Netlist Analysis Tools

NETA: NETLIST ANALYSIS TOOLSET

VERSION [0.3.2]

MARCH 19, 2018

Presented by
Travis Meade
Shaojie Zhang
Yier Jin

https://github.com/jinyier/NetA

Table of Contents

NETA Installation/Uninstalling

NETA Library Format

The Tools

RELIC

REFSM

REPATH

REBUS

REPCA

Other Tools

Tarjan's SCC Algorithm

REDPEN

Case Study: Working HARPOON

NETA Installation/Uninstalling

To install in the base directory (the directory with the “`Makefile`”) run the command

```
> make
```

After installation a series of bin files will be created in the various tool directories, “`logic_extraction`”, “`logic_identification`”, “`misc_tools`”, and “`word_partitioning`”, quotes for carity.

To uninstall the tools, simply run the command,

```
> make clean
```

in the same directory.

NETA Library Format

By default the tools of the NETA repository reads in netlist using the .bench file format. In order to use a different cell library for netlist format the cells must be passed into the file using the LIBRARY_FLAG, "`--lib`" quotes for clarity, in the program's arguments. As an example you could run the following command from the base directory,

```
> ./logic_extraction/bin/refsm --net data/netlists/uart.v --lib
data/libraries/typical_xor.rlb --word data/words/uart_word.in
```

The Reverse Engineering Library (RLB) format follows a very simple grammar that can make readability somewhat difficult. Each token in the RLB is separated by white space, which means that no token of the RLB file can contain white space (e.g. "AND size 2", quotes for clarity, would be an invalid cell name). The only non-white space characters should be the cell descriptions in the following format,

```
<cell_name>
<input_size> <internal_size> <output_size>
<input_1_name> <input_2_name> ... <input_<input_size>_name>
<internal_pin_1_description>
<internal_pin_2_description>
.
.
.
<internal_pin_<internal_size>_description>
<output_pin_1_descriptions>
<output_pin_2_descriptions>
.
.
.
<output_pin_<output_size>_descriptions>
```

WARNING: The netlist parser will tokenize on " . ", " , ", " (", and ")" characters.

The internal pins should have the following description,

```
<AND_OR_XOR_symbols> <num_of_inputs>
<temporal_vs_spatial_symbol>
<complement_vs_original_1_symbol> <input_pin_1_id>
<complement_vs_original_2_symbol> <input_pin_2_id>
.
.
.
<complement_vs_original_<num_of_inputs>_symbol> <input_pin_<num_of_inputs>_id>
```

The `<AND_OR_XOR_symbols>` symbols consists of one of the following,

A	This denotes an "AND" type of pin, where in order for the pin's signal to be high each input signal specified in the internal pin description must be high.
O	This denotes an "OR" type of pin, where in order for the pin's signal to be low each input signal specified in the internal pin description must be low
X	This denotes an "XOR" type of pin, where in order for the pin's signal to be high an odd number of input pin's signals must be high, and every other pin must be low.
N	This denotes an untyped pin, where the resulting pin is the same as the input pin. This type should only be used with pins that have exactly one pin.

The `<temporal_vs_spatial_symbol>` consists of one of the following,

G	This denotes a normal "Gate" type of pin. This pin has its output updated when the inputs are updated enough to determine the gates behavior.
R	This denotes a "Register" type of pin. This pin type has its output updated **ON THE FOLLOWING CLOCK CYCLE** if the inputs were updated enough to determine the gates behavior.

The `<complement_vs_original_symbol>` consists of one of the following,

C	This denotes the given input's signal is inverted for the pin.
O	This denotes the given input's signal is left unchanged for the pin.

The output pins have the following description

```
<output_pin_name> <AND_OR_XOR_symbols> <num_of_inputs>
<temporal_vs_spatial_symbol>
<complement_vs_original_1_symbol> <input_pin_1_id>
<complement_vs_original_2_symbol> <input_pin_2_id>
.
.
.
<complement_vs_original_<num_of_inputs>_symbol> <input_pin_<num_of_inputs>_id>
```

The Tools

The first goal of the tool set is to help IP users analyze netlist for potentially malicious code. To aid users in this endeavor we, the authors of the toolset, tried to reduce the problem to that of high-level function recovery of netlists. This strategy allows us to off load part of the burden of Trojan detection to you, the netlist user.

RELIC 2

The first step to most netlist analysis techniques should fall to either netlist partitioning or logic identification. Knowing the logical components of a netlist is useful since it can allow for other analysis methods. RELIC 1 uses a heuristic based fan-in structure matching to determine the uniqueness of each signal in the netlist.

Conceptually logic wires would have structure that is unique, while data wires should have a fan-in structure similar to the fan-in structures of the other wires in the same word. This conjecture allows for the base of a method for finding logic wires. RELIC 1 tries to match the wires composing the fan-in of a given gate pair such that the sum of the scores of the matched wires is maximized. To prevent infinite recursion a depth is used to cut off the search.

```
> ./logic_identification/bin/relic --net <netlist_file_name>
```

RELIC 2 not only uses RELIC 1's method for find word similarity, but it also uses certain deeper fan-in information to better find outlying wire structure. RELIC 2 mergers the RELIC 1 information and deep wire information using Principle Component Analysis.

PCA Component Selection

Depending on the purpose different component selection might be more desirable. To help with this two parameters can be adjusted. The first is the number of components under analysis. To adjust this parameter pass the flag "--cmps" followed by the number of components to use. Additionally to remove some number of components before performing clustering methods the flag "--shft" can be used. This is useful if the user wants to not consider some of the higher represented components. By default the shift value is very large (i.e. 100) to focus on the least used components. This large shift tends to work well on netlists that have a large amount of data registers compared to logic registers (a ratio larger than 1000).

Raw Component Output

Sometimes if a user wants to perform additional analysis on the component data extracted the user can use the flag "--raw" to output the extracted component for the input to standard output (or a specified file using the "--out" flag). The resulting raw data will be unlabeled. Each signal vector will exist on its own line and the element for signal vector will be tab separated.

Buffer Inverter Behavior Modification

Typically inverters and buffers have their signals merged up into the signal of their next gate. This allows netlists that have a large fan-out of buffers or inverters to be treated as one signal instead of as a chain of multiple signals. This acts as a double edged sword since it might merge certain wires that could be conceptually separated by the number of gates to an original signal. To turn off this behavior modification one can include the flag "--buf".

Custom Feat Sets

To change the feats that are used the following flag and file name can be passed to the command line as arguments "--feat <feat_file_name>". The feat file should be of the following form.

```
<Feature Count>
<Feature Description 1>
<Feature Description 2>
.
```

```
.
.
<Feature Description <Feature Count>>
```

The type of features you can have is the following

Feature Type	Feature Description
AND	1 if the signal function is an AND and 0 otherwise
OR	1 if the signal function is an OR and 0 otherwise
XOR	1 if the signal function is an XOR and 0 otherwise
FANIN	The size of the fan in set at a particular level (determined by the number of prior FANOUT features)
FANOUT	The size of the fan out set at a particular level (determined by the number of prior FANOUT features)
INPUT	The minimum number of clock cycles for a primary input signal to affect the signal
OUTPUT	The minimum number of clock cycles for the signal to affect a primary output signal
REGOUT	The minimum number of gates from the signal to a register
REGIN	The minimum number of gates from a register to the signal
SELFAFFECT	1 if the signal can affect itself and 0 otherwise
OLDRELIC	A relic score from a particular register (chosen by the lowest relic score maximum) and the signal

Example

Here is two examples of the RS232 uart each with a different feature sets

```
            Example 1                         Example 2
Feature     20                                12
Set         AND                               OLDRELIC
            OR                                OLDRELIC
            XOR                               OLDRELIC
            REG                               OLDRELIC
            FANIN                             OLDRELIC
            FANIN                             OLDRELIC
            FANIN                             OLDRELIC
            FANOUT                            OLDRELIC
            FANOUT                            OLDRELIC
            FANOUT                            OLDRELIC
            INPUT                             OLDRELIC
            OUTPUT                            OLDRELIC
            REGOUT
            REGIN
            SELFAFFECT
            OLDRELIC
            OLDRELIC
            OLDRELIC
            OLDRELIC
            OLDRELIC
Output      ID :: Z-score :: Name             ID :: Z-score :: Name
            28 0.0168161 n124                 16 0.000195462 n102
            27 0.0296331 \iRECEIVER/rec_datSyncH  19 0.000262961 n121
            18 0.030971 n27                   48 0.000262961 n29
            51 0.0315628 n123                 29 0.000282949 n122
            49 0.0319609 n116                 15 0.000333641 n26
            22 0.0368101 n28                  18 0.000473886 n27
```

```
33 0.0436969 n104
15 0.0544032 n26
29 0.0822973 n122
55 0.145899 \iXMIT/bitCell_cntrH[0]
23 0.145963 n120
5 0.153631 n111
60 0.155739 rec_dataH_temp[7]
61 0.155739 rec_dataH_temp[6]
62 0.155739 rec_dataH_temp[5]
63 0.155739 rec_dataH_temp[4]
64 0.155739 rec_dataH_temp[3]
65 0.155739 rec_dataH_temp[2]
66 0.155739 rec_dataH_temp[1]
67 0.155739 rec_dataH_temp[0]
36 0.157482 n32
38 0.157482 n22
40 0.157482 n31
42 0.157482 n21
44 0.157482 n30
46 0.157482 n20
48 0.157482 n29
32 0.167276 \iRECEIVER/n44
19 0.171684 n121
52 0.186363 n24
16 0.186452 n102
6 0.229342 n110
7 0.229342 n109
8 0.229342 n108
9 0.229342 n107
10 0.229342 n106
11 0.229342 n105
31 0.233405 rec_dataH_rec[0]
35 0.233405 rec_dataH_rec[1]
37 0.233405 rec_dataH_rec[2]
39 0.233405 rec_dataH_rec[3]
41 0.233405 rec_dataH_rec[4]
43 0.233405 rec_dataH_rec[5]
45 0.233405 rec_dataH_rec[6]
47 0.233405 rec_dataH_rec[7]
59 0.240824 \iXMIT/bitCell_cntrH[2]
34 0.276677 n33
50 0.287846 n18
14 0.28867 n101
12 0.335134 n99
26 0.335134 n118
57 0.388862 \iXMIT/bitCell_cntrH[1]
21 0.518404 n125
58 0.577428 \iXMIT/bitCell_cntrH[3]
54 0.582897 n114
56 0.657045 \iXMIT/n67
17 0.699986 n98
25 0.715023 n119
24 1.45012 n19
13 1.45868 n100
53 1.57902 n115
```

```
47 0.000473886 rec_dataH_rec[7]
28 0.000595136 n124
60 0.000595136 rec_dataH_temp[7]
61 0.000595136 rec_dataH_temp[6]
62 0.000595136 rec_dataH_temp[5]
63 0.000595136 rec_dataH_temp[4]
64 0.000595136 rec_dataH_temp[3]
65 0.000595136 rec_dataH_temp[2]
66 0.000595136 rec_dataH_temp[1]
67 0.000595136 rec_dataH_temp[0]
22 0.00113291 n28
31 0.00115517 rec_dataH_rec[0]
35 0.00115517 rec_dataH_rec[1]
37 0.00115517 rec_dataH_rec[2]
39 0.00115517 rec_dataH_rec[3]
41 0.00115517 rec_dataH_rec[4]
43 0.00115517 rec_dataH_rec[5]
45 0.00115517 rec_dataH_rec[6]
21 0.00403696 n125
55 0.00471424 \iXMIT/bitCell_cntrH[0]
57 0.00471424 \iXMIT/bitCell_cntrH[1]
58 0.00471424 \iXMIT/bitCell_cntrH[3]
59 0.00471424 \iXMIT/bitCell_cntrH[2]
5 0.00635133 n111
6 0.00635133 n110
7 0.00635133 n109
8 0.00635133 n108
9 0.00635133 n107
10 0.00635133 n106
11 0.00635133 n105
32 0.00702847 \iRECEIVER/n44
36 0.00702847 n32
38 0.00702847 n22
40 0.00702847 n31
42 0.00702847 n21
44 0.00702847 n30
46 0.00702847 n20
34 0.00979758 n33
3 0.0101627 n17
14 0.014204 n101
23 0.0269908 n120
20 0.0480315 n23
17 0.0584265 n98
56 0.0584265 \iXMIT/n67
25 0.0641585 n119
12 0.0647744 n99
26 0.0647744 n118
13 0.1092 n100
53 0.11123 n115
49 0.113598 n116
4 0.166148 n113
50 0.213034 n18
51 0.22545 n123
54 0.22545 n114
52 0.294333 n24
```

```
3 3.46617 n17
4 5.9 n113
20 21.6284 n23
30 27.351 n117
0 30.5376 n103
2 36.3648 n112
1 61.2533 n25
```

```
24 0.31422 n19
0 6.20463 n103
30 17.1094 n117
33 24.1326 n104
1 42.8383 n25
27 55.6946 \iRECEIVER/rec_datSyncH
2 55.6947 n112
```

REFSM

For actual function recovery REFSM is typically relied upon. Its goal is given a netlist and a set of registers, construct a FSM that represents the behavior of the register set over the course of the operation of the netlist. REFSM can be very useful for checking how register values interact with each other at a higher level. Here is example output of running REFSM on the uart,

```
Reading the library...
Done Reading the Library.
Number of Gates is 168
Number of Regs is 59

The Word file used is data/words/uart_word.in.

The transitions will be written to standard out

Looking for reset...
Trying xmit_dataH[0] as reset
Trying xmit_dataH[1] as reset
Trying xmit_dataH[2] as reset
Trying xmit_dataH[3] as reset
Trying xmit_dataH[4] as reset
Trying xmit_dataH[5] as reset
Trying xmit_dataH[6] as reset
Trying xmit_dataH[7] as reset
Trying sys_clk as reset
Trying sys_rst_l as reset

Using the signal sys_rst_l for reset.

Looking for States...
Found States.

Looking for Transitions...
Transitions
000 --> 000
000 --> 001
001 --> 001
001 --> 011
011 --> 011
011 --> 100
011 --> 110
100 --> 011
110 --> 000
110 --> 110
Found Transitions.

Exiting...
```

The key part of the output being the state transitions after the line "`Transitions`".

Reset Signals

REFSM can require some finagling to work. REFSM will by default try to find the reset state of the given netlists. The reason for the extra reset signal search being that if such a signal exists, the FSM constructed from the words would have many extra edges for when the reset signal is used. This is due in part to the traversal to the reset state. These extra edges can very easily obfuscate the actual function of the FSM. When searching for a reset signal the following types of messages will be written to standard out,

```
The transitions will be written to standard out

Looking for reset...
Trying xmit_dataH[0] as reset
Trying xmit_dataH[1] as reset
Trying xmit_dataH[2] as reset
Trying xmit_dataH[3] as reset
Trying xmit_dataH[4] as reset
Trying xmit_dataH[5] as reset
Trying xmit_dataH[6] as reset
Trying xmit_dataH[7] as reset
Trying sys_clk as reset
Trying sys_rst_l as reset

Using the signal sys_rst_l for reset.

Looking for States...
```

To turn off the reset finding part of the code the argument "`--norst`" should be used when running the REFSM command. When no reset is used REFSM assumes that any input signal is fair game for use and that the reset state is the state where each signal of the provide signal set is set to low.

```
The transitions will be written to standard out

Looking for States...
```

Please note the differences in the outputted FSMs when using reset and not using resets,

No Reset
```
000 --> 000
000 --> 001
001 --> 000
001 --> 001
001 --> 011
011 --> 000
011 --> 011
011 --> 100
011 --> 110
100 --> 000
100 --> 011
110 --> 000
110 --> 110
```

With Reset
```
000 --> 000
000 --> 001
001 --> 001
001 --> 011
011 --> 011
011 --> 100
011 --> 110
100 --> 011
110 --> 000
110 --> 110
```

Decomposition

Secondly the extracted FSM might be the combination of different FSMs which could have their own independent function. To help determine when this composition of FSMs might occur a tool for separating FSMs is provided that allows for FSM decomposition. The decomposition tool does rely on a heuristic and for that matter it does not necessarily have 100% accuracy. In fact the method used is prone to separate FSMs that are dependent, so if the method finds that the FSMs are dependent it is most-likely the case unless the FSMs have a similar coloring, which case decomposition is actually provably impossible from the functional FSM alone.

TODO: <Decomposition example/ pictures>

Printing Options

There are several ways (or formats) to receive the data which is found by running REFSM.

Transition Conditions

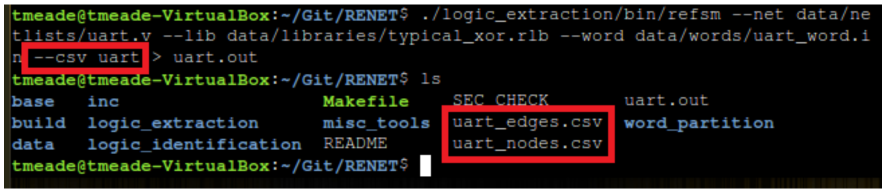
Another important thing to note about REFSM is that by default the tool does not print the conditions required for transitions. To print such conditions the command line argument "`--cond`" should be used when running the REFSM command. Each condition for transition is separated by the token "`::`"

```
Looking for Transitions...
Transitions :: Conditions
000 --> 001 :: xmitH = 1
000 --> 000 :: xmitH = 0
001 --> 011 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 :: \iXMIT/bitCell_cntrH[3] = 1 ::
\iXMIT/bitCell_cntrH[2] = 1 :: \iXMIT/bitCell_cntrH[0] = 1
001 --> 001 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 :: \iXMIT/bitCell_cntrH[3] = 1 ::
\iXMIT/bitCell_cntrH[2] = 1 :: \iXMIT/bitCell_cntrH[0] = 0
001 --> 001 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 :: \iXMIT/bitCell_cntrH[3] = 1 ::
\iXMIT/bitCell_cntrH[2] = 0
001 --> 001 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 :: \iXMIT/bitCell_cntrH[3] = 0
001 --> 001 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 0
001 --> 001 :: n98 = 0
011 --> 100 :: \iXMIT/n67 = 1 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 ::
\iXMIT/bitCell_cntrH[3] = 1 :: \iXMIT/bitCell_cntrH[2] = 1 :: n99 = 1
011 --> 110 :: \iXMIT/n67 = 1 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 ::
\iXMIT/bitCell_cntrH[3] = 1 :: \iXMIT/bitCell_cntrH[2] = 1 :: n99 = 0 :: n100 = 1 ::
n101 = 1 :: n102 = 1
011 --> 100 :: \iXMIT/n67 = 1 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 ::
\iXMIT/bitCell_cntrH[3] = 1 :: \iXMIT/bitCell_cntrH[2] = 1 :: n99 = 0 :: n100 = 1 ::
n101 = 1 :: n102 = 0
011 --> 100 :: \iXMIT/n67 = 1 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 ::
\iXMIT/bitCell_cntrH[3] = 1 :: \iXMIT/bitCell_cntrH[2] = 1 :: n99 = 0 :: n100 = 1 ::
n101 = 0
011 --> 100 :: \iXMIT/n67 = 1 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 ::
\iXMIT/bitCell_cntrH[3] = 1 :: \iXMIT/bitCell_cntrH[2] = 1 :: n99 = 0 :: n100 = 0
011 --> 011 :: \iXMIT/n67 = 1 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 ::
\iXMIT/bitCell_cntrH[3] = 1 :: \iXMIT/bitCell_cntrH[2] = 0
011 --> 011 :: \iXMIT/n67 = 1 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 ::
\iXMIT/bitCell_cntrH[3] = 0
011 --> 011 :: \iXMIT/n67 = 1 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 0
011 --> 011 :: \iXMIT/n67 = 1 :: n98 = 0
011 --> 011 :: \iXMIT/n67 = 0
100 --> 011
110 --> 000 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 :: \iXMIT/bitCell_cntrH[3] = 1 ::
\iXMIT/bitCell_cntrH[2] = 1 :: \iXMIT/bitCell_cntrH[0] = 1
110 --> 110 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 :: \iXMIT/bitCell_cntrH[3] = 1 ::
\iXMIT/bitCell_cntrH[2] = 1 :: \iXMIT/bitCell_cntrH[0] = 0
110 --> 110 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 :: \iXMIT/bitCell_cntrH[3] = 1 ::
\iXMIT/bitCell_cntrH[2] = 0
110 --> 110 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 1 :: \iXMIT/bitCell_cntrH[3] = 0
110 --> 110 :: n98 = 1 :: \iXMIT/bitCell_cntrH[1] = 0
110 --> 110 :: n98 = 0
Found Transitions.
```

Each transition condition (the text following the first "`::`" on a line) follows the corresponding transition. Each part of the transition condition is sperated by "`::`", and for the condition to hold true each part of the condition must be true (e.g. "`S_i --> S_j :: A :: B :: C`" means that the FSM at state `S_i` will transition to state `S_j` if conditions A, B, and C are true).

Comma Separated Values

To print out the FSM graph to a format accepted by graphical graph analysis tools (e.g. gephi https://gephi.org/) use the “--csv” command followed by the desired prefix of the output file names. This option will create a Comma Separated Value (CSV) file for both the nodes and the edges of the FSM.



```
tmeade@tmeade-VirtualBox:~/Git/RENET$ ./logic_extraction/bin/refsm --net data/ne
tlists/uart.v --lib data/libraries/typical_xor.rlb --word data/words/uart_word.i
n --csv uart > uart.out
tmeade@tmeade-VirtualBox:~/Git/RENET$ ls
base   inc                  Makefile    SEC_CHECK        uart.out
build  logic_extraction     misc_tools  uart_edges.csv   word_partition
data   logic_identification README      uart_nodes.csv
tmeade@tmeade-VirtualBox:~/Git/RENET$
```

Here is an example of the CSV files for a given set of output transitions.

```
Original Transitions      nodes.csv            edges.csv
       000 --> 000              Id,Label             Source,Target,Type,Id,Label,Weight
       000 --> 001              0,000                0,0,Directed,1,,1.0
       001 --> 001              1,001                0,1,Directed,2,,1.0
       001 --> 011              2,011                1,1,Directed,3,,1.0
       011 --> 011              3,100                1,2,Directed,4,,1.0
       011 --> 100              4,110                2,2,Directed,5,,1.0
       011 --> 110                                   2,3,Directed,6,,1.0
       100 --> 011                                   2,4,Directed,7,,1.0
       110 --> 000                                   3,2,Directed,8,,1.0
       110 --> 110                                   4,0,Directed,9,,1.0
                                                     4,4,Directed,10,,1.0
```

Print to File

Additionally to write the transitions (or transitions and conditions) to an output file the command “--out" can be used followed by the desired output file name. Although the transitions are written to the specified file the status of the program will be written to standard output so the user can determine what step REFSM is in.

The Word File

To get pass in the desired set of signals to form the FSM around you must use a word file. The file should have the following form

```
<Number of Signals>
<Signal Name 1>
<Signal Name 2>
.
.
.
<Signal Name <Number of Signals>>
```

When determining the signals for the word it is strongly recommended that one does not use the signal and its complement in the word file. This is because when no reset signals are found each signal is defaulted to 0. If a signal and its complement is 0 something can go logically very wrong within the netlist and unexpected FSM behavior could be observed which could cause issues with the resulting FSM.

REPATH

A method similar to REFSM is that of REPATH. What REPATH tries to do at a high level is find a series of input patterns that will force a FSM that starts at some reset state into a state that is specified by the user. The state can be singular or a set of states. The REPATH will always find the least number of steps needed to get to that state. However, since a user might not omit registers from the FSM word certain transition conditions might not be achievable at certain clock cycles used by the REPATH method.

For running the method you can use the following command

```
> ./logic_extraction/bin/repath --net <netlist_file_name> --word <word_file_name> --bs
<base_state_file_name>
```

Here is a sample output from the REPATH method

```
Reading the library...
Done Reading the Library.
Number of Gates is 168
Number of Regs is 59
The Word file used is data/words/uart_word.in.

The path will be written to standard out
Looking for reset...
Trying xmit_dataH[0] as reset
Trying xmit_dataH[1] as reset
Trying xmit_dataH[2] as reset
Trying xmit_dataH[3] as reset
Trying xmit_dataH[4] as reset
Trying xmit_dataH[5] as reset
Trying xmit_dataH[6] as reset
Trying xmit_dataH[7] as reset
Trying sys_clk as reset
Trying sys_rst_l as reset

Using the signal sys_rst_l for reset.
Looking for States...
Looking for Transitions...

000 --> 001
sys_rst_l = 1  && xmitH = 1
001 --> 011
sys_rst_l = 1  && \iXMIT/bitCell_cntrH[0] = 1  && \iXMIT/bitCell_cntrH[2] = 1  &&
\iXMIT/bitCell_cntrH[3] = 1  && \iXMIT/bitCell_cntrH[1] = 1  && n98 = 1
Exiting...
```

The first thing you might note is that the output of REPATH is fairly similar to that of REFSM. The reason this is the case is because REPATH uses REFSM to find the FSM and then uses that data to find the paths/conditions for transitioning.

Print to File

Like REFSM, REPATH is also capable of writing the transitions and conditions to an output file via the command “--out". Also like REFSM the status of the REPATH will be written to standard output so the user can determine what step REPATH is in.

The Input Files

The base state file should have the following form

```
<Number of States>
<State 1>
<State 2>
.
.
.
<State <Number of States>>
```

Where each State is one of the states your FSM can enter to finish.

The word file has the same form as the word file used in other tools (e.g. REFSM).

REBUS

A problem frequently discussed in research for Hardware Trojan detection is that of data flow analysis. The theory is that many Trojan methods would leak sensitive information, and by determining where data was going an IC user could spot places where information was improperly. One of the most popular methods for doing such an analysis was that of WordRev (by W. Li et al). WordRev utilized a series of wire comparison techniques to evaluate the likelihood of same word membership and a method called forward propagation to find potential word pairs. With these two techniques combined together, WordRev had very good runtime and results, compared to previous methods. However, WordRev's method did have some inaccuracies mentioned in the paper; with certain optimization parameters WordRev failed to find the signal for each word. This error meant that WordRev was able to find the general topology, but certain signals could potentially able to interact with other parts of the netlist and the IP users would be unaware. For this matter we developed our own tool that also focuses on finding words by leveraging the data bus in the netlist, but rather than relying on a series of comparison tests, we leverage our own comparison method that had been mentioned earlier in this document. For REBUS we use RELIC's signal comparison to verify the relationship between signals.

Forward Propagation

As mentioned earlier WordRev and for that matter REBUS both use forward propagation to find potential signal pairs. Ideally when identifying a pair of signals as having a membership in the same word other signal potential pairs should be found for comparison. Forward propagation finds potential signal pairs by looking in the merged word's fan-out. The merged words have their pair of signals added to a list and checked at a later part of the REBUS method.

Input Word

To find the potential using forward propagation certain initial information needs to be passed to REBUS. This can be supplied with the input word pins by using the --word flag the word flag file uses the following format

```
<number_of_input_words>
<number_of_signals_1>
<word_1_signal_1>
<word_1_signal_2>
.
.
.
<word_1_signal_<number_of_signals_1>>
<number_of_signals_2>
<word_2_signal_1>
<word_2_signal_2>
.
.
.
<word_2_signal_<number_of_signals_2>>
.
.
.
<number_of_signals_<number_of_input_words>>
<word_<number_of_input_words>_signal_1>
<word_<number_of_input_words>_signal_2>
.
.
.
<word_<number_of_input_words>_signal_<number_of_signals_<number_of_input_words>>>
```

Custom Libraries

As usual users can specify a custom cell library in the RLB file format. The command line argument for using a custom library is --lib, which should be followed by the library file name.

RELIC Artifacts

Since REBUS uses RELIC for matching verifying signal pairs, several parameters for tweaking the RELIC function exists. For REBUS access to threshold, depth, and buffer behavior.

Threshold

Like with RELIC a threshold for signals to be considers similar enough exists. To adjust this parameter simply use the flag --thesh followed by the desired threshold amount between 0 and 1.

Depth

Since RELIC is recursive with no guarantee of end on certain netlists a depth parameter is used to break out of the recursion and allow signals to heuristically be consider a perfect match. To specify the depth for REBUS simply use the command line argument --depth followed by the desired depth.

Buffer Behavior

Since depending on the intention or the synthesis of the chip a buffer tree might make matching the logic structure difficult, we enable users to disable and merge buffers trees into singular wires by using the --buf flag.

Outputting

The output of REBUS consists of the registers of the original netlists. Each register of a group is on its own line and the groups of words are separated by blank lines. Here is an example output for the rsa,

```
> ./word_partition/bin/rebus --net data/netlists/rsa.v --lib
data/libraries/typical_xor.rlb --word data/words/rsa-input.in

ready

tempin[0]

internal wire 10

sqrin[1]

internal wire 14

sqrin[2]


internal wire 17

sqrin[3]

internal wire 20

sqrin[22]
sqrin[20]
```

```
sqrin[24]
sqrin[12]
sqrin[10]
sqrin[6]
sqrin[4]

internal wire 80
internal wire 74
internal wire 87
internal wire 49
internal wire 42
internal wire 30
internal wire 24

sqrin[21]
sqrin[19]
sqrin[18]
sqrin[17]
sqrin[31]
sqrin[30]
sqrin[29]
sqrin[28]
sqrin[23]
sqrin[15]
sqrin[13]
sqrin[11]
sqrin[9]
sqrin[8]
sqrin[7]
sqrin[5]

<Output word sets omitted for brevity>

\modmultiply/mpreg[0]

\modsqr/mpreg[31]
\modsqr/mcreg[0]

\modsqr/mpreg[25]
\modsqr/mpreg[26]
\modsqr/mpreg[24]
\modsqr/mpreg[0]
```

To File

The --out flag allows users to print the output to a file rather than to standard output. The file name for output should follow the --out flag in the argument list.

REPCA

Forward propagation based methods tend to suffer in situations when early data bus words are not completely found. Missing a word at the beginning and perpetuate errors throughout the entire netlist. Since forward propagation is prone to such catastrophic error, we worked on developing a tool that could try to recover from certain mismatches. RELIC by itself is too slow as it takes an amount of time proportional to the square of the number of signals. PCA based methods with smart distance finding could have a good potential for finding quickly, and with a high accuracy, the words that make up the netlist (not just those along the data bus).

Overview

To use REPCA you must use the “relic_pca” tool in the word_partition tool kit.

```
> ./word_partition/bin/relic_pca --net data/netlists/aes.v --lib
data/libraries/typical_xor.rlb --feat data/feats/feat_1.txt --out tmp.out --persplt .01

k0[63]
k0[62]
k0[61]
k0[60]
k0[59]
k0[58]
k0[57]
k0[56]
k0[49]
k0[47]
k0[46]
k0[45]
k0[44]
k0[43]
k0[42]
k0[41]
k0[40]
k0[39]
k0[38]
k0[37]
k0[36]
k0[35]
k0[34]
k0[33]
k0[32]

k0[55]
k0[54]
k0[53]
k0[52]
k0[51]
k0[50]
k0[48]
k0[95]
k0[94]
k0[93]
k0[92]
k0[91]
k0[90]
k0[89]
k0[81]
k0[79]
```

```
k0[78]
k0[77]
k0[76]
k0[75]
k0[74]
k0[73]
k0[72]
k0[71]
k0[70]
k0[69]
k0[68]
k0[67]
k0[66]
k0[65]
k0[64]

k0[88]

<Output omitted for brevity>

\r9/p31[19]
\r9/p20[27]
\r9/p32[11]
\r9/p33[3]
\r8/p32[11]
\r8/p33[3]

\rf/p00[7]
\rf/p22[7]
\rf/p21[7]
\rf/p23[7]

\rf/p22[2]
\rf/p10[2]
\rf/p21[2]
\rf/p23[2]

\rf/p20[7]

\r8/p01[17]
\r8/p02[9]
\r6/p02[9]
\r6/p31[18]
\r6/p32[10]

s6[26]
s5[26]
```

RELIC 2 Artifacts: PCA Parameters

Since RELIC 2 uses PCA to find the potential logic registers of the netlist all of the PCA based parameters in RELIC 2 are also in REPCA. In fact each parameter used by RELIC 2 are used by REPCA, in part because RELIC is a usable variable for REPCA.

Extra PCA Parameters

Since REPCA focuses on finding words with PCA rather than individual signals that are likely to be logic, additional parameters might be needed to accurately find the words of the netlist.

Clustering

The method for finding groups in a set of n-dimensional points is commonly referred to as clustering. For the problem at hand we are trying to cluster unlabeled data. Several methods exist. The more common methods include K-Means, EM, Density based clustering, and Nearest Neighbor. For our tools we used a modified Nearest Neighbor to find create groups. REPCA either takes in a distance or via a randomize algorithm estimates an appropriate distance for creating clusters. Using a greedy method and the distance the clusters are created.

To specify the distance manually, one can pass the parameters "--rad <cluster_radius>" into the REPCA command. The second method for creating a custom radius is to use the command "--persplt <ratio_of_signals>". The ratio should be in the range [0.0, 1.0]. The radius will be selected such that the number of signals in each cluster will be around <ratio_of_signals> times the <number_of_registers>.

Other Tools

Aside from doing all the analysis based on netlist and their structure sometimes simple analysis outside the scope of Integrated Circuits can assist in understanding the function of a netlist. To help with this a few simple tools have been developed for IC analysis minus the IC.

Tarjan’s Strongly Connected Component Algorithm

One of the most popular tools used by our developers (and others) for finding the high level function behavior embedded in the gates of a circuit is that of Strongly Connected Component (SCC) finding. For those of you that are unaware of what an SCC is, an SCC is a subgraph of a directed graph such that each node within the SCC is capable of reaching each other node. What this can mean for the sake of IC analysis an FSM that leaves its state graph SCC will have effectively cutoff other FSM states. Transitioning out of an SCC implies that the control logic of the netlist has entered into a particular mode that effectively locks the behavior of the IC until the unit is powered off (or something else to that extent). This analysis can be used for finding “black hole” states within an obfuscated FSM or finding functioning states in a sequential locking FSM. The analysis also allows for a more detailed categorization of register and their behaviors and interactions, which can allow for finding FSM words (as seen in Shi et. al’s “A Highly Efficient Method for Extracting FSMs from Flattened Gate-Level Netlist”).

How we supply the methods for performing such an analysis is by using Tarjan’s algorithm. His method is commonly used since it’s run time is O(V + E), which is about as good as you can get in terms of meaningful graph algorithms. The tool itself is contained in the “`/misc_tools/`” directory of the toolset. To run it simply use the command

```
> ./misc_tools/bin/tjscc --tran <transition_file_name>
```

The transition file needs to be of the form

```
<edge_1_begin> --> <edge_1_end>
<edge_2_begin> --> <edge_2_end>
.
.
.
<edge_E_begin> --> <edge_E_end>
```

Here's an example of our Tarjan implementation

uart.in file

```
000 --> 111
111 --> 101
101 --> 111
001 --> 010
111 --> 010
010 --> 010
000 --> 000
000 --> 001
001 --> 001
001 --> 011
011 --> 011
011 --> 100
011 --> 110
100 --> 011
110 --> 000
110 --> 110
```

Output

```
Transition file used is uart.in
0 010

1 101
1 111

2 110
2 100
2 011
2 001
2 000

SCC Transitions
0 (1) --> 0 (1)
1 (2) --> 0 (1)
1 (2) --> 1 (2)
2 (5) --> 0 (1)
2 (5) --> 1 (2)
2 (5) --> 2 (5)

Sink SCCs are the following,
0 (1)

Source SCCs are the following,
2 (5)
```

Output Options

To output the results to a file rather than standard out the option "--out" followed by the desired output file name can be passed in the arguments.

REDPEN

For doing simple analysis on the register interaction the REDPEN tool can be utilized. The purpose of the REDPEN is to find the register dependency graph. The tool outputs the dependency as a transition graph which looks like the following

```
<register_affecter_1> --> <dependant_register_1>
<register_affecter_2> --> <dependant_register_2>
.
.
.
<register_affecter_n> --> <dependant_register_n>
```

The runtime of this is O(|R||N|), where R is the set of registers and N is the set of all gates. However, due to how fan-out/fan-in trees are formed in practice the runtime tends to be closer to O(|N|).

```
> ./misc_tools/bin/redpen --net <netlist_file_name>
```

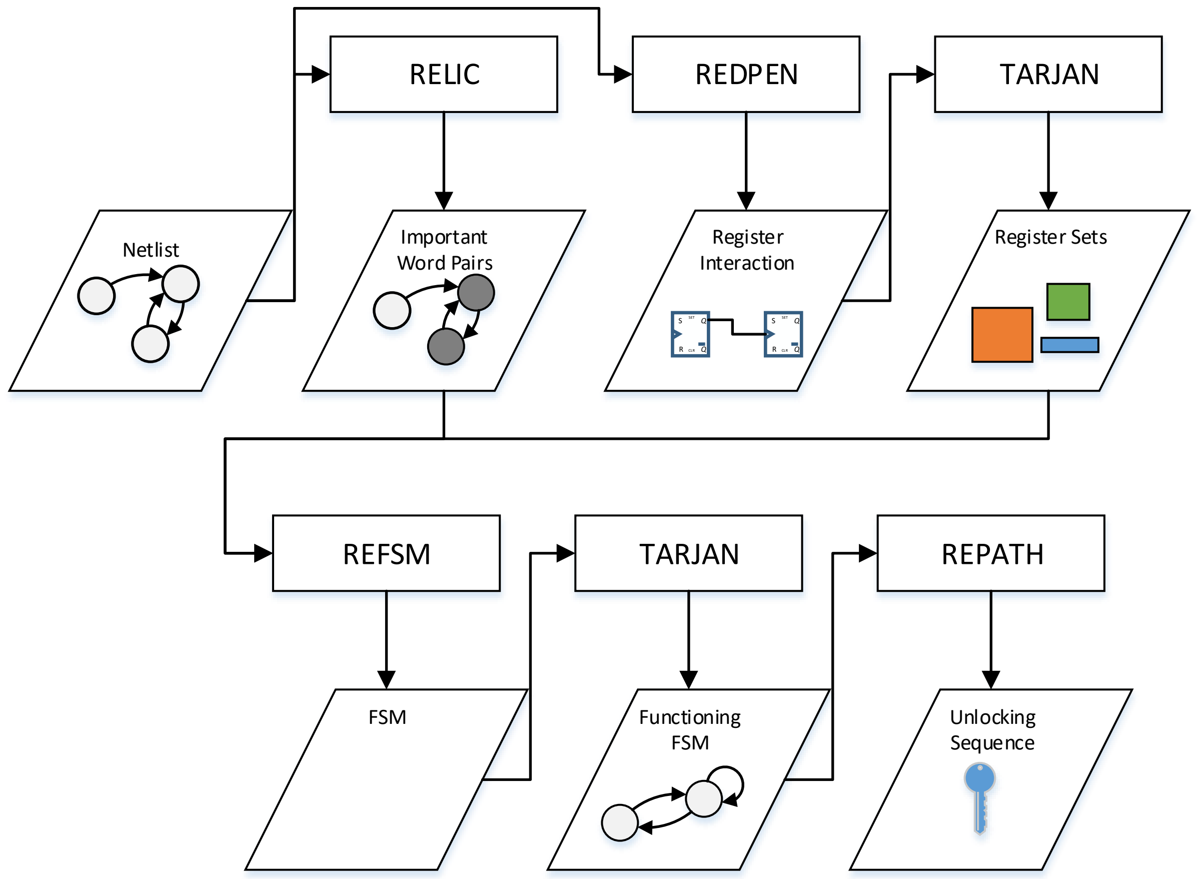
Like many of the other tools REDPEN can use a RLB as a library.

Output Options

To output the results to a file rather than standard out the option “--out" followed by the desired output file name can be passed in the arguments.

Case Study: Working HARPOON

This case study will work with a second netlist that was locked using a HARPOON based locking method.



Use RELLIC 2

Using the RELIC 2 tool we can appropriately find registers that are most likely the logic registers.

```
> ./logic_identification/bin/relic --net data/netlists/aes_harp_2.v --lib
data/libraries/harpoon.rlb --shft 0 --out tmp.out -feat feats_2.txt

<14-pages of output omitted for brevity>


415 10.0009 text_out34
376 10.5984 n16445
336 10.6924 text_out19
359 10.9018 n16387
416 11.862 n16432
420 11.862 n16386
868 12.3554 n32181
337 12.4554 n16452
869 13.6004 n16357
335 14.5433 text_out20
885 21.2615 u0_r0_rcnt00
886 23.3122 n16162
2 31.9352 n32191
0 105.452 done
```

```
1 114.144 ff_qn0
899 193.091 obf_n12
```

From the output we can guess that obf_n12 is going to be important for determining behavior. We will keep the word small initially and only use obf_n12 and it's SCC

Use REDPEN + Tarjan's (optional)

Here is the output of the register interaction extracted using redpen and writing to a file.

```
> ./misc_tools/bin/redpen --net data/netlists/aes_harp_2.v --lib
data/libraries/harpoon.rlb --out tmp.out

Reading the library...
Done Reading the Library.
Number of Gates is 16766
Number of Regs is 546
Writing results to tmp.out
Exiting...
```

After running Tarjan's here is the SCC of register interaction.

```
> ./misc_tools/bin/tjscc --tran tmp.out

Transition file used is tmp.out
0 n16452
0 text_out2
0 n16457
0 n16442
0 n16384
0 n16428
0 text_out3
0 n16386
0 text_out118
0 n16207
0 n16344
0 n16211
0 n16333
0 n16221

<10 pages omitted>

105 n16247

106 n16296

107 n16237

108 n16309

109 test3
109 test2
109 obf_n12
109 obf_mux_sel1
109 test1
109 n16133
109 obf_n126
109 obf_n9
109 obf_n8
109 n16189
```

```
SCC Transitions
0 (258) --> 0 (258)
1 (5) --> 0 (258)
1 (5) --> 1 (5)
2 (1) --> 0 (258)
2 (1) --> 1 (5)

<9 pages omitted>

108 (1) --> 108 (1)
109 (10) --> 101 (1)
109 (10) --> 102 (2)
109 (10) --> 103 (1)
109 (10) --> 104 (1)
109 (10) --> 105 (1)
109 (10) --> 106 (1)
109 (10) --> 107 (1)
109 (10) --> 108 (1)
109 (10) --> 109 (10)

Sink SCCs are the following,
0 (258)

Source SCCs are the following,
109 (10)
```

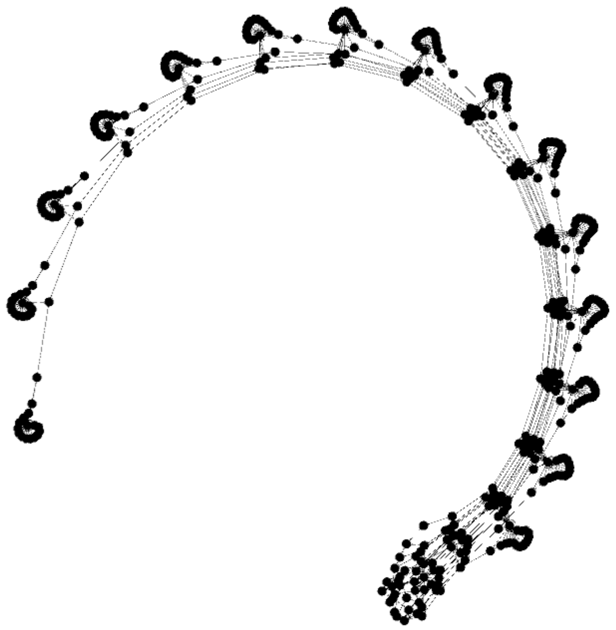
We will use the SCC with id 109 since it contains the wire we are interested in.

Use REFSM

After finding the registers for simulation we can run REFSM to extract the FSM of the logic graph. Here is the resulting output of REFSM on SCC 109. Note we have to adjust the signals to contain the Q pin, since if no reset signal is found we assume that the reset state is all zeros. It is commonly accepted that the Q pin is initially low on power on.

```
./logic_extraction/bin/refsm --net data/netlists/aes_harp_2.v --lib
data/libraries/harpoon.rlb --word data/words/aes_harp_2_word.in --out tmp.out

Reading the library...
Done Reading the Library.
Number of Gates is 16766
Number of Regs is 546

The Word file used is data/words/aes_harp_2_word.in.

The transitions will be written to tmp.out

Looking for reset...
Trying clk as reset
Trying ld as reset
Trying key127 as reset
Trying key126 as reset

<Potential reset signals omitted>

Trying text_in126 as reset
Trying text_in125 as reset
Trying text_in127 as reset
Trying rst as reset

WARNING: Reset signal not found.

Looking for States...
Found States.

Looking for Transitions...
Found Transitions.
Transitions
Transitions
0000000000 --> 0000000000
0000000000 --> 0000000001
0000000001 --> 0000000000

<Transitions omitted for brevity>

1111111001 --> 1111110101
1111111000 --> 1111100101
1111111000 --> 1111111101

Exitting...
```

Unfortunately it appears that there is no signal that resets the FSM, but we will try to work with what we have and try to determine the high level function. Below you will find the picture of the resulting FSM.



Use Tarjan's

At this point it is useful to cover two broad type of FSM encryption that can be employed. The first is that of a staticly locked FSM. For such FSM the key that is used to unlock the FSM is unchanging. Some researchers have found methods for breaking this scheme by reducing it to a SAT problem (assuming they have a certain form of the high level function).

Since the key in a staticly locked FSM is unchanging there is a "low" number of possible keys. This brings the topic to the second type of locked FSM, These second type FSMs are sequentially locked, and ideally after an initial input sequence the FSM will enter a functioning FSM state. It normally is the case that each functioning FSM state can reach each other functioning FSM state and no functioning FSM state can reach a non-functioning FSM state. This directly implies that the functioning FSM in a FSM should be a "sink" connected component. To find such components in a FSM Tarjan's can be leveraged. Here is a picture of the found FSM

```
./misc_tools/bin/tjscc --tran tmp.out

Transition file used is tmp.out
0 1111100110
0 1111101101
0 1111100101
0 1111100100
0 1111100010
0 1111100000

1 1111101000

2 1111100111

<SCCs omitted>

163 0000110001

164 0000010000

165 0000001111
165 0000001110
165 0000000111
165 0000000110
165 0000001011
165 0000001010
165 0000000011
165 0000000010
165 0000001101
165 0000001100
165 0000000101
165 0000000100
165 0000001001
165 0000001000
165 0000000001
165 0000000000

SCC Transitions
0 (6) --> 0 (6)
1 (1) --> 0 (6)
2 (1) --> 0 (6)

<Transitions omitted>
```

```
164 (1) --> 163 (1)
165 (16) --> 164 (1)
165 (16) --> 165 (16)

Sink SCCs are the following,
0 (6)

Source SCCs are the following,
165 (16)
```

Base on the resulting output we can see that all states are within the same SCC. This is problematic, because this means that the locking FSM can be accessed by the functioning FSM.

Going back and performing REFSM on the word which also contains the signals from the SCCs that affect the initial SCC. We get a new transition graph that contains one large SCC that is composed of 524 states. This leads us to believe that something about the modified netlist is incorrect. Either the states chosen for locking were not checked for existing within the original netlist, or the reset signal was broken and does not deterministically transition to the correct reset state.

Use REPATH

Finally REPATH can be used to find the unlocking sequence for extracting the FSM. Since we are interested in the sink SCC as it would have the highest chance of containing our function FSM, we will find an input sequence to get the FSM into one of those states. It should be noted that if multiple sink SCCs exist it could mean that black hole states were incorporated to trap the FSM. Here is the result of running REPATH.

```
> ./logic_extraction/bin/repath --net data/netlists/aes_harp_2.v --lib
data/libraries/harpoon.rlb --word data/words/aes_harp_2_word.in --bs
data/states/aes_harp_2_states.txt

Reset signal not found.
Looking for States...

Looking for Transitions...

0000000000 --> 0000000001
ld = 0  && key126 = 0  && key125 = 0  && key127 = 1
0000000001 --> 0000001000
ld = 0  && key125 = 1  && key126 = 1  && key127 = 1
0000001000 --> 0000001001
ld = 0  && key126 = 0  && key125 = 0  && key127 = 1
0000001001 --> 0000000100
key127 = 0  && ld = 1  && key125 = 0  && key126 = 1
0000000100 --> 0000000101
key127 = 1  && ld = 1  && key126 = 0  && key125 = 1
0000000101 --> 0000001100
key127 = 1  && key126 = 1  && ld = 0  && key125 = 0
0000001100 --> 0000001101
key127 = 0  && key126 = 1  && ld = 0  && key125 = 0
0000001101 --> 0000000010
key127 = 1  && key126 = 1  && ld = 0  && key125 = 0
0000000010 --> 0000000011
ld = 0  && key125 = 1  && key126 = 1  && key127 = 1
0000000011 --> 0000001010
key127 = 0  && ld = 1  && key126 = 0  && key125 = 1
0000001010 --> 0000001011
ld = 0  && key127 = 0  && key126 = 0  && key125 = 1
```

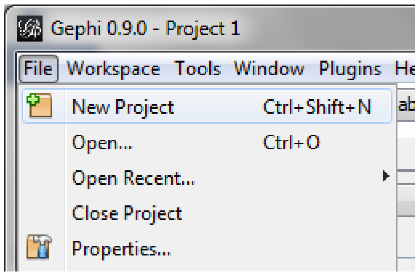
```
0000001011 --> 0000000110
key127 = 0  && key126 = 1  && ld = 0  && key125 = 0
0000000110 --> 0000000111
ld = 0  && key127 = 0  && key126 = 0  && key125 = 1
0000000111 --> 0000001110
ld = 0  && key125 = 0  && key127 = 0  && key126 = 0
0000001110 --> 0000001111
key127 = 0  && key126 = 1  && ld = 0  && key125 = 0
0000001111 --> 0000010000
ld = 0  && key127 = 0  && key126 = 0  && key125 = 1
0000010000 --> 0000110001
key127 = 0  && ld = 1  && key126 = 0  && key125 = 1
0000110001 --> 0100011000
key127 = 0  && key126 = 1  && ld = 0  && key125 = 0
0100011000 --> 0100111001
ld = 0  && key125 = 0  && key127 = 0  && key126 = 0
0100111001 --> 1000010100
ld = 1  && key125 = 0  && key126 = 1  && key127 = 1
1000010100 --> 1000110101
ld = 0  && key125 = 1  && key126 = 1  && key127 = 1
1000110101 --> 1100011100
ld = 0  && key127 = 0  && key126 = 0  && key125 = 1
1100011100 --> 1100111101
key127 = 0  && ld = 1  && key126 = 0  && key125 = 1
1100111101 --> 0010010010
ld = 1  && key125 = 0  && key127 = 0  && key126 = 0
0010010010 --> 0010110011
ld = 1  && key125 = 0  && key127 = 0  && key126 = 0
0010110011 --> 0110011010
key127 = 0  && key126 = 1  && ld = 0  && key125 = 0
0110011010 --> 0110111011
ld = 0  && key127 = 0  && key126 = 0  && key125 = 1
0110111011 --> 1010010110
ld = 0  && key126 = 0  && key125 = 0  && key127 = 1
1010010110 --> 1010110111
key127 = 0  && key125 = 1  && key126 = 1  && ld = 1
1010110111 --> 1110011110
ld = 0  && key127 = 0  && key126 = 0  && key125 = 1
1110011110 --> 1110100000
key125 = 1  && key126 = 1
1110100000 --> 1111100000
key125 = 1  && key127 = 1
Exiting...
```

It can be observed that the sequential “key” requires 124 correct bits.

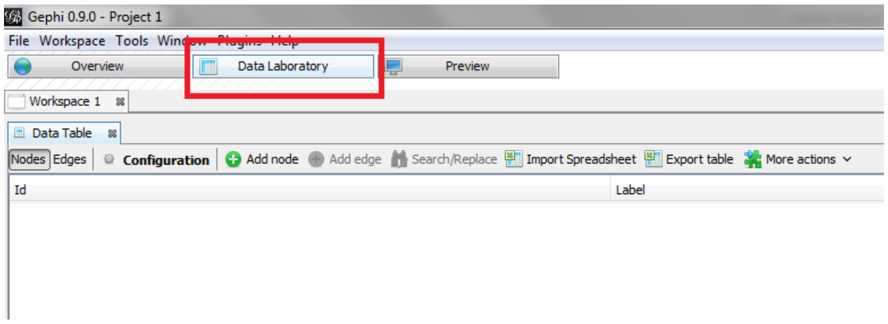
Appendix

Due to the size of certain generated data files, understanding of the information can be quite taxing. To help with this we suggest using a graph visualization tool (Gephi). It is especially helpful when analyzing the resulting FSMs. To help with expediting the process of generating the graphs the comma separated value files generated by the --csv in REFSM can be imported into the Gephi software very easily. Simply follow these steps:

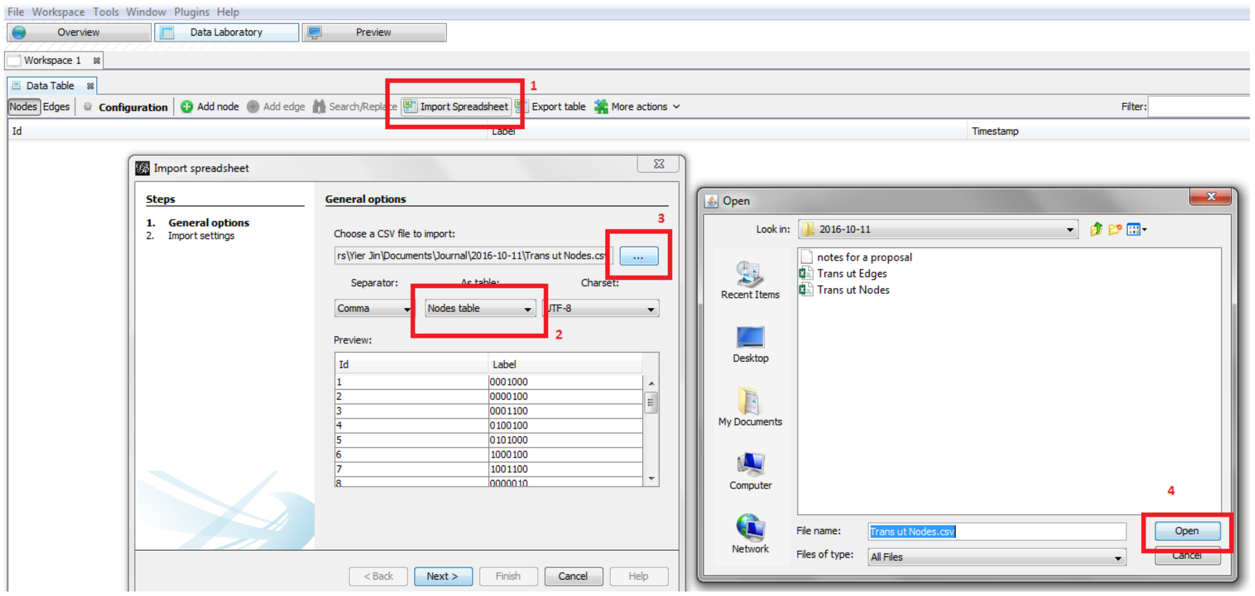
1. Create a new project



2. Go to the data laboratory tab



3. Import the nodes file [xxx_nodes.csv]



Choose a CSV file to import: rs\Yier Jin\Documents\Journal\2016-10-11\Trans ut Nodes.csv

Separator: Comma

As table: Nodes table

Charset: UTF-8

Preview:

Id	Label
1	0001000
2	0000100
3	0001100
4	0100100
5	0101000
6	1000100
7	1001100
8	0000010

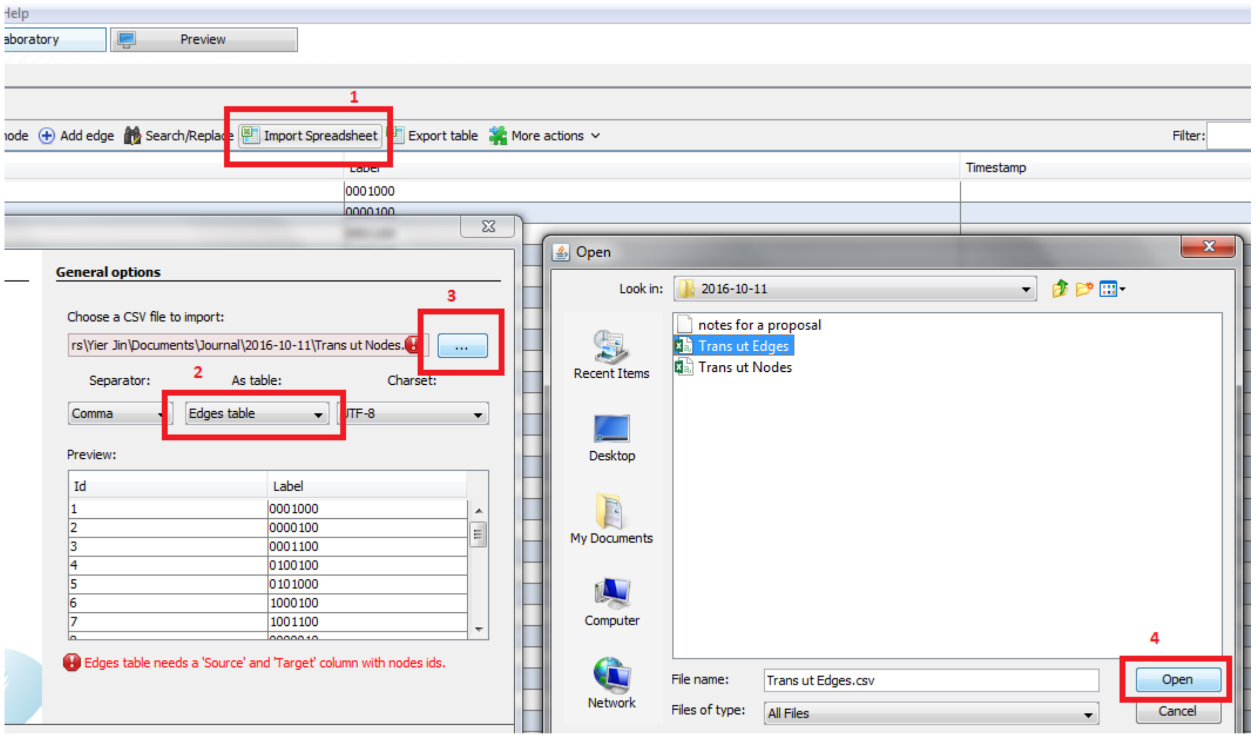
Look in: 2016-10-11

notes for a proposal
Trans ut Edges
Trans ut Nodes

File name: Trans ut Nodes.csv

Files of type: All Files

4. Import the edges file [xxx_edges.cvs]



General options

Choose a CSV file to import: rs\Yier Jin\Documents\Journal\2016-10-11\Trans ut Nodes.

Separator: Comma

As table: Edges table

Charset: UTF-8

Preview:

Id	Label
1	0001000
2	0000100
3	0001100
4	0100100
5	0101000
6	1000100
7	1001100

Edges table needs a 'Source' and 'Target' column with nodes ids.

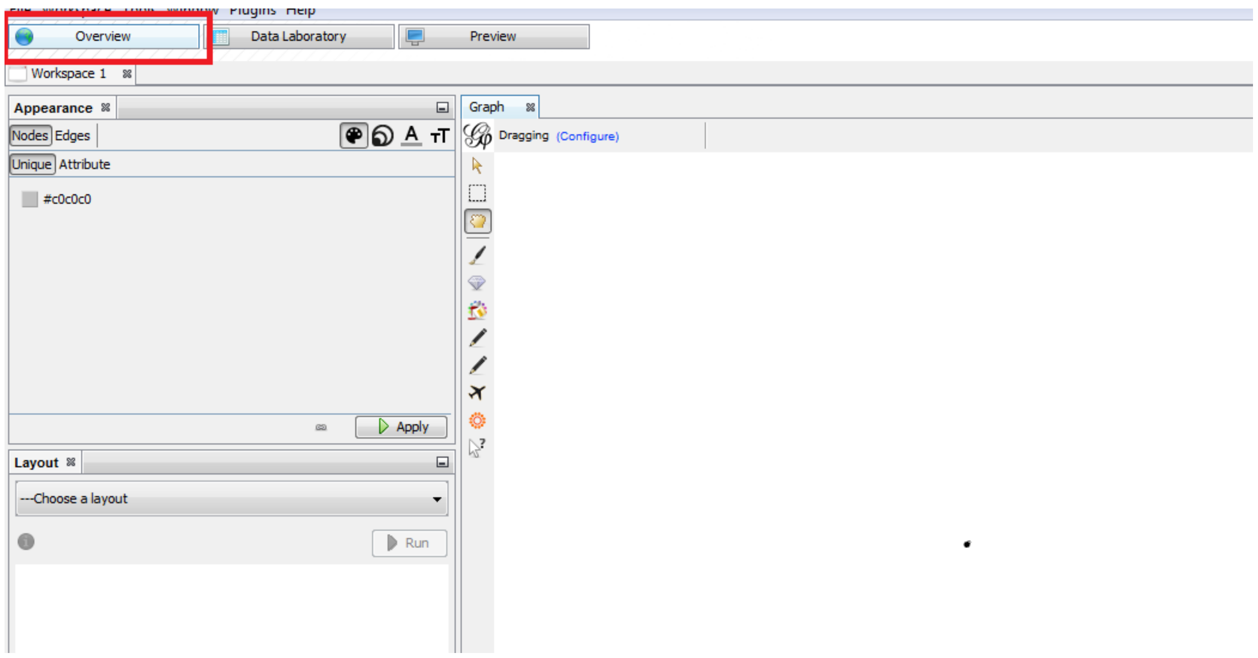
Look in: 2016-10-11

notes for a proposal
Trans ut Edges
Trans ut Nodes

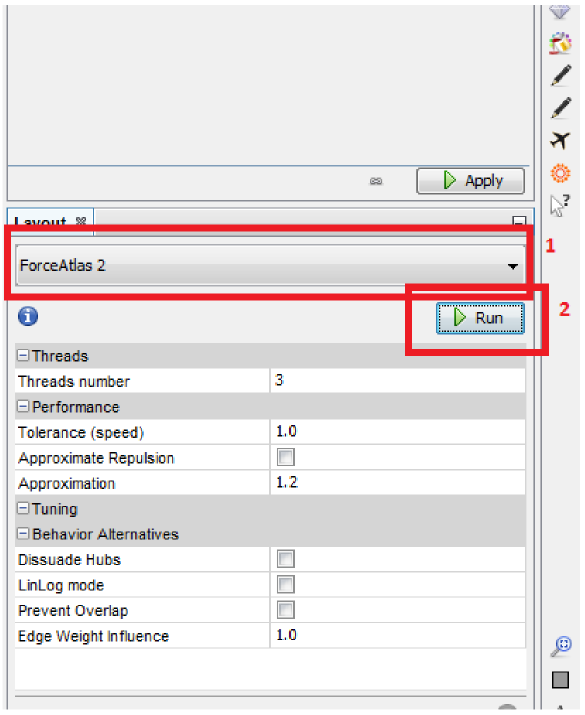
File name: Trans ut Edges.csv

Files of type: All Files

5. Go back to the overview tab



6. Use a layout option to adjust the graph into something reasonable (suggested atlas / atlas 2)



ForceAtlas 2

⊟ Threads	
Threads number	3
⊟ Performance	
Tolerance (speed)	1.0
Approximate Repulsion	☐
Approximation	1.2
⊟ Tuning	
⊟ Behavior Alternatives	
Dissuade Hubs	☐
LinLog mode	☐
Prevent Overlap	☐
Edge Weight Influence	1.0