRAPH----------
NOTICE: ALL the C files are still printed in the correct order
though they are not shown in the graph
    |--------Main.cpp
        |--------SimpleMem.hpp
        |--------SimpleTimer.hpp
        |--------SimpleTop.hpp
        |--------Interconnect.hpp
            |--------AddressMap.hpp
```

Figure 7: **Flatten** script output on Industry2 SystemC_Tester example with *-v* and *-g* flag

5. **Jpeg Encoder Model from UCI**: The shell output is shown in Figure 8, part of the graph is omitted because the inclusion tree is too long to fit in a single figure. Output file has been verified to contain

all SystemC header files and implementation files in the correct order. In this example there are files with depth level of 6, which means the script successfully transversed the file hierarchy even though it is complicated. Both g++ compiler and RISC compiler compiled the flattened source code without error, the executables are also verified to be working correctly.

```
[user]:> flatten -i jpeg.cc -o Main_flat.cpp -v -g
User input name for main, the path is:  /jpeg_encoder/jpeg.cc
SYSTEM_INCLUSION: {'#include "systemc.h"'}
H_FILES: ['dct.h', 'quantize.h', 'zigzag.h', 'config.h',
'rgb2ycc.h', 'encoder.h', 'huffman.h', 'types.h', 'stimulus.h',
'dut.h', 'monitor.h', 'jpeg.h']
C_FILES: ['dct.cc', 'quantize.cc', 'zigzag.cc', 'rgb2ycc.cc',
'encoder.cc', 'huffman.cc', 'stimulus.cc', 'dut.cc',
'monitor.cc', 'jpeg.cc']
---DONE FLATTENING---
---------GRAPH----------
NOTICE: ALL the C files are still printed in the correct order
though they are not shown in the graph
    |--------jpeg.cc <-(None)
        |--------jpeg.h <-(jpeg.cc)
            |--------types.h <-(jpeg.h)
                |--------config.h <-(types.h)
            |--------stimulus.h <-(jpeg.h)
                |--------types.h <-(types.h)
                    |--------config.h <-(types.h)
            |--------dut.h <-(jpeg.h)
                |--------types.h <-(types.h)
......
                        |--------types.h <-(types.h)
                            |--------config.h <-(types.h)
                    |--------zigzag.h <-(encoder.h)
                        |--------types.h <-(types.h)
                            |--------config.h <-(types.h)
                |--------huffman.h <-(dut.h)
                    |--------types.h <-(types.h)
                        |--------config.h <-(types.h)
            |--------monitor.h <-(jpeg.h)
                    |--------types.h <-(types.h)
                        |--------config.h <-(types.h)
```

Figure 8: **Flatten** script output on Jpeg Encoder example with *-v* and *-g* flag

6. **Canny Model from UCI**: The shell output is shown in Figure 9, part of the graph is omitted because the inclusion tree is too long to fit in a single figure. Output file has been verified to contain all 43 SystemC header files and implementation files in the correct order. This example shows that **flatten** can handle the standard file extensions and large number of files. Both g++ and RISC compiler were able to compile the flattened file with no error message. However, while the executable from g++ compiler ran without any issue, the executable from RISC compile failed to run properly and had to

abort.

```
[user]:> flatten -i Main.cc -o test.cc -v -g
User input name for main, the path is:  /canny/Main.cc
SYSTEM_INCLUSION: {'#include "systemc.h"'}
H_FILES: ['coordinatorx.h', 'blurx.h', 'coordinatory.h',
'blury.h', 'blurx_par.h', 'blury_par.h', 'prep.h',
'derivative_x_y.h', 'magnitude_x_y.h', 'gaussian_smooth.h',
'mag_delta.h', 'non_max_supp.h', 'apply_hysteresis.h', 'datain.h',
'dut.h', 'dataout.h', 'config.h', 'stimulus.h', 'platform.h',
'monitor.h', 'types.h', 'top.h']
C_FILES: ['coordinatorx.cc', 'blurx.cc', 'blurx_par.cc',
'coordinatory.cc', 'blury.cc', 'blury_par.cc', 'blurx_par.cc',
'blury_par.cc', 'prep.cc', 'derivative_x_y.cc', 'magnitude_x_y.cc',
'gaussian_smooth.cc', 'mag_delta.cc', 'non_max_supp.cc',
'apply_hysteresis.cc', 'datain.cc', 'dut.cc', 'dataout.cc',
'stimulus.cc', 'platform.cc', 'monitor.cc', 'top.cc', 'Main.cc']
---DONE FLATTENING---
---------GRAPH----------
NOTICE: ALL the C files are still printed in the correct order
though they are not shown in the graph
    |--------Main.cc <-(None)
        |--------types.h <-(Main.cc)
            |--------config.h <-(types.h)
        |--------config.h <-(types.h)
        |--------top.h <-(Main.cc)
            |--------config.h <-(types.h)
            |--------types.h <-(top.h)
                |--------config.h <-(top.h)
            |--------stimulus.h <-(top.h)
                |--------config.h <-(top.h)
......
            |--------monitor.h <-(blurx_par.h)
                |--------config.h <-(blurx_par.h)
                |--------types.h <-(blurx_par.h)
                    |--------config.h <-(blurx_par.h)
```

Figure 9: **Flatten** script output on Canny example with *-v* and *-g* flag

7. **Skunk from UCI**: The script cannot flatten files from projects that utilize a python script to dynamically generate source files. Therefore the flattened output invalid and cannot be compiled by either compiler.

8. **PNG Encoder Model from UCI**: The shell output is shown in Figure 10. The output file has been verified to contain all SystemC header files and implementation files in the correct order. In this project, if define statement is used for system library inclusion which caused problem for g++ compiler. This issue was solved by deleting the unsupported package inclusions (2 lines of code in this case). The executable from g++ compiler was verified to run properly. RISC compiler on the other hand reported

"Segmentation fault" error message and failed to compile.

```
[user]:> flatten -i png.cc -o test.cc -v -g
User input name for main, the path is:  png_encoder/png.cc
SYSTEM_INCLUSION: {'#include <stdio.h>', '# include <fcntl.h>',
'#include "zlib.h"', '#include "systemc.h"', '#include <string.h>
// for strlen', '# include <io.h>', '#include <assert.h>'}
H_FILES: ['config.h', 'types.h', 'stimulus.h', 'monitor.h',
'addfilter.h', 'upfilter.h', 'paeth.h', 'png.h']
C_FILES: ['stimulus.cc', 'monitor.cc', 'addfilter.cc',
'upfilter.cc', 'paeth.cc', 'png.cc']
---DONE FLATTENING---
---------GRAPH----------
NOTICE: ALL the C files are still printed in the correct order
though they are not shown in the graph
    |--------png.cc <-(None)
        |--------png.h <-(png.cc)
            |--------types.h <-(png.h)
                |--------config.h <-(png.h)
            |--------stimulus.h <-(png.h)
                |--------types.h <-(png.h)
                    |--------config.h <-(png.h)
            |--------monitor.h <-(png.h)
                |--------types.h <-(png.h)
                    |--------config.h <-(png.h)
            |--------addfilter.h <-(png.h)
                |--------types.h <-(png.h)
                    |--------config.h <-(png.h)
            |--------upfilter.h <-(png.h)
                |--------types.h <-(png.h)
                    |--------config.h <-(png.h)
            |--------paeth.h <-(png.h)
                |--------types.h <-(png.h)
                    |--------config.h <-(png.h)
```

Figure 10: **Flatten** script output on png_encoder from UCI example with *-v* and *-g* flag

5 Conclusion and Future Work

Flatten at its current state is fully functional and integrated with the upcoming RISC release (version 0.6.3) and contains a test function in the Makefile. In terms of future development, other functionalities, like automatic compile after flatten, are in consideration.

References

[1] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[2] Z. Cheng D. Mendoza R. Dömer G. Liu, T. Schmidt. Risc compiler and simulator, release v0.6.0: Out-of-Order Parallel Simulatable SystemC Subset. Technical report.

[3] RISC open source. http://www.cecs.uci.edu/~doemer/risc.html.

[4] Open SystemC Initiative, http://www.systemc.org. *Functional Specification for SystemC 2.0*, 2000.

[5] Z. Cheng R. Dömer G. Liu, T. Schmidt. RISC Compiler and Simulator, Release V0.4.0: Out-of-Order Parallel Simulatable SystemC Subset. Technical report.

[6] Z. Cheng D. Mendoza R. Dömer G. Liu, T. Schmidt. RISC Compiler and Simulator, Release V0.5.0: Out-of-Order Parallel Simulatable SystemC Subset. Technical report.

[7] R. Dömer Z. Cheng, T. Schmidt. Enabling IP Reuse and Protection in Out-of-Order Parallel SystemC Simulation. Proceedings of the International Embedded Systems Symposium, Springer, Friedrichshafen, Germany, September 2019.

A Appendix

NAME

flatten – Script to combine multiple source files for RISC compiler

SYNOPSIS

flatten [*options*]

DESCRIPTION

flatten is a tool designed to generate a single compilable file for the Recoding Infrastructure for SystemC (RISC) compiler. **flatten** takes multiple source files from the user based on the include preprocessor directives in each file, and outputs a single combined source file with the included files inserted in correct order. **flatten** also supports displaying the graph of the file inclusion as well as auto-detecting the main file.

For example, to flatten and view the inclusion structure of multiple source files of a given design, use the following command:

flatten -i main.cc -o output.cc -g

flatten displays the inclusion tree as text in the terminal with each included filename and their parents.

ARGUMENTS

If no argument is provided, the **flatten** script will scan for a ”main” file; if not found or multiple files that contain ”main” are found, the script will terminate and ask for manual input.

OPTIONS

–h use this argument to print a brief message on the usage of the tool and quit

–v use this argument to print a list of included files

–g use this argument to print a graph (tree) of the dependencies

–x use this argument to auto-compile the flattened file using RISC

–i [input] use this argument to indicate the main file, no need to input multiple files

–o [output] use this argument to change the name of the output file; if omitted, this defaults to ”Main_flat.cpp”

–d use this argument to turn on debug mode, display important internal data

VERSION

flatten is release version 0.6.3.

AUTHORS

Yutong (Tom) Wang <yutongw5@uci.edu>

COPYRIGHT

LICENSE

BUGS, LIMITATIONS

This is an academic proof-of-concept prototype implementation, not commercial-quality software.