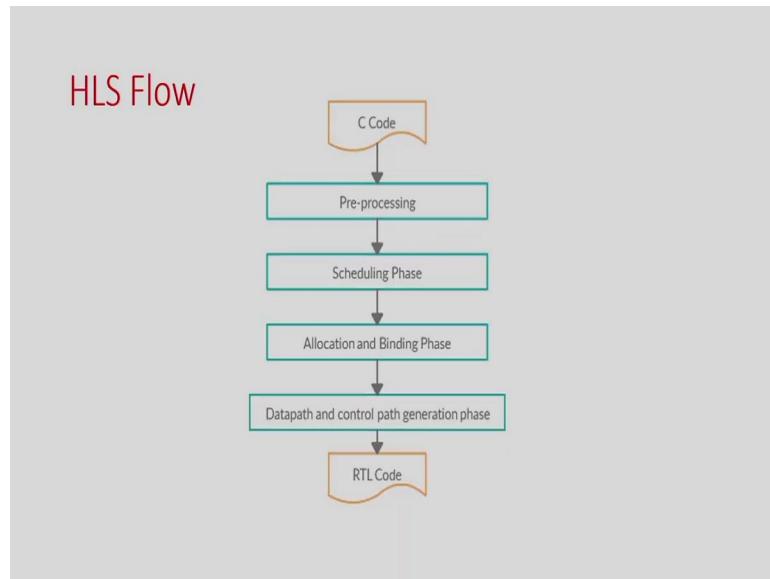


C-Based VLSI Design
Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 10
Verification of High-level Synthesis
Lecture - 35
C to RTL Equivalence Checking for High-Level Synthesis

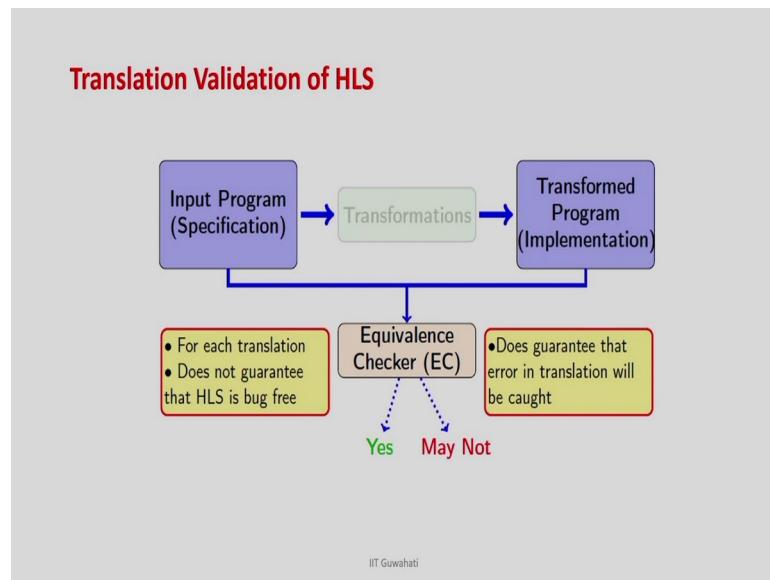
Welcome everyone. In today's class, we are going to discuss about C to RTL equivalence checking for high-level synthesis.

(Refer Slide Time: 01:01)



So, as we already know, this high-level synthesis converts C code into equivalent RTL code and it goes through various sub-processes like pre-processing, scheduling, allocation binding, data path and control, and path generation phase right.

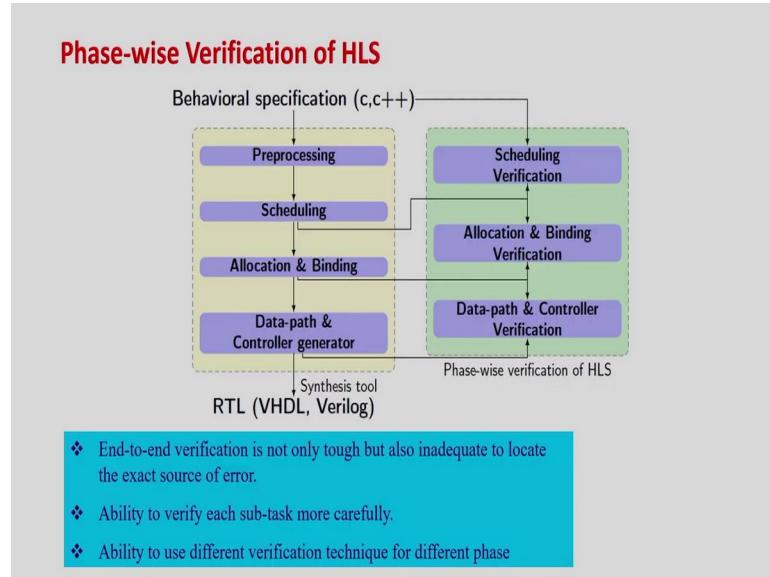
(Refer Slide Time: 01:15)



And there are two ways we can verify this process; one is simulation-based verification and another is a formal method-based verification. So, for formal method-based verification, we already discussed in last class that this translation validation is something is very popularly used for high-level synthesis.

And in translation validation, we basically take one input program we run through high-level synthesis we get an RTL and then we check the equivalence between them right. And inherently, this equivalence problem is undecidable. So, we cannot have a complete method. So, which can say yes, no in all cases; yes for equivalence no for non-equivalence, but the method usually sounds. So, it means whenever it says yes; that means, it is equivalent and it may not able to detect whether it is equivalent or not right. So, it will say may not ok. This I have we have already discussed.

(Refer Slide Time: 02:10)

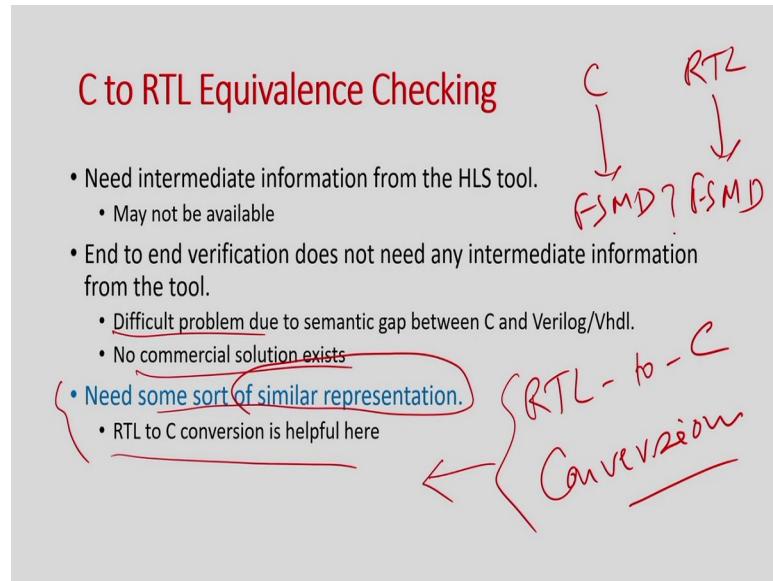


And also, we have argued in the last class that, because this C and RTL there is a lot of semantic gaps they are completely different execution patterns. So, having a direct C to RTL equivalence checking is a very difficult task right and as a result, most of the common research targets phase-wise verification of high-level synthesis right.

So, in the sense that you verify the scheduling phase, you verify the allocation and binding phase you verify the data path and controller generation phase right. So, that we have already discussed. And in the last class, we have seen the basic equivalence checking method which basically models the input-output of one phase as FSMD finite state machine data path.

And then, we define equivalence of FSMDs and then we have seen how we can actually adapt that equivalence method to verify the phases right. So, that we have already discussed.

(Refer Slide Time: 03:09)



Now, the point here is that once you try to do a phase-wise verification which actually goes to it basically verifies all the difficulties of every phases. And you can actually point out the bug that actually occurred in which phase right. The bug might occur in the scheduling phase which might occur in the allocation phase or so on right.

So, it can pinpoint that, but the disadvantage of the whole process is that you need the intermediate information right. So, you need the scheduling output right. So, whatever the scheduling output you need to get that from the tool. You need the all information after allocation and binding right.

So, which may not be always available. Specifically for commercial tools, this kind of information in detail may not be available. So, in such a case verifying this phase-wise is difficult right. On the other hand, C and RTL are the input and output of the program and is always available.

So, having an end-to-end equivalence checking for high-level synthesis where you have only the C-code and you have the RTL. There is no intermediate information available with you is something will be very important and it will be useful ok. So, that is something we are going to discuss. And we have already discussed that this particular is a very difficult problem which is checking equivalence between C and RTL; it is a very difficult problem, because of their semantic pattern right.

So, basically, the way C executes the way this RTL executes is completely different having this direct equivalence is a difficult process. But why? So, in this particular class, we are going to discuss if we want to do that what will be a possible way right and what are the limitations of that method and how we can do that how can improve the efficiency of the whole process right. So, let us discuss that.

So, the problem statement for today's class is that you have given a C code you have given a RTL which is generated by the high-level synthesis tool any high-level synthesis tool from that input C code. And I want to check whether this C and RTL equivalent or not right. So, that is the problem statement we want to do.

And; obviously, as I try to argue multiple times the way this C executes the way RTL executes is completely different. So, having an equivalence checking method basically the way I just defined like if FSMD based equivalence checking, cannot be directly applied right, because you cannot have a model of the model of V that Verilog versus the model of C you cannot compare right. Their execution pattern is different.

So, what is the way out. So, fortunately, in the last-to-last class we have discussed a converter right RTL to C converter right. So, that we have already discussed. I hope you remember that. So, specifically, the RTL that is generated by the high-level synthesis tool has a very specific structure that I have talked about that it has a data path and a controller.

And just by analyzing the data path for each controller signal is in state controllers, and assignments, we can extract the operation that is happening in the data path. And this way the controller FSM converts it into FSMD and our program model is a FSMD right. So, we have already modeled our input C behavior as FSMD only right.

And now, I am talking about how I can convert that FSM controller FSM into n FSMD. So, now, these two ICS can be comparable right. So, that something actually this RTL to C conversion actually gives an opportunity to do an end to end equivalence checking which is something that does not exist so far.

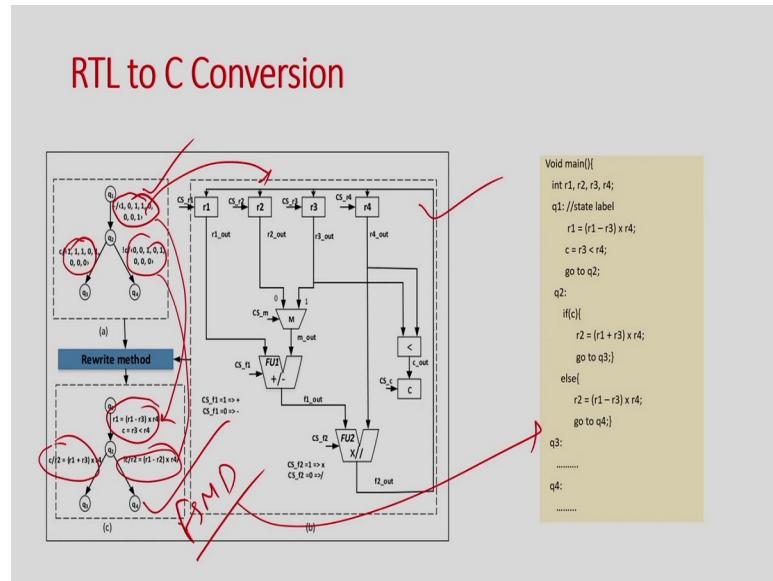
If you look into the literature at the current moment, there is no such open tools available there is no commercial solution exists. So, there is no such offering right from industry or academia. So, the point that I want to argue here is that this C to RTL conversion

actually keeps open the door right. So, you have C code you have an FSMD which we represent by C code, but say it is basically an FSMD.

So, now we have two FSMD whose 6-execution pattern is exactly the same the semantics are exactly the same. So, I can now compare right. So, the argument that I try to make to do a C to RTL equivalence checking you need some sort of similar representation right. So, a similar representation that I emphasize otherwise, you cannot compare apple with mango right. So, they are two different things.

So, similarly, we need to differ the same representation, and fortunately, because of this RTL-to-C conversion I have a similar representation now. So, this RTL to C conversion is helpful here. And that converts this C that RTL to FSMD right and I have the input C which we can represent as an FSMD; already we have discussed in the last class. So, now, I can check the right equivalence. So, that is the way I am going to do ok.

(Refer Slide Time: 08:27)



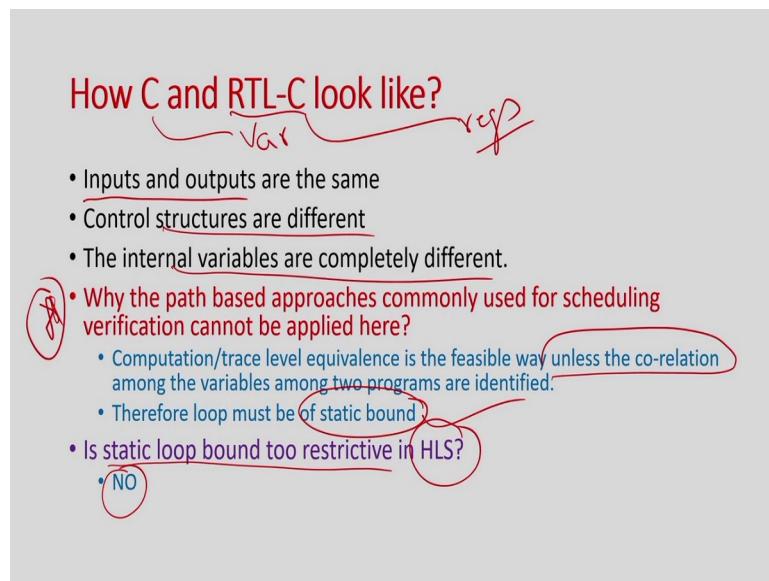
So, this RTL-C conversion I have already discussed in the last class that you have this controller FSM. Just to recap this you have the data path, what I am going to do? In each state whatever the control assignment, I will analyze the circuit for this, and I will identify, that because of these control signal values these operations are happening right.

So, I will replace this with this. I am going to do this for all states. I am going to do this to all states and once I do this is nothing, but my FSMD. And if this particular behavior

has no clock, no reset, no memory nothing this is simple high-level C code right. So, that is what we have already discussed. So, this we can represent as a C code as well.

So, this actually helps us ok. So, the approach that we are going to take is that we convert this RTL into an FSMD by this RTL this conversion method that we have already discussed. And now, we have two behavior which is comparable. Now, I am going to check what could be the equivalence checking method here. What are the problems here? Although we have two FSMDs did the same problem that we have solved for scheduling verification allocation binding verification and so on or it is something different right. So, let us try to understand that.

(Refer Slide Time: 09:43)



So, let us try to understand how the C and RTL-C which is RTL-C is something the C that is extracted from RTL ok. So, let us try to understand that. And for that, I actually will see you some a real example that our tool actually generates from a benchmark ok. So, let me just show you that.

(Refer Slide Time: 10:05)

Say I have this GitHub is publicly available. You can go into this FastSim if you just search FastSim, you will get this GitHub.

(Refer Slide Time: 10:16)

So, let us take a very simple example ok. So, this is a waka example which has this is a behavior right, this is my input C code ok. So, you can see here there are a lot of operations which are happening; if-else and it has these are the inputs right. So, there are 32 I mean so, in1 to in32. There are certain inputs and there are three outputs ok.

Now, if I run this particular code using Vivado HLS tool, I will get this RTL ok. You can see here fortunately the inputs are the same right. So, in1 to in32 the same inputs. It has all the same outputs out13, out30, and out31 right. The only thing it has some valid signal which is basically say it which time that particular output becomes valid right. So, that something is there.

So, I can conclude here that the input and output specifications are exactly the same. So, here, you can see the input variables are t_1 , t_2 , t_3 ; these are the temporary variables that are used in this program right. What about here. It has you have to check the registers right.

(Refer Slide Time: 11:22)

So, you can see here that there are so many registers right and. So, wire is basically an intermediate signal. We do not have to bother about that. But if you see here, the register some like reg_361 some t61; you can see here there are many registers.

So, we can see that these internal variables are different. Input-output are same for both Verilog and RTL, but the intermediate variables are different because here intermediate variables are the temporary variables of the program and here it is basically the registers right.

And why I am talking about registers is because if you remember the rewriting method, we actually rewrite everything in terms of registers right. Finally, in the FSMD this wire does not exist. Everything will be in terms of registers ok.

(Refer Slide Time: 12:13)

So, now you see here, this is the while I will talk about that. So, this is the FSM controller FSM right. you can see here that in state 1; if such condition hold, then I will go to this state otherwise, I will go to this state, and so on. In this state, I am going to do this. This is my controller FSM the way I just explained right.

(Refer Slide Time: 12:24)

And these are the operations that are actually enabled state-wise. For example, if you see in the state 6, I am going to do this signal equal to 1 right. So, similarly, in this particular state 3, I am going to do these operations. Here, this is basically a register transfer operation.

(Refer Slide Time: 12:46)

```

// File: source_c_code/nis_macc.c
// Line 33: 30 slices, 927 Bytes
// Author: ozajpy2007 des + nips + motion
// Date: 2020-11-17 12:46:00
// Commit: abe1a1c on Nov 17, 2020
// History: 1 contributor

33 lines (30 slice) 927 Bytes

// File: source_c_code/nis_macc.v
// Line 100: 477 bytes
// Author: ozajpy2007 des + nips + motion
// Date: 2020-11-17 12:46:00
// Commit: abe1a1c on Nov 17, 2020
// History: 1 contributor

33 lines (30 slice) 927 Bytes

// File: source_c_code/nis_macc.c
// Line 33: 30 slices, 927 Bytes
// Author: ozajpy2007 des + nips + motion
// Date: 2020-11-17 12:46:00
// Commit: abe1a1c on Nov 17, 2020
// History: 1 contributor

33 lines (30 slice) 927 Bytes

// File: source_c_code/nis_macc.v
// Line 100: 477 bytes
// Author: ozajpy2007 des + nips + motion
// Date: 2020-11-17 12:46:00
// Commit: abe1a1c on Nov 17, 2020
// History: 1 contributor

```

So, now, from this, I will keep doing rewriting and there is this assignment statement which is always happening always true right.

(Refer Slide Time: 12:55)

```

// File: source_c_code/nis_macc.c
// Line 33: 30 slices, 927 Bytes
// Author: ozajpy2007 des + nips + motion
// Date: 2020-11-17 12:46:00
// Commit: abe1a1c on Nov 17, 2020
// History: 1 contributor

33 lines (30 slice) 927 Bytes

// File: source_c_code/nis_macc.v
// Line 100: 477 bytes
// Author: ozajpy2007 des + nips + motion
// Date: 2020-11-17 12:46:00
// Commit: abe1a1c on Nov 17, 2020
// History: 1 contributor

33 lines (30 slice) 927 Bytes

// File: source_c_code/nis_macc.c
// Line 33: 30 slices, 927 Bytes
// Author: ozajpy2007 des + nips + motion
// Date: 2020-11-17 12:46:00
// Commit: abe1a1c on Nov 17, 2020
// History: 1 contributor

33 lines (30 slice) 927 Bytes

// File: source_c_code/nis_macc.v
// Line 100: 477 bytes
// Author: ozajpy2007 des + nips + motion
// Date: 2020-11-17 12:46:00
// Commit: abe1a1c on Nov 17, 2020
// History: 1 contributor

```

(Refer Slide Time: 13:03)

So, I am going to. So, for example, if you search this temp 294 p2. So, let us go here, you can see here 294 p2. So, it is getting replaced by this right. Then, I have to retake this and you have to take I will find operation which is basically we will replace this right.

So, this way it will be rewriting operation is happening. So, this is the RTL. It is actually generated where this assigned statement at the wire assignment is always true and these are the register transfer operations initiation happening state-wise right.

(Refer Slide Time: 13:28)

So, from this behavior, we are able to generate a C code right.

(Refer Slide Time: 13:33)

The screenshot displays two GitHub repository pages. The left page is for 'FastSim/benchmark_examples/Bambu/waka/source_c_code/his_macc.c' and the right page is for 'FastSim/benchmark_examples/Bambu/waka/generated_c_code/code_his_macc.c'. Both pages show the same commit history from 'jayarajshw' on November 17, 2020. The left page contains 33 lines of C code, while the right page contains 17 lines of C code, indicating significant code generation.

So, this is my generated C code. Let me look into that.

(Refer Slide Time: 13:35)

The screenshot displays two GitHub repository pages. The left page is for 'FastSim/benchmark_examples/Bambu/waka/source_c_code/his_macc.c' and the right page is for 'FastSim/benchmark_examples/Bambu/waka/generated_c_code/code_his_macc.c'. Both pages show the same commit history from 'jayarajshw' on November 17, 2020. The right page contains 342 lines of C code, which is significantly more than the 33 lines in the left page, demonstrating extensive code generation.

So, in this code; so, as I mentioned that I am going to define every variable as an unsigned long int. So, these are the variables and you can see here the inputs are basically the same inputs.

(Refer Slide Time: 13:51)

And you can see it here. So, many variables have to define right and I just initialize them to 0 right.

(Refer Slide Time: 13:55)

So, these are the all variables that is there in the Verilog. So, we have defined it.

(Refer Slide Time: 14:01)

And this is the behavior. In state 1, I just assign that old to new things that I have already discussed that what data inconsistency and the these are the operation which happening in state 1 right.

(Refer Slide Time: 14:09)

So, under this condition, I am going to states 2. Under this condition, I am going to go to about state 1 and so on. In the state 2, again old to new assignments new to old assignments and then, these are the operation is happening right. So, then this is state 3 and so on.

(Refer Slide Time: 14:26)

FastSim/macc at master · x +

→ https://github.com/clarifai/FastSim/blob/master/benchmark_examples/Bamboo/waka/source_c_code/his.macc

If master · FastSim · benchmark_examples · Bamboo · waka · source_c_code · His · his.macc · Go to file · ...

zajayi2007 des mips - motion · Latest commit ethwya on Nov 17, 2020 · History

1 contributor

33 lines (30 sloc) 827 Bytes Raw Blame Copy

```
1 //include<stdio.h>
2
3 int His_main(int argc, char *argv[])
4 {
5     int i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z;
6     int a[10], b[10], c[10], d[10], e[10], f[10], g[10], h[10];
7     int max1, max2, max3, max4, max5, max6, max7, max8;
8
9     //printf("Input matrix A:\n");
10    for(i=0; i<10; i++)
11    {
12        for(j=0; j<10; j++)
13        {
14            a[i][j] = rand() % 100;
15        }
16    }
17
18    //printf("Input matrix B:\n");
19    for(i=0; i<10; i++)
20    {
21        for(j=0; j<10; j++)
22        {
23            b[i][j] = rand() % 100;
24        }
25    }
26
27    //printf("Input matrix C:\n");
28    for(i=0; i<10; i++)
29    {
30        for(j=0; j<10; j++)
31        {
32            c[i][j] = rand() % 100;
33        }
34    }
35
36    //printf("Input matrix D:\n");
37    for(i=0; i<10; i++)
38    {
39        for(j=0; j<10; j++)
40        {
41            d[i][j] = rand() % 100;
42        }
43    }
44
45    //printf("Input matrix E:\n");
46    for(i=0; i<10; i++)
47    {
48        for(j=0; j<10; j++)
49        {
50            e[i][j] = rand() % 100;
51        }
52    }
53
54    //printf("Input matrix F:\n");
55    for(i=0; i<10; i++)
56    {
57        for(j=0; j<10; j++)
58        {
59            f[i][j] = rand() % 100;
60        }
61    }
62
63    //printf("Input matrix G:\n");
64    for(i=0; i<10; i++)
65    {
66        for(j=0; j<10; j++)
67        {
68            g[i][j] = rand() % 100;
69        }
70    }
71
72    //printf("Input matrix H:\n");
73    for(i=0; i<10; i++)
74    {
75        for(j=0; j<10; j++)
76        {
77            h[i][j] = rand() % 100;
78        }
79    }
80
81    max1 = a[0][0]; /* = output node */
82    max2 = b[0][0]; /* = output node */
83    max3 = c[0][0]; /* = output node */
84    max4 = d[0][0]; /* = output node */
85    max5 = e[0][0]; /* = output node */
86    max6 = f[0][0]; /* = output node */
87    max7 = g[0][0]; /* = output node */
88    max8 = h[0][0]; /* = output node */
89
90    return 0;
91 }
```

FastSim/macc at master · x +

→ https://github.com/clarifai/FastSim/blob/master/benchmark_examples/Bamboo/waka/generated_c_code_his_macc.c

If master · FastSim · benchmark_examples · Bamboo · waka · generated_c_code · His · his.macc · Go to file · ...

zajayi2007 des mips - motion · Latest commit ethwya on Nov 17, 2020 · History

1 contributor

105 lines (100 sloc) 2.02 KB Raw Blame Copy

```
1 //include<stdio.h>
2
3 #include<sys/types.h>
4 #include<sys/stat.h>
5
6 #include<sys/types.h>
7 #include<sys/stat.h>
8
9 #include<sys/types.h>
10 #include<sys/stat.h>
11
12 #include<sys/types.h>
13 #include<sys/stat.h>
14
15 #include<sys/types.h>
16 #include<sys/stat.h>
17
18 #include<sys/types.h>
19 #include<sys/stat.h>
20
21 #include<sys/types.h>
22 #include<sys/stat.h>
23
24 #include<sys/types.h>
25 #include<sys/stat.h>
26
27 #include<sys/types.h>
28 #include<sys/stat.h>
29
30 #include<sys/types.h>
31 #include<sys/stat.h>
32
33 #include<sys/types.h>
34 #include<sys/stat.h>
35
36 #include<sys/types.h>
37 #include<sys/stat.h>
38
39 #include<sys/types.h>
40 #include<sys/stat.h>
41
42 #include<sys/types.h>
43 #include<sys/stat.h>
44
45 #include<sys/types.h>
46 #include<sys/stat.h>
47
48 #include<sys/types.h>
49 #include<sys/stat.h>
50
51 #include<sys/types.h>
52 #include<sys/stat.h>
53
54 #include<sys/types.h>
55 #include<sys/stat.h>
56
57 #include<sys/types.h>
58 #include<sys/stat.h>
59
60 #include<sys/types.h>
61 #include<sys/stat.h>
62
63 #include<sys/types.h>
64 #include<sys/stat.h>
65
66 #include<sys/types.h>
67 #include<sys/stat.h>
68
69 #include<sys/types.h>
70 #include<sys/stat.h>
71
72 #include<sys/types.h>
73 #include<sys/stat.h>
74
75 #include<sys/types.h>
76 #include<sys/stat.h>
77
78 #include<sys/types.h>
79 #include<sys/stat.h>
80
81 #include<sys/types.h>
82 #include<sys/stat.h>
83
84 #include<sys/types.h>
85 #include<sys/stat.h>
86
87 #include<sys/types.h>
88 #include<sys/stat.h>
89
90 #include<sys/types.h>
91 #include<sys/stat.h>
92
93 #include<sys/types.h>
94 #include<sys/stat.h>
95
96 #include<sys/types.h>
97 #include<sys/stat.h>
98
99 #include<sys/types.h>
100 #include<sys/stat.h>
```

So, what we have seen here. So, this is my input program which is basically of 30 lines right.

(Refer Slide Time: 14:31)

Here, let us see the code. It is basically 380 lines. So, you can understand this is something completely different structure. The variables are different operations are different control structures are different. So, this is. So, we have to check the equivalence between this five this code and this code that is the problem statement for us and this is a very simple code.

I open the smallest test case available to us ok. So, let us now move to the presentation. So, what we understood from a practical example is this once you have the C and RTL-C, we have the inputs and outputs are the same right; both have the same input-output and there the name is also the same. So, that correlation is available to us.

Control structures are different. They get changed due to scheduling and the internal variables are completely different. So, for the C program, the internal variables are the variables and for this, this registers right. So, that is something that is the given thing to us.

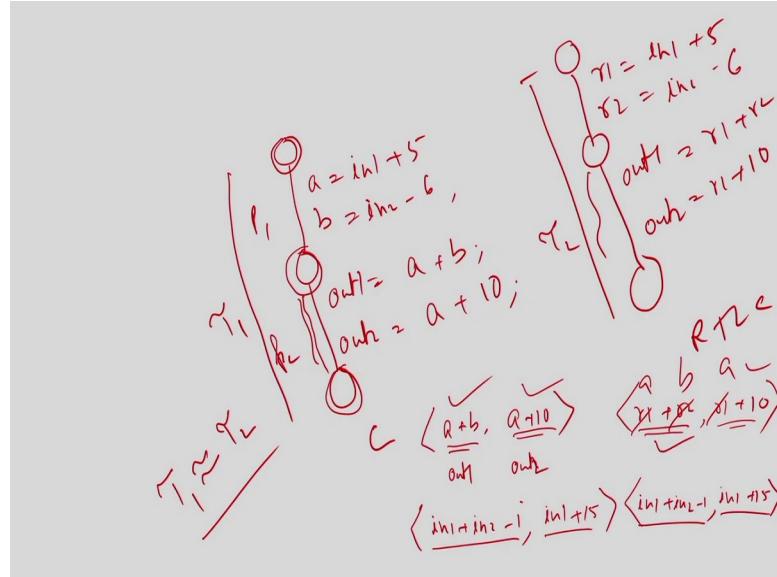
Now, if you just remember that for this that path-based equivalence checking method that we have already discussed in the previous class that we have talked about that this the basic equivalence one. When we can say two FSMDs are equivalent that when you say for every trace or every computation of one behavior, there is an equivalent computation right.

That means you give an input you get a trace one execution trace and you find the equivalent trace in other behavior and you do for all possible traces right. So, this is n square kind of comparison. And we argued that, because dynamic loop bound when the loop bound is not static, it is in input dependent. You might have a very large number of these traces.

So, that equivalence checking is a difficult problem right. And because of that, we actually insert cut points right. We break the problem into smaller problems by introducing cut points in the behavior and then, we identify the path covers. The path covers is a path between the cut points, not the complete trace.

And then, we come up with an equivalence checking method that actually does not identify the trace level equivalence but rather path level equivalence for every path of this path cover, we try to find the equivalent path in the other FSMDs. And since this path cover is a finite representation. So, this equivalence checking problem becomes scalable right. And it has the advantage that it can actually handle loops of dynamic bound right. Now, let us try to see. Here, if I try to do a similar approach and what is the problem right.

(Refer Slide Time: 17:08)



So, say suppose I take a simple example. So, suppose I have a path here. So, this is my say path 1, where say $a = in_1 + 5$ and $b = in_2 - 6$ right and this is my path 2.

Here, I am doing say $out_1 = a + b$ and $out_2 = a + 10$ right and say this I decided a cut point right and this is a cut point say. So, these are the say cut points.

So, there are two paths and according to our previous definition, I want to check the equivalence of this path right. So, I will take this path. Now, let us consider the corresponding RTL-C behavior. This becomes $r_1 = in_1 + 5$. This becomes $r_2 = in_2 - 6$ and there this is $out_1 = r_1 + r_2$ and then, $out_2 = r_1 + 10$ right this is my RTL-C and this is my C.

So, now if you try to compare this path with this path. So, what will be the expressions of out_1 . $out_1 = a + b$ right and $out_2 = a + 10$ right. This is the data transformation and say there is no condition. And here, what will be the case. It will be $r_1 + r_2$ and $r_1 + 10$ right this is my out_1 and this is my out_2 .

Now, problem is that once you try to find this path equivalent to this path, you have to compare this expression right you have to say the data transformation are equivalent; that means, this expression $\langle a+b \rangle$ and this expression $\langle r_1+r_2 \rangle$ is equivalent. This expression $\langle 1+10 \rangle$ and this expression $\langle r_1+10 \rangle$ is equivalent.

But here, I cannot compare, because they are using the intermediate variables a and b here it is r1 and r2 and I have no information whether a is r1 or b is r2 or not right. So, I cannot just compare. So, that is a big problem right. So, the problem is that this cut point-based method will not work for this, because when you take an intermediate path, my data transformation or the expressions will be in terms of the intermediate variables and the intermediate variables are completely different right.

So, I cannot just compare these two paths. So, that will not work, but now let us take the complete trace. So, this is the trace right. So, this is trace τ_1 and this is trace τ_2 . Now, let us try to explain x again and try to rewrite out1 in terms of inputs right. So, now, the rewriting will happen. Now, my out1 = a + b, where a=n1+5 and b=in2 - 6.

So, out1= in1 + in2 - 1. This is my out1 and this out2 =a+10, will be out2 = in1 + 15 right. And $\langle r1+r2, r1+10 \rangle$ here, again it will become $\langle in1 + in2 - 1, in1+15 \rangle$. So, once you take a trace, I can actually represent my output in terms of the input and constant right.

So, now I can say these expressions are the same. These expressions and this expression same. So, I will say take it $\tau_1 \equiv \tau_2$. So, the point that I try to make here is that for this problem when we have the C and RTL-C; since my intermediate variables are completely different. So, I cannot just compare by intermediate paths, because their expression or representation will be in terms of the intermediate variable and there is no relation available to us.

On the other hand, when you take a trace; a trace is something the complete path from the input to output from the start node to the end node right. So, for trace, I can represent my output in terms of inputs only, because the intermediate variables are not defined they are always get defined and used. So, they will be replaced by the inputs right. So, finally, these output expressions will be in terms of the inputs and then I can compare right.

So, the important point that is very important is that whatever the path-based approach that we have talked about for phase-based verification, we cannot apply it here , because the intermediate variables correlation is not available right. So, we cannot just do that. We have to do that trace level equivalence right.

So, we have seen that to compare to FSMDs, we can say that for every trace there is an equivalent trace and vice versa right. So, that is something we have to do here. There is no alternative until we have this correlation is given to somebody. If you give that this variable r_1 is nothing, but a and r_2 is nothing, but if this information is available with. I can replace this with a . I can replace it with this b . I can replay by this and then I can say this expression is equal to this right.

So, if this relation is available with us, we can actually go for that scalable cut point base equivalence checking or path-based equivalence checking. Whereas in this particular problem that is since this I assume that there is no intermediate information is available this correlation is not available, I have to go for trace level equivalence ok.

So, for that the [FL]; obviously, we have to assume that the loop bounds are static, because if the loop bound is dynamic I so, then I do not know how many traces are there. So, this particular if. So, if this bound is not given and if I say this is a integer. So, I have to run for 2 to the power 32 iterations of the loop right. Or if it is long int, it is 2 to the power 64 iterations to the long int. So, those many number of traces will be there.

So, we cannot assume that. So, we will assume that the loop bounds are static and it is a very strict bound that your loops cannot have a dynamic bound. It will always say, i to 10 or i to 100 i to 10000, but it is not i to n , n is an input ok. So, that must be there otherwise, the number of traces we cannot enumerate right that is the problem.

So, now I try to argue. So, this is the first difficulty that comes with making the C to RTL equivalence checking. And now, I try to argue whether this static loop bound is very restrictive for high-level synthesis because we try to do it not for general programs right. We are not solving the general program equivalence rather we are trying to solve the equivalence for high-level synthesis.

So, let us try to understand whether this is a too restrictive bound for high-level synthesis or not. My argument is no, why? If you remember in high-level synthesis, we cannot we do not support dynamic memory access right dynamic memory allocation we do not access. So, the malloc calloc; those will not thing is not allocated.

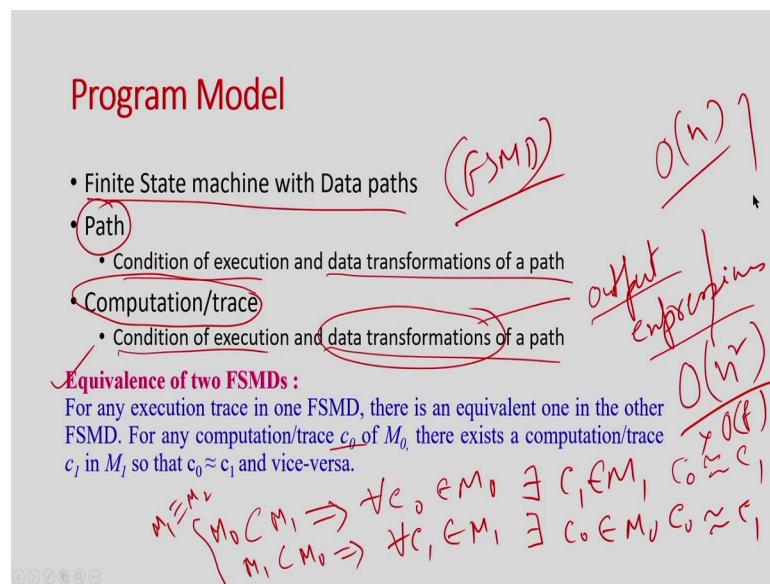
So, the array is basically a fixed bound right. So, any array we cannot have we can just malloc during the program execution because that dynamic memory allocation does not

support by high-level synthesis right. So, it means that the arrays are of static bound or fixed size right. Now, usually in the for loops, what we do and this for loops usually manipulate arrays most of the time right. And if the array is a fixed size, what is the point of keeping the loop bound dynamic right. So, it will also be static.

So, in general, my argument is that in the context of high-level synthesis, the static loop bound is kind of a very it is not too restrictive assumption rather it is kind of very common right you do not find any for loops in high-level synthesis context most of the time. In this high-level synthesis context, you will not find very rarely test cases for loops have dynamic bound right.

So, most of the cases it will be have a static bound and hence, this is not too restrictive bound. There may be some scenarios where your for loop can be of dynamic bound, but that is very rare scenarios ok. So, although this end-to-end equivalence checking is a trace level equivalence and path levels scanner method cannot be applied. But we found that this which inherently tell us that you cannot have a static loop dynamic loop bound, but we argued here that this that is not a too restrictive assumption ok. Let us now move on.

(Refer Slide Time: 25:56)



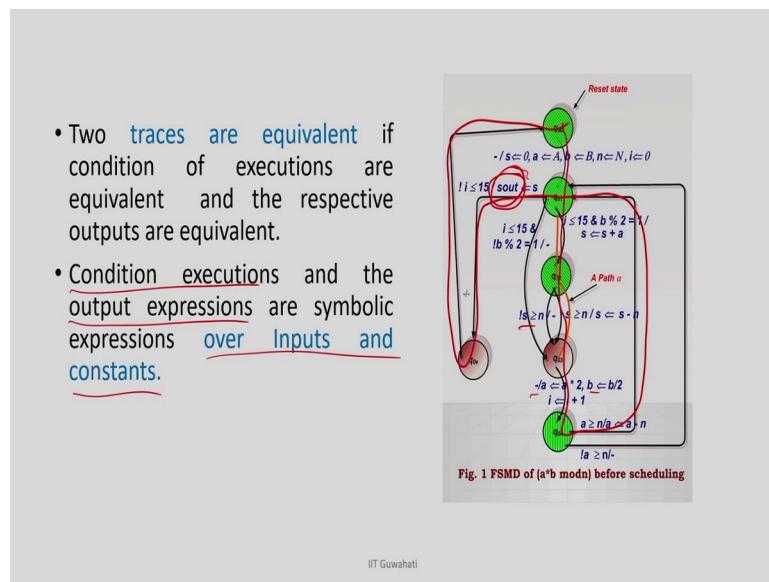
So, program model again we are going to use the FSMD. So, that we have already discussed. So, I am not going into detail of that. That is already discussed in the last

class. And in FSMD, we have paths the similar definition that have taken in the last class.

And for every path, we have a condition of execution and data transformation which basically represent the final values of the every variable in symbolic expressions right in terms of the inputs or variables right. And condition of execution is something; the condition that has to be satisfied by the path.

So, I can actually have. Similarly, I have computation and trace. Trace is basically a path which start from the start node and in the end node is basically one execution of the program right. And similarly, for this compute trace or computation, we can define the condition of execution and data transformation of the path. So, for this once you take that trace, we do not have to check the intermediate variables expression. We have to just check the expression of the output. So, the data transformation is basically the output expressions right.

(Refer Slide Time: 27:04)



So, that is the difference from the earlier cases. So, basically, we say that this, because if you want to take a trace like this right. So, suppose this trace you take. So, here finally, your output will be in terms of the inputs right.

So, I do not have to take the intermediate value of say a 's a b those things I do not need, because those are already intermediate variables. I have to just check once you have a

trace whether the outputs are different equivalent or not. What is the intermediate variable? I do not I do not care alright.

So, for this data transformation, it is not the expression for all variables rather only for the outputs ok. And most importantly in this context, as I already argued that once you take a trace, I can represent this condition of execution and this output expressions of any trace over the inputs and constant.

So, they that the intermediate variable like a b a s (Refer Time: 27:58) those will not come to picture, because that I have already given an example here right. So, I have already argued here, once you take a trace from the start node to end node, my output expression will be in terms of inputs ok that we have already understood.

So, that is something gives up the opportunity to compare this C and RTL-C otherwise, we cannot compare at all right. So, that is the only correlation available and we can actually utilize that and we can compare ok. So, that we understood that for every trace, we will get a condition of execution and output expressions for every output and those are will be in terms of the input variables and constant.

And we will say two trace are equivalent when their condition are equivalent and outputs are equivalent. So, you take two trace you say their conditions are equivalent and the output. So, if there are two output 1 is equivalent for both trace, output 2 is also equivalent for both trace and this is a symbolic expression there is no value right. Then, we will say two trace are equivalent.

And the equivalence of FSMD is already discussed that for every computed trace, there is equivalent trace in other behavior and vice versa. So, you have to make sure that for every trace there is equivalent trace there and for every trace here there is an equivalent trace there ok. So, that is how I am going to prove the equivalence ok.

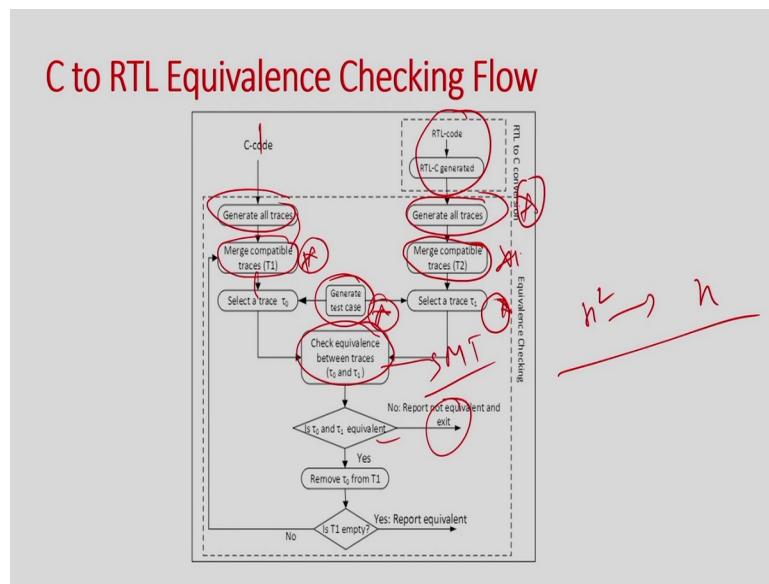
And as I argued that we have to go for this method right. So, basically you have to check for all c_0 of M_0 there exist c_1 in M_1 such that c_0 equivalent to c_1 right. So, M_0 M_1 containing M_1 implies this right. And similarly, M_1 contains in M_0 implies for all c_1 belongs to M_1 , there exist a c_0 in M_0 ; such that c_0 equivalent to c_1 . So, you have to check these two, then I can say M_1 equivalent to M_2 programs right FSMD M_1 and M_2 . So, that.

So, basically what this particular. And we cannot go for this cut point-based method already discussed. So, what we understand from here, that we have to take a trace and I have to identify a corresponding trace. And if there are number of traces basically n . So, you have to take a trace and compare n and you have to check for all n right you have to compare with all n to find out which is equivalent right. So, that is that mean order of n .

And now, if you have to do it for all traces here. So, the complexity will be order of n square of this. This is only the number of comparisons. And for every every two trace, the formula back the formula that I have to check that also involves some computations this into the complexity of formula equivalence right; the way you want to compare the formulas ok.

So. So, this is something is the method that we are going to do, but what we try to say that. So, this is a inevitable for our problem. So, can we do some kind of do some kind of technique or you can apply certain kind of techniques which will improve the efficiency of the whole process ok. So, can I do these things in order of n right; that is the something interesting.

(Refer Slide Time: 31:15)



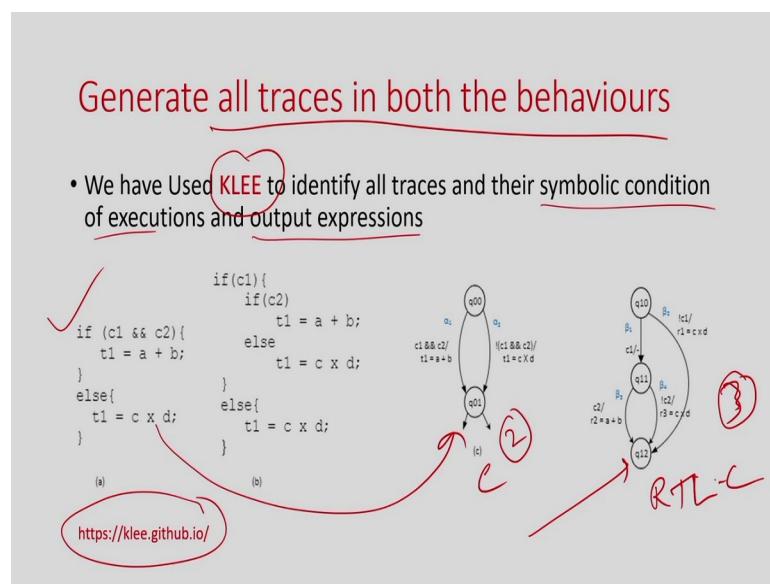
So, we try to come up with certain techniques which is basically tell us to improve the efficiency of this C to RTL equivalence checking ok. So, this is the overall flow. So, this is the RTL to C conversions. The basic idea of here that we will identify.

So, I will let us go into the method first. So, basically what I am going to do, I have to identify all the traces. So, I am going to find out all the traces in both the behavior. This is my input C code this is the RTL. And then, there is interesting things I am going to do is the merge compatible trace here right. So, whatever the traces there, I will try to find out the compatible trace and I will try to merge them into one. So, it will reduce the size. Same thing I am going to do it here right. So, I am not checking anything beyond this.

Then, the process is that you select trace τ_0 here and you find a trace τ_1 here right. So, these things I am going to use a method called the data-dependent method right. So, that is very interesting. So, using that from $O(n^*n)$ comparison, I can reduce into $O(n)$ computation. So, that is data dependency checking. So, I am going to talk about that.

Once I found the corresponding trace, I have to check the equivalence that is. This equivalence checking will be done by the SMT tool right. I will explain that also. And if I found they are actually equivalent, then I will go for the next trace otherwise, I will say [FL] they are not equivalent they may not be equivalent. This is the overall process. The two interesting thing that we are doing here, is this two that merge compatible trace and this data driven finding corresponding trace using data driven approach. So, I am going to talk about the steps of the overall process ok.

(Refer Slide Time: 32:51)



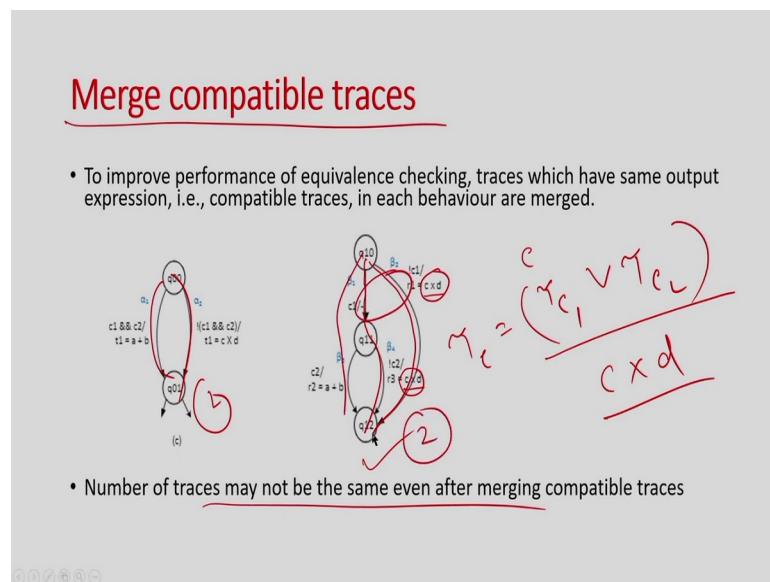
So, first thing is that you identify all traces in one behavior right. So, given a behavior, we have to identify the trace and for trace, you have to identify the condition of

execution and the output expressions. So, we have to do it right. And for that, we have used a tool called KLEE, which is a symbol of symbolic execution tool, and if you give C program, which actually identifies all the traces and their condition of execution and data transformation in symbolically right. So, this is the link for the tool and you can actually try that.

Let us take an example here. Say suppose, this is my input behavior and, because of this conditional scheduling; for this, the representation is this and this is the final RTL-C right say and this is my C. I am not going to explain here how it is happening. So, one thing you can understand here that there are two traces here. I take a very small example to explain and here, there are three traces right.

So, there are two traces here and there are three traces here. So, you can understand here that the number of traces may not be the same in both the behaviors and, because of the conditional optimizations. So, if you just compare for this path you may not find the equivalent path there right equivalent trace there. So, that is a problem right. So, but this is the first step given the behavior, I will identify the number of traces in C and I will identify the number of traces in RTL-C ok this is my step 1.

(Refer Slide Time: 34:14)



Next, is very interesting is a merge compatible trace; what I am going to do. We identify that the number of traces in both the behavior may not be the same ok, but there may be some trace within the behavior. So, I am not taking two behavior now, I am going to take

a single behavior. Say, let us say I took this behavior now. And I try to identify two traces say, this is the trace 1 this is my trace 2. In these two traces, I try to identify if the data output expressions are the same or not right.

So, let us say here the output is basically said $r3$ right. So, I can see here that this $r1=c * d$ this is also $r3 = c * d$ right so; that means, the output of these two traces is actually equivalent right. So, this $r1 = c * d$ and this $r3= c * d$. So, this is basically a compatible trace.

So, basically, we will say these two traces are compatible within a particular program if their output equivalent expressions are the same ok. In that case, I am going to merge this into one. So, how we will do. So, whatever the condition of this path; say, this is a condition is a τc_1 and this τc_2 .

So, this will be my condition of the merge computation right ($\tau c = \tau c_1 \vee \tau c_2$). So, I will just merge into one and the computation of condition of execution becomes or of these two conditions and the expression is same. So, the output expression will remain $c * d$ right. So, this is the first thing I am going to do.

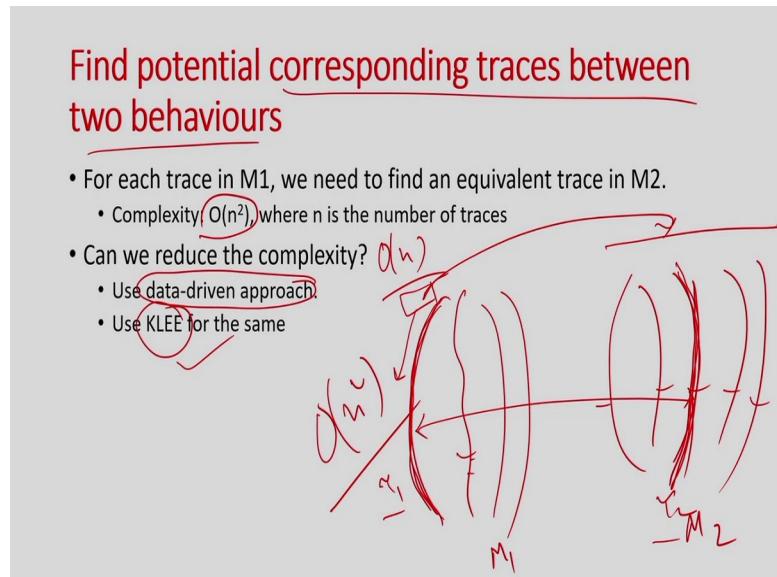
And we found that this is very important otherwise, the complexity of your equivalence checking will grow and we found many cases this kind of compatible traces, because of the conditional transformation this kind of thing happens and you can actually find out the compatible trace and merge them into one right. So, this is the first step I am going to do,.

But we found several scenarios, where the number of traces did even not become same for both the program after merging. For this case, it will become same. There are two trace here, trace1, trace2. Here, this will be trace1 and this union will be trace2. So, there are two traces. So, for this case, the number of traces become the same after merging compatible traces, but in general, the number of the trace may not become same even after this, because of some complex transformation which stop us from merging this compatible trace ok.

So, this is actually a useful trick to reduce the complexity, because whatever I am saying, I have to compare this trace versus trace, but the technique that I am going to propose here will reduce the complexity. So, here, because 2 and 3 I cannot compare. So, I merge

these two traces into one. Now, 2 into 2 so, I can compare them easily right. So, that will reduce the complexity of the equivalence checking.

(Refer Slide Time: 36:51)



Next, I am doing a very interesting thing that is finding potential corresponding traces between two behaviors. So, what I have told you is that you take a trace here, and there are say n number of traces here right in this program 2. This is program 1 this is program 2. So, if you have to compare this with this, this with this, this with this, this with this and this. So, it is basically all n you have to compare right. And I have argued that since there are n number of traces here, it will be $O(n)$ square complexity that I have already talked about right.

So, can I reduce this complexity to $O(n)$? Yes. So, we can do that using a technique called data driven approach. So, idea is very simple. The idea is that you take a trace here ok and you identify a input for which this trace will be execute ok. Now, you apply the same trace to this program right. So, if you give the input to this program, what will happen; one of the trace will execute right not all of them, because if you give an input only one of the trace will execute.

So, now suppose, I took this trace and I take a input which actually make sure this trace will execute. And now, I apply this input to this program and I found that this is the trace that is going to execute for this input. I will say these two traces are potentially equivalent right.

So, now, I do not have to compare with all traces. So, I will just take this trace τ_1 this trace τ_2 and I will just check whether they are equivalent or not. If I found they are equivalent, al then I will say yes; they are equivalent if not, there I say this program may not be equivalent. So, I am not comparing this with all n rather with an input with the help of input, I identify directly the corresponding trace ok.

And. So, this $O(n)$ checking is gone. So, the overall complexity will become now, $O(n)$ not $O(n^2)$ right. So, that is the basic idea of this and we can again use the tool KLEE to do this. So, with KLEE you can actually set some input and it will give you a trace. And then with this input, you can actually identify the corresponding trace right. So, we use the tool and that actually help to reduce the complexity ok. This is the basic idea.

(Refer Slide Time: 39:10)

Checking one to one equivalence

- For each potential corresponding traces, we check if their respective condition of executions and data transformations are equivalent
- Use **SMT Solver** for formally prove the equivalence of potential corresponding traces.

The diagram shows two sets of conditions being checked for equivalence. On the left, there is a handwritten note: "check out UNSAT" with an oval around it. To the right, there are two sets of conditions: $C_{\tau_1} = C_{\tau_2}$ and $O_M = O_M$. Below these conditions, there are two sets of brackets: $\{C_{\tau_1}, \langle O_{\tau_1} \rangle\}$ and $\{C_{\tau_2}, \langle O_{\tau_2} \rangle\}$.

So, once you have this potential trace right. So, I have trace 1 here, I have trace2 right so, τ_1 and τ_2 . The next problem is that you have to compare these two right. So, these two trace whether they are equivalent or not. And this comparison we have two arithmetic expression, we have to compare right. This comparison how can I do.

So, for that we use a tool SMT Solver and with the tool, we use Z3 ok. So, the idea here is that and this SMT Solver is basically just check the satisfiability formula right. So, the equivalence problem of this two paths has to be model as a satisfiability formula for a SMT solver. So, how we can formulate that?

The formula is that for this τ_1 we c_{τ_1} which is the condition of execution and the condition of execution is c_{τ_2} here. And it has O_{τ_1} , which is the output expression for this, and for this it is O_{τ_2} right. These are the things we have. Now, I have to model a satisfiability formula. So, there will be a formula that will be either satisfiable or unsatisfiable right.

And the way I have to model this equivalence problem of a path into the satisfiability formula; is such that when the formula become UNSAT, I will say these two are equivalent ok. So, that is the idea. So, how do I going to model it? So, I will say that $c_{\tau_1} \equiv c_{\tau_2}$ and $O_{\tau_1} \equiv O_{\tau_2}$. So, if we try to you just give this formula and you ask the tool you just find a sat right check sat. What it will do? It will basically check this and we will try to find out an input for which it will this formula is true right.

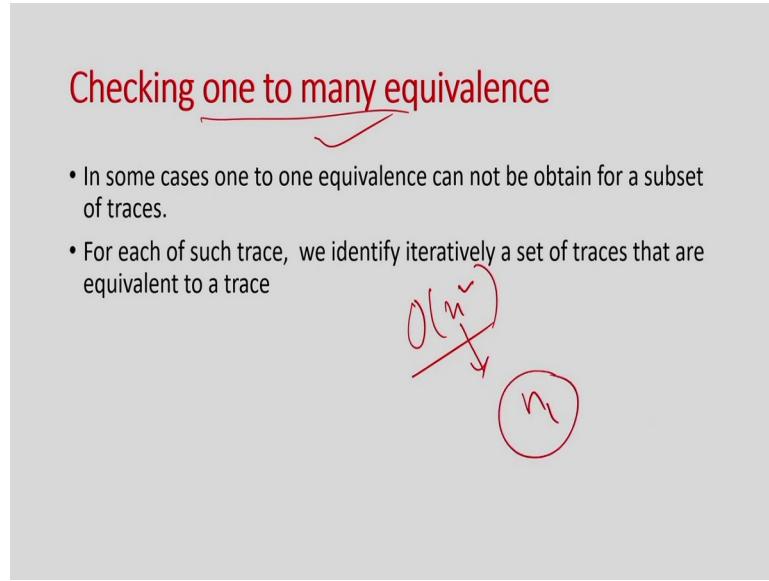
So, now the problem here, is that it was just written one input value for which this formula says it is true ok. So, as a result, you are not actually formally proving the equivalence right. It is just checking for this input this formula is satisfiable or true right. So, what I am going to do? I will take the negation of this formula ok and I will check the UNSAT right.

So, I will now, check the sat again. So, what will happen? I try to; so, what does it mean? It means here, that you find an input for which these two paths are not equivalent right. If the tool finds an input for which these two paths are not equivalent, then indeed it is basically they are not equivalent trace right.

But it is if it is written UNSAT, what does it mean? It means it does not find any input for which this path is not equivalent. What does it mean? It is basically equivalent right. So, I basically take the negation of the satisfiability problem and ask the tool to identify your input for which these two path are not equivalent. And since these two paths, if the paths are equivalent, you will not find any input for which they are actually different right.

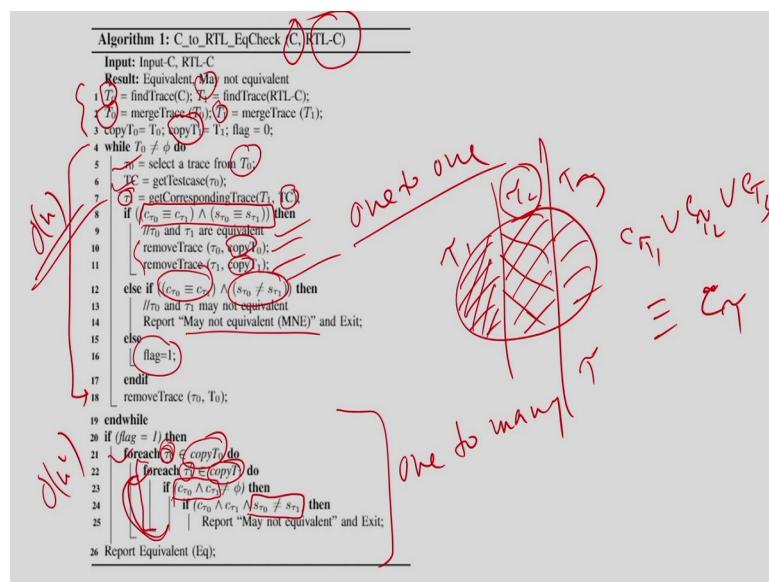
So, it will return. So, this is this formula is UNSAT and the UNSAT of the formula says these two are traces are equivalent. So, this is how I am going to check the equivalence of two traces ok using the SMT Solver.

(Refer Slide Time: 42:31)



And I have seen that in some cases, if the one-to-one correspondence cannot be shown, then what we have to do one-to-many equivalence right and that needs $O(n^*n)$ cross-checking. But we will say we will only do it for very few traces for which one-to-one correspondence cannot be proven ok. So, that is the idea of this. So, in the worst case we have to do this, but this it has to be done for very not for all n right. It will be for some n_1 number of things, where n is very less than n ok.

(Refer Slide Time: 43:01)



So, let us now explain the overall algorithm. So, as I said. So, you identify the traces. We have given two programs; C and RTL-C. You model them as FSMD. You find traces in both the programs and you also, find the traces and then you merge the compatible traces also.

So, you reduce the SAT ok. Then, what I am doing is very interesting. This is this one-to-one correspondence right. So, this is one-to-one checking and this is basically one-to-many ok.

So, in the one-to-one what I am doing. I will just take a trace of this one program and for that, I will take a test case, and then I run the other program with this test case to get a corresponding trace the potential corresponding trace. So, I am not. I am checking $O(n*n)$. You see this is $O(n)$ algorithm and this is $O(n*n)$ algorithm there are two for loops right here only one while loop.

So, I identify a corresponding trace and this is the formula I am just checking right. So, whether that is equivalent or not. If they are equivalent. So, I will say I identify the equivalence. So, I will just remove these two traces, because for them, equivalence is already obtained. And if I found that condition is same, but the data transformation is difficult different the output is different, then I say they are not equivalent.

Because if the condition same, these two traces are basically must be equivalent otherwise, it is a problem, but I found that output expressions are different. So, I will say they may not be equivalent. Otherwise, what I will say. This trace τ_1 I cannot prove the equivalence ok. So, for this $O(n*n)$ is needed right.

So, I will just keep that into this copy of τ_1 right. So, in the copy of τ_1 , I will just remove all the traces for which equivalence is found. And the trace for which equivalence cannot be found in $O(n)$ time I will keep them into copy τ_0 ; this traces as I right. So, this will go on for all tracers.

And for all cases where one-to-one equivalence can be proven, it will be done here right. If it is not done here, the remaining traces will remain here. And for that, you can see I am doing a $O(n*n)$ checking and what I am doing here, I will just So, basically what is going to happen.

So, there is a trace and this is the say the condition space right and there will be multiple trace here. So, this is the condition for one trace here, and there may be one trace which will which condition will be this space in the other program. There will be another trace for which the condition will be this part and there will be another trace for which condition will be this part.

So, these three traces. So, this is a τ_1 , τ_2 and τ_3 , which will be union of these three will be equivalent to τ . So, $\tau = (\tau_1 \cup \tau_2 \cup \tau_3)$ is the complete condition. So, that is what is happening here. So, I will take a trace τ_0 and I will identify a trace τ_1 ; so that their condition is non-null; that means, there is some overlap. So, I will identify this part ok.

And then, I will see if their output expressions are equivalent right. So, what I am going to see. I identify a trace in the other behavior, whose condition is basically a subset of the trace and I will check if their output is equivalent or not. If they are equivalent, then I will say it is fine. If not, they will say they may not be equivalent right.

If they are equivalent, I will identify the next one. So, I identify τ_2 and I will identify their sub trace are equivalent, and then, I will again check their output expressions, if they are the same. I will say they are equivalent move on and then I try to find out the next trace τ_3 .

So, this way this inner loop will identify for every trace τ_1 , it will identify all the traces which is a subset of this condition of τ_1 , and since this program is deterministic, so $c_\tau = (c_{\tau_1} \cup c_{\tau_2} \cup c_{\tau_3})$ right, because this is a deterministic program.

So, this way for every trace, I will identify a set of traces in other programs whose union of them will be equivalent to this trace ok. So, this you remember that again I am not doing for all traces, because in most of the cases the equivalence will be found here. Only very corner cases, I have to do this.

So, this is the overall idea right. So, what I have done here, is that this still trace level equivalence has to be done, but we try to do several kinds of tricks to reduce the complexity ok.

(Refer Slide Time: 47:17)

Bench	#in	#out	C code		RTL code		RTL-C		Traces		Equivalent		Not Equivalent		
			#line	#var	#line	#regs	#line	#var	#C	#RTL-C	#merged	time (s)	result	time (s)	result
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)
Waka	20	3	33	32	270	12	382	126	3	4	(3, 3)	1.709	Eq	0.669s	MNE
Arf	11	4	53	43	351	19	607	158	4	4	(3, 3)	1.890	Eq	0.949	MNE
Parker	6	1	62	14	196	10	275	100	12	23	(2, 2)	1.614	Eq	0.976	MNE
FindMin8	8	1	40	15	175	11	780	243	128	128	(8, 8)	22.246	Eq	17.141	MNE
MatrixAdd	2	1	48	4	734	44	2595	241	1	1	(1, 1)	1.684	Eq	0.749	MNE
SumArray	1	1	19	4	263	15	541	100	1	1	(1, 1)	0.754	Eq	0.706	MNE
Motion	10	3	52	43	413	29	881	235	1	1	(1, 1)	0.681	Eq	0.663	MNE
Dfadd	2	1	719	70	1975	113	9353	1041	67	68	(21,42)	1016.052	Eq	960.238	MNE

So, let us explain some examples here. So, experimented results. So, these are the test case we have taken. As you can see here these examples are moderate size; the number of variables is high and this is the RTL code size and this is the RTL-C. So, it is the number of lines of code is much higher than the C code and we have also seen that think seen in an example right. Interesting things to be noticed here, that the number of registers ok.

So, a number of registers here and this is the number of variables. Here, you can see that this is always less than the number of variables 32 12 43 29 right. So, usually, because of this the register allocation happened. So, most of the cases is basically the number of registers. We will try to optimize the number of registers right. So, it is the number of registers is less than the number of variables.

In some cases it is basically more because this array mapped two registers right. So, there are two cases; the matrix array and the sum array, where this array becomes. An array is a single variable here that is mapped to the register. So, that is why the number of registers more in this scenario otherwise, it is always the same and this is a number of traces.

You can see in many cases, the number of traces are not equivalent right. So, the number of traces are not equivalent we can understand that. So, this is so, that is why this compatible trace is important and the number of traces are not equivalent here also. So,

then we do the merge compatible traces. And after that, this is a number of trace in program1 this is the program2.

We can see here that most of the cases the number of traces becomes equivalent except in one scenario. So, this tells us the usefulness of the merge equivalent trace many cases the number of traces are not same before you compare merging compatible trace, but after that they become simple right.

Most interest; one important thing say that there are 120 after traces and there are 128 traces here. Then, these 128 traces merge and become 8 traces. So, merging is compatible trace helps to reduce the number of traces also because we found that there are numbers of traces, where the outputs are equivalent, I can merge them.

So, the complexity also reduces here. So, this is a very interesting and important step that actually make my problem or this equivalence checking problem scalable right or actually makes it very helpful ok. So, then after that we check. So, we can understand that. So, out of this I think there are 7 or 8 test cases are there; 7 test cases the number of traces based come equivalent. In only one case, it is not equivalent.

So, most of the cases in this scenario $O(n)$ will suffice right. Only this case, I have to go for $O(n*n)$ checking, because the number of tests is not artifice not same. So, this is very helpful right. So, in most of the cases merge compatible trace reduce the problem into a such problem that one-to-one corresponding checking can help right. And we can identify that the number of time taken is very less right.

So, most of the cases are within seconds right. So, the only case where you have to go for n square the time is more; than 1000 seconds right, but still that time is not so high right. So, this, because in every case we are actually using we call the SMT Solver and that takes a lot of time, because of that actually check the formula, then it checks the UNSAT and SAT.

So, it also that this time involves the SMT time as well. So, we can understand that including the SMT time also, the time is also very reasonable. And then we identify may we. Basically, take this example. We introduce certain bugs in the RTL-C and we check the non-equivalent and most of the time we have very quickly also prove the non-equivalence right.

So, it shows that if there is a bug our tool can easily detect it right and if there is no bug, we can show the equivalence ok.

(Refer Slide Time: 51:07)

Summary

- We have proposed an RTL to C translation validation framework for verification HLS results.
- The innovative part of our framework to improve the efficiency
 - Extraction of a RTL-C from the RTL to reduce the semantic gap between C and RTL
 - Merging compatible traces within a behaviour to handle control structure modification during HLS
 - Use of a data-driven approach to find the potential equivalent trace pairs between two behaviours.
- The equivalence of potential equivalence traces are then formally proved using an SMT solver.

So, that is all; so, just to summarize. In this discussion, we propose an RTL to C translation validation framework right for high-level synthesis. So, we have a RTL and C. We try to propose an equivalence. And to make this particular equivalence checking framework, there are certain things is very innovative in our flow.

First of the things is that this extraction of the RTL-C. Unless we have the C, we cannot just compare. So, this RTL to C conversion is very helpful and very interesting in this context. So, that only enables this whole problem. And also, we proposed this merging of compatible traces, which also helps us to reduce the complexity most of the time and is useful to make the number of traces equivalent in or equivalent or same in both programs. So, that is also very useful.

And then, we also use an approach called data-driven approach. So, which actually help us to identify the potential equivalent trace between two programs, in order of n time instead of n square such. So, that also improves efficiency. And finally, the formal you have proved the for equivalence of two trace formally, using that solver SMT tool right this is 3. So, this is the overall idea ok.

Thank you.