DssEC: A Deep State Sequence Based Equivalence Checker

Jian Hu
The Sixty-third Research Institute, National University of Defense Technology, Nanjing 210000, China
Hujian198681@126.com

Yun Kang
The Sixty-third Research Institute, National University of Defense Technology, Nanjing 210000, China
yk@126.com

Yongyang Hu
The Sixty-third Research Institute, National University of Defense Technology, Nanjing 210000, China
yyh@126.com

Haitao Yang
The Sixty-third Research Institute, National University of Defense Technology, Nanjing 210000, China
hty@126.com

Le Tong
The Sixty-third Research Institute, National University of Defense Technology, Nanjing 210000, China
lt@126.com

Jie Cheng
The Sixty-third Research Institute, National University of Defense Technology, Nanjing 210000, China
jc@126.com

Junquan Deng
The Sixty-third Research Institute, National University of Defense Technology, Nanjing 210000, China
djq@126.com

ABSTRACT

Human-guided transformations or a compiler have carried out on some source to source transformations for software or hardware optimizations. Since the compiling process is complex and error prone, there may be some errors in the implementation. Equivalence checking proves the target program be a correct translation of the compiled source program. In this paper, we propose an equivalence checking tool for verification of the source and target programs; Finite State Machines with Datapath (FSMD) is used to model the original and the transformed programs. The tool is based on a deep state sequence (DSS) strategy proposed in our previews work. The false computation problem of value propagation (VP) based method can be solved by our tool. The experiment results show the effectiveness and efficiency of our tool.

CCS CONCEPTS

• **Hardware**; • **Hardware validation**; • **Functional verification**; • **Equivalence checking**;

KEYWORDS

Equivalence checking, System level modeling, Deep state sequence, Register-transfer level

ACM Reference Format:
Jian Hu, Yun Kang, Yongyang Hu, Haitao Yang, Le Tong, Jie Cheng, and Junquan Deng. 2021. DssEC: A Deep State Sequence Based Equivalence Checker. In *The 5th International Conference on Computer Science and Application Engineering (CSAE 2021), October 19–21, 2021, Sanya, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3487075.3487106

1 INTRODUCTION

High-level synthesis (HLS) [1] transforms a source code to a target code for optimizations. Thus, programmers can focus only on the functionality and the correctness of the program being developed. Scheduling maps each operation to a specific clock cycle under some constraints, such as area and/or delay and data dependencies.Code motion (CM) technique in scheduling phase of HLS schedules the operations between different basic blocks, but it must preserve the existing control dependence and all data dependence. CM is widely used in HLS to improve the quality of designs with complex structures and loops. After CM, the data-flow of a behavior may be changed considerably. Hence, it is indispensable to check the equivalence between the source behavior and the scheduled one generated by HLS.

Equivalence checking is a technique that verifies the transformed program generated by the HLS is a correct translation of the source program. Although the equivalence checking technique can not prove the HLS tool bug-free, it guarantees the translation process is correct after HLS. Our work in this paper proposes an equivalence checking tool for code motion transformations.

There are several related works in recent years addressing the equivalence checking problem in high-level synthesis. Some path-based equivalence checking methods are presented in [2-7] for verification of FSMD models. Since a path cannot be extended beyond a loop, all these methods fail to deal with the case of code motion across loops and loop invariant code motion in nested loops. A state-of-the-art DSS based equivalence checking method proposed in [8] can handle the code motions across loop bodies. This is accomplished by comparing the whole path instead of path segments between cut-points.

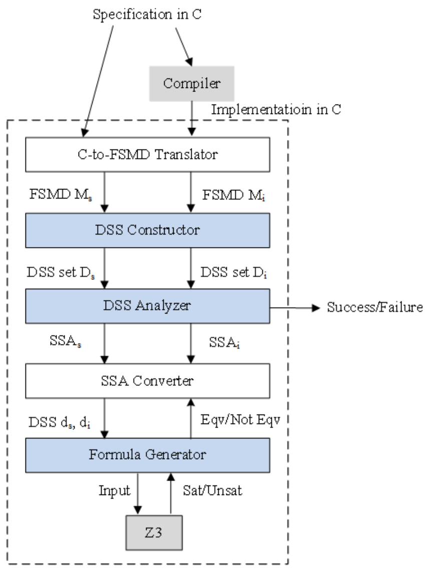


Figure 1: Framework of Equivalence Checking.

In the earlier work [8], the author Hu shows the detail of their equivalence checking method for validating the correctness of transformations between the target program and the source program. However, that paper does not report the equivalence checking tool. This paper describes the verification tool named ***DssEC*** presented in [8], which is based on pycparser [9], a complete parser of the C language, written in pure Python.

The rest of this paper is organized as follows. In section 2, the tool architecture is presented. Section 3 shows the functionalities of modules in the tool. In section 4, an example illustrating our method is presented. Some experimental results is shown in section 5. Section 6 concludes our work.

2 TOOL ARCHITECTURE

Figure 1 shows the framework of the equivalence checker. The blue boxes denote those modules which are implemented in C, the white boxes are implemented in python and the other colors boxes are external to our tool. We will briefly describe each of the modules shown in the figure below.

The inputs to the equivalence checker are the original program in C and an optimized program also in C produced by a compiler. In the equivalence checker, a C-to-FSMD translator implemented based on pycparser is first used to obtain the corresponding FSMDs M_s and M_i of the specification and the implementation. The DSS set D_s and D_i of the respective FSMDs M_s and M_i are then obtained by a DSS construction module implemented in C. The DSS analyzer module tries to find corresponding equivalent DSS pairs (DSS d_s in D_s and DSS d_i in D_i). The equivalence checker outputs ***success***, if all the DSS in D_s can find their equivalent DSS in D_i. The equivalence checker outputs ***fail***, if any DSS in D_s fail to find their equivalent DSS in D_i. Symbolic simulation is used to obtain the formula to validate the equivalence of the two compared DSS. After the formula generation, the codes of the two DSS are converted into static single assignment (SSA) form [10]. The formula is fed to an SMT solver Z3 [11] to obtain the results.

3 FUNCTIONALITIES OF MODULES

The tool has been implemented in python and C respectively. The modules ***SSA Converter*** and ***C-to-FSMD Translator*** have been carried out using python. The ***DSS Analyzer*** and ***DSS Constructor*** and ***Formula Generator*** have been implemented in C. The five modules of the core system is depicted in Figure 1

3.1 C-to-FSMD Translator

FSMD is a widely used intermediate form for sequential programs, since it is not difficult to extract FSMDs from the programs [12]. There are three basic nodes for any sequential program: (i)sequential statements in a basic block without any bifurcation of control flow, i.e., assignment node, (ii) branch node, and (iii) loop node. The module ***C-to-FSMD Translator*** can generate the FSMD from any C program. It traverses through the abstract syntax tree (AST) and generate FSMD states based on the type of AST node.

At the beginning of the program, our tool creates a start node. When an assignment node in a basic block is encountered, no new node is created and only the assignment operation is stored. When a branch node is encountered, two FSMDs are constructed corresponding to ***true*** branch BB_1 and ***false*** branch BB_2 first. The branch node is obtained from these two FSMD. First, the start states of two FSMD are merged into one start state and the end states of two FSMD into one end state. Second, the condition c is placed as the condition of BB_1 and the condition $\neg c$ is placed in the transition of BB_2. When a loop node is encountered, the FSMD M of the loop body is constructed. The the loop node is obtained as follows. First, the start and the end state of M are merged into one state q_0, and placing the condition c in the transitions from q_0. Second, a transition from q_0 with condition $\neg c$ is added as the exit path from the loop in the FSMD.

We formally define the node processing rules in an AST as follows:

Assignment node, e1=e2:
operation+=" e1=e2;"
For node, ***for(e1;e2;e3)***:
fsmd="-/"+operation+',+newstate1;
fsmd ='('+e2+')'+"/-,"+newstate2+",!("+e2+")/-,"+ newstate3;
fsmd ="-/"+operation+',+newstate1;

While node, ***while(e1)***:
fsmd="-/"+operation+',+newstate1;
fsmd='('+e1+')'+"/-,"+newstate2+",!("+e1+")/-,"+ newstate3;
fsmd="-/"+operation+',+newstate1;

If+else node, ***if(e1) ; else e2***:
fsmd="-/"+operation+',+newstate1;
fsmd='('+e1+')'+"/-,"+newstate2+",!("+e1+")/-,"+ newstate3;
fsmd= "-/"+ operation+',+newstate4;
fsmd= "-/"+ operation+',+newstate4;

The variable ***newstate*** is a new generated state of the FSMD. The variable ***operation*** is used to store assignments on the edge in the FSMD. The variable ***fsmd*** is used to store the resulting FSMD structure.

3.2 DSS Constructor

DSS Constructor tries to produce all the DSS of both FSMDs in a depth-first manner from the respective reset states of the FSMD M_1 and M_2 as shown in Algorithm 1.

Algorithm 1 DSSGenerator(*seq*, *num*, *list*, *p*, *j*)

```
1: $Seq[num][j] \leftarrow p.vertex$
2: if $InSeq(p.vertex, Seq[num], j-1)$ then
3:     $p_n \leftarrow list[p.vertex].next$
4:     while $p_n \neq Null$ do
5:         if $InSeq(p.vertex, Seq[num], j-1)$ then
6:             $p_n \leftarrow p_n.next$
7:         else
8:             $Generate\_Seq(p_n, Seq[num], list, j+1)$
9:         end if
10:    end while
11: else
12:    $p_n \leftarrow list[p.vertex].next$
13:    if $p_n \neq Null$ then
14:        $Generate\_Seq(p_n, Seq[num], list, j+1)$
15:    end if
16: end if
17: $p \leftarrow p.next$
18: while $p \neq Null$ do
19:    if $InSeq(p.vertex, Seq[num], j-1)$ then
20:        $p \leftarrow p.next$
21:    else
22:        $num \leftarrow num + 1$
23:        $Generate\_Seq(p, Seq[num], list, j)$
24:    end if
25: end while
26: $Return\ Seq$
```

Recursive method is used to generate the DSS. During the recursion process, the algorithm stores the unrepeated paths. DSS and its number are stored in variable ***Seq*** and variable ***num***. The current FSMD node and its number are stored in variable ***p*** and variable ***j***. The algorithm traverses the AST in a depth-first manner and stores the unvisited nodes determined by the function ***InSeq()***.

3.3 DSS Analyzer

The DSS analyzer module tries to find corresponding equivalent DSS pairs (d_s in D_s and DSS d_i in D_i). The equivalence checker outputs ***success***, if all the DSS in D_s can find their equivalent DSS in D_i. The equivalence checker outputs ***fail***, if any DSS in D_s fail to find their equivalent DSS in D_i.

First, module ***DSS Analyzer*** will select a d_s from the DSS set if the D_s is not empty. Second, automatic test vector generation technique (ATVG) [13] is used to generate test vector for the selected d_s. If no tests can satisfy the d_s, our tool will take it as a false computation and delete it from the DSS set. In the end, our tool applies the generated test to the transformed program to obtain the corresponding d_i.

3.4 SSA Converter

In an SMT solver, all the statements are represented as asserts. However, the asserts do not capture the order of execution of the statements . Hence, the code needs to be transformed into its equivalent SSA form. Next, the asserts are generated by the formula generator as the input for SMT solver. In the converting process, our tool first creates a dictionary to store the variable and its subscript. The subscript of the variable increases only when the variable is assigned a value. If our tool meets an assignment statement and the variable assigned a value is not in the dictionary, it is added to the dictionary and assign 0 as its subscript. If the variable is already in the dictionary, we add the subscript of it by one. Figure 2 shows the program snippet, its equivalent SSA form and the dictionary in the algorithm.

```
x = a + b;        x0 = a + b;
x = x + a;        x1 = x0 + a;        Dic={x:1,z:0}
z = x + y;        z0 = x1 + y;

   (a)               (b)                  (c)
```

Figure 2: (a) Original Program; (b) Program in SSA Form; (c) Generated Dictionary.

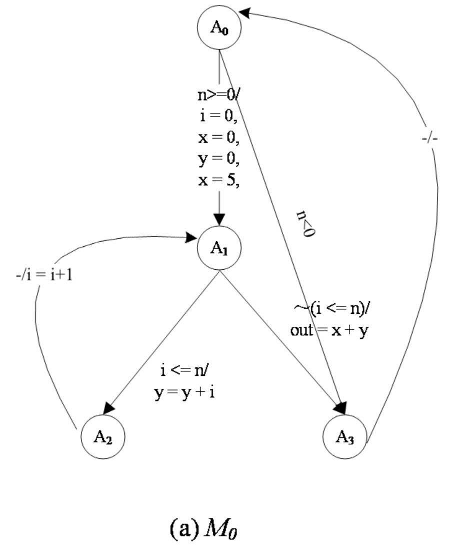
3.5 Formula Generator

The ***Formula Generator*** is used to obtain the formula to validate the equivalence of the two compared DSS. First, the conjunction expression of SSA of the compared DSS-pairs are generated using symbolic simulation technique. Then an SMT solver is used to solve the conjunction expression to prove the equivalence.

4 RUNNING EXAMPLE

Figure 3 is an example for illustration of our approach. Figure 3 (a) and (b) represent FSMD models of C program before and after code motion across loop. The state name is represented as A_i and the state transition condition e_1 and update function e_2 are the form e_1/e_2. Our tool first generates the AST of the example as bellow.

```
FileAST:
FuncDef:
Decl: main, [], [], []
FuncDecl:
TypeDecl: main, []
IdentifierType: ['int']
Compound:
Decl: out, [], [], []
TypeDecl: out, []
IdentifierType: ['int']
. . . . . . . . . . . ..
.. . . . . . . . . . . ..
While:
BinaryOp: <=
ID: i
ID: n
Compound:
Assignment: =
ID: y
BinaryOp: +
ID: y
ID: i
Assignment: =
ID: i
BinaryOp: +
ID: i
Constant: int, 1
. . . . . . . . . . . ..
. . . . . . . . . . . ..
Return:
ID: out
```



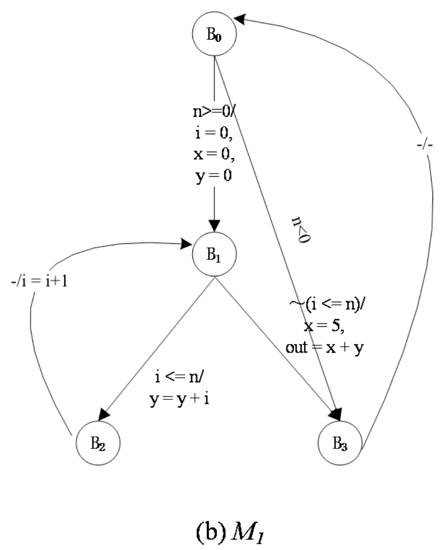


Figure 3: (a) Original FSMD; (b) Transformed FSMD.

Next, ***C-to-FSMD*** Translator generates the FSMD of the program. The output of our tool is in the bellow. The number in the first column is the source state and the number after the comma is the destination state. The operations in the same transition are separated using semicolon. Since the symbol '$/$' is used to represent the division operation, '$/$' is replaced by '$|$' to separate the condition and data transformation of execution. The negation of a condition is represented using symbol '$!$' (instead of $\neg$). The symbol '$-$' represents no operation as the data transformation of an execution and ***true*** as the condition of execution. It should note that the number of transitions from a state to another in an FSMD has no limitation, as long as the conditions of executions are mutually exclusive from each other. And the final state loops back to the reset state with ***true*** condition and no operation.

```
The generated FSMD is:
0: '(n>=0) |x0=0; y0=0; i0=0;,1', !(n>=0)|out0=-1;,3',
1: '(i0<=n) |y1=y0+i0; x1=5,2, !(i0<=n)|out0=x1+y1;,3',
2: '-|i1=i0+1;,1',
3: '-|-,0'
```

Next, module ***DSS Constructor*** extracts DSS from the FSMD generated from ***C-to-FSMD*** . The generated DSS is shown below, in which "A_i" is the state name of the FSMD.

```
The generated DSS is:
A0->A1->A2->A1->A3->A0
A0->A1->A3->A0
A0->A3->A0
```

Next, module ***DSS Analyzer*** generates tests for all the generated DSS. We use ATVG to generate the test vectors for all the DSS in our approach. First, symbolic simulation generates the symbolic expressions of all the DSS.

Symbolic simulation

- $A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow A_1 \rightarrow A_3 \rightarrow A_0 : (n_0 \geq 0) \wedge (i_0 = 0) \wedge (x_0 = 0) \wedge (y_0 = 0) \wedge (i_0 \leq n_0) \wedge (x_1 = 5) \wedge (y_1 = y_0 + i) \wedge (i_1 = i_0 + 1) \wedge (i_1 \geq n) \wedge (out_0 = x_1 + y_1)$
- $A_0 \rightarrow A_1 \rightarrow A_3 \rightarrow A_0 : (n_0 \geq 0) \wedge (i_0 = 0) \wedge (x_0 = 0) \wedge (y_0 = 0) \wedge (i_1 \geq n) \wedge (out_0 = x_1 + y_1)$
- $A_0 \rightarrow A_3 \rightarrow A_0 : (n_0 \geq 0) \wedge (out_0 = -1)$

Next, we fed the symbolic expressions into an SMT solver to obtain the vectors. In this case, we find that the path $A_0 \rightarrow A_1 \rightarrow A_3 \rightarrow A_0$ can not be satisfied while the other two paths can. It shows that the path $A_0 \rightarrow A_1 \rightarrow A_3 \rightarrow A_0$ will not be actually executed, which means it is a false computation. This unsatisfied path will be excluded by our method. However, the VP based method in [14] does not exclude the false computation and fails to check the equivalence.

Next, our tool will automatically insert states into the transformed program and simulates the program using the obtained test vectors. After simulation, the corresponding DSS in the transformed program are generated as shown below.

DSS of transformed program

- $B_0 \rightarrow B_3 \rightarrow B_0$
- $B_0 \rightarrow B_1 \rightarrow B_2 \rightarrow B_1 \rightarrow B_3 \rightarrow B_0$

Finally, module ***SSA Converter*** generates the SSA expression for each DSS. Module ***Formula Generator*** generates the SMT formula and invokes an SMT solver to solve the formula. The formula $\wedge(I_{0i} == I_{1i})$ for all the corresponding inputs are conjuncted to the generated formula. The disjunction (such as $(O_{0i} \neq O_{1i})$) for all the corresponding outputs are conjuncted to the generated formula. The DSS-pair $A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow A_1 \rightarrow A_3 \rightarrow A_0$ and $B_0 \rightarrow B_1 \rightarrow B_2 \rightarrow B_1 \rightarrow B_3 \rightarrow B_0$ is taken as an example. The resulting conjunction is $(n_{00} \geq 0) \wedge (i_{00}= 0) \wedge (x_{00}= 0) \wedge (y_{00}= 0) \wedge (i_{00} \leq 0) \wedge (x_{01}= 5) \wedge (y_{01}=y_{00}+i_{00}) \wedge (i_{01}=i_{00}+1) \wedge (i_{01} \geq n_{00}) \wedge (out_{00}=x_{01}+y_{01}) \wedge (n_{10} \geq 0) \wedge (i_{10}= 0) \wedge (x_{10}= 0) \wedge (y_{10}= 0) \wedge$

Table 1: Experimental Results

Benchmarks	#loop	#assert	#query	VP [15]		Our Method	
				Equivalent	Time(ms)	Equivalent	Time(ms)
ASSORT	2	98	6	P	84	P	64
DIFFEQ	1	86	4	P	23	P	20
MODN	1	40	2	P	28	F	24
QRS	0	106	8	P	232	P	185
Test 1	1	45	2	F	4	P	14
Test 2	1	27	2	F	8	P	14
Test 3	2	86	2	F	16	P	26

$(i_{10} \leq n_{10}) \wedge (x_{11}=5) \wedge (y_{11}=y_{10}+i_{10}) \wedge (i_{11}=i_{10}+1) \wedge (i_{10} \geq n_{10}) \wedge (out_{10}=x_{11}+y_{11}) \wedge (n_{00}=n_{10}) \wedge (out_{00} \neq out_{10})$.
The SMT formula generated by our tool is as follows.

```
1. (declare-const n00 Int) (declare-const i00 Int)
2. (declare-const x00 Real) (declare-const y00 Real)
3. (declare-const n01 Int) (declare-const x01 Real)
. . .. . .. . .. . .. . ..
10.(assert (>= n00 0)) (assert (= i00 0))
11.(assert (= x00 0)) (assert (= y00 0))
12.(assert (<= i00 n00)) (assert (= x01 5))
13.(assert (= y01 (+ y00 i00))) (assert (= i01 (+ i00 1)))
14.(assert (>= i01 n00)) (assert (= out00 (+ x_01 y_01)))
15.(assert (>= n10 0)) (assert (= i10 0))
. . .. . .. . .. . .. . ...
20.(assert (= n00 n10))
21.(assert (not (= out00 out10)))
22.(check-sat)
```

We declared the types of the variables in steps 1-9. Statements 10-19 use ASSERT statements to represent the operations occurring in the DSS. Statement 20 is the equivalence of the input variables. Statement 21 checks whether all the output variables attain the same values or not at the end of the DSS. On feeding this input, the output generated by the SMT solver is '***unsat***', which shows the equivalence of the corresponding DSS-pair. All the corresponding DSS-pairs are proved equivalent. Therefore, the original program is equivalent to the transformed program.

5 EXPERIMENTAL RESULTS

The experiments compare the performance of our tool with the earlier equivalence checker [15]. The synthesizer used in our experiments is SPARK [16]. The experiments are conducted on a 3.3 GHz Intel Core i7 CPU and 16G RAM. Table 1 tabulates the execution time for the benchmarks required by [15] and our tool. The second column "#loops" in the table shows the number of loops in the source code. The number of assertions is given in the third column and the number of queries is given in fourth column. The verification time by the equivalence checker in [15] and the current equivalence checker ***DssEC*** are measured in milliseconds (ms). Rows 1-4 shows both tools are able to prove the equivalence in these benchmarks.

We manually created the test cases in rows 5-7. For these cases, the method in [15] fails to prove the equivalence, but our method can. In these three test cases, there are invariant operation motions which generate some false computations. The results in Table 1 show the efficiency and effectiveness of our method. For the cases without false computations our method is more efficient due to the complete path comparison. For the cases with false computations, our method can handle the cases where the existing method fails to establish the equivalence.

6 CONCLUSIONS

In this paper, we present a DSS-based equivalence checker named ***DssEC***. Our tool is more efficient and effective than the equivalence checker [15] based on VP. For cases with loop invariant motions, our tool can prove the equivalence while the tool in [15] fails. For the cases without false computation, our method can also improve the verification efficiency due to the complete path comparisons. However, our tool has a scalability limitation. Due to the large number of generated DSS, it is time-consuming to exclude and compare them one by one. It may be useful to use some heuristic in path generation (such as machine learning [17]) to obtain a relative small number of DSS.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (61902421).

REFERENCES

[1] D. D. Gajski, N. D. Dutt, A. C, Wu and S. Y. Lin (1999). High-Level Synthesis: Introduction to Chip and System Design. Kluwer Academic Publishers.
[2] S. Kundu, S. Lerner and R. K. Guptac (2010). Translation Validation of High-Level Synthesis, In TCAD, 29(4), 566-579.
[3] Tun Li, Yang Guo, Wanwei Liu, Chi yuan Ma (2013). Efficient Translation Validation of High-Level Synthesis, In ISQED, pp.516-523.
[4] Camposano, R (1991). Path-based scheduling for synthesis. In TCAD. 10(1): 85-93.
[5] Tun Li, Yang Guo, Wanwei Liu, Mingsheng Tang (2013). Translation Validation of Scheduling in High Level Synthesis, In GLSVLSI, pp.101-106.
[6] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar (2008). An equivalence-checking method for scheduling verification in high-level synthesis. In TCAD, 27:556-569.
[7] C.-H. Lee, C.-H. Shih, J.-D. Huang, and J.-Y. Jou (2011). Equivalence checking of scheduling with speculative code transformations in high-level synthesis. In ASP-DAC, 497-502.
[8] Jian Hu, Guanwu Wang, Guilin Chen, and XiangLin Wei (2019). Equivalence Checking of Scheduling in High-Level Synthesis Using Deep State Sequences. IEEE Access 7: 183435-183443.
[9] http://pypi.python.org/pypi/pycparser} (access time: June, 2021).
[10] R. Cytron, J. Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck (1991). Efficiently computing static single assignment form and the control dependence graph. In TOPLAS, 13(4):451-490.
[11] http://z3.codeplex.com/} (access time: June, 2021).

[12] Chandan Karfa, Chittaranjan A. Mandal, and Dipankar Sarkar (2012). Formal verification of code motion techniques using data-flow-driven equivalence checking. Transactions on Design Automation of Electronic Systems (TODAES): 30:1-30:37.

[13] Tun Li, Yang Guo, GongJie Liu, and Sikun Li (2005). Functional vectors generation for rt-level verilog descriptions based on path enumeration and constraint logic programming. In 8th Euromicro Conference on Digital System Design (DSD'05), 17-23.

[14] Banerjee, K., Karfa, C., Sarkar, D., Mandal, C.A. (2014). Verification of code motion techniques using value propagation. In TCAD. 33(8): 1180-1193.

[15] http://cse.iitkgp.ac.in/\%7Echitta/pubs} (access time: June, 2021).

[16] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau (2003). Spark: a high-level synthesis framework for applying parallelizing compiler transformations, In VLSID, pp.461-466.

[17] Bishop C (2008). Pattern Recognition and Machine Learning [M]. Berlin: Springer.