IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received November 24, 2019, accepted December 15, 2019, date of publication December 18, 2019, date of current version December 27, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2960511

Equivalence Checking of Scheduling in High-Level Synthesis Using Deep State Sequences

JIAN HU, GUANWU WANG, GUILIN CHEN, AND XIANGLIN WEI
The 63rd Research Institute, National University of Defense Technology, Nanjing 210000, China
Corresponding authors: Xianglin Wei (wei_xianglin@163.com), Jian Hu (hujian198681@126.com)

This work was supported by the National Natural Science Foundation of China under Grant 61902421.

ABSTRACT By using high-level synthesis tools, electronic system level design provides a promising solution to fill the growing design productivity gap of high quality hardware systems. However, an error may exist in the implementation of a compiler due to the complex and error prone compiling process. Equivalence checking is the process of proving that the target code is a correct translation of the source code being compiled. In this paper, we present a novel approach to solve the false-negative problem of value propagation (VP) based equivalence checking method. Finite State Machine with Datapath (FSMD) is used to model the original and the transformed programs. Our method proves the equivalence by comparing the deep state sequences (DSS) between the original and the transformed FSMD. Automatic test vector generation (ATVG) and simulation technique are used to recognize the corresponding DSS and exclude the false paths to solve the false-negative problem. The promising experimental results show the effectiveness of the proposed method to solve the false-negative problem in VP based equivalence checking method.

INDEX TERMS Equivalence checking, high-level synthesis, deep state sequence, FSMD.

I. INTRODUCTION

High-level synthesis (HLS) is generally a process of translating a source code into a target code, often with an objective to save critical resources and/or reduce the execution time. Thus, it can make the programmer write an efficient code and focus only on the correctness and functionality of the program being developed. HLS is seen as a solution to fill the gap of design productivity and consists of several inter-dependent subtasks such as compiler transformation, scheduling, binding and code generation [1]. Scheduling, one of subtasks in HLS, assigns operations of a behavior description with specific clock cycles based on given constraints of area, delay and data dependencies. Code motion based optimizations [2]–[5] are used in the scheduling phase of HLS tools to improve the quality of synthesis results. Most of the code motion results cannot be one-to-one mapped to their original behavior descriptions, which raises a tough verification challenge in HLS. Hence, it is necessary to validate the functional equivalence between the input program to HLS (i.e., source code) and the scheduled program generated by HLS (i.e., transformed code).

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

Several methods have made contributions to address the equivalence checking problem in high-level synthesis in recent years. Kundu *et al.* [6] presents an approach to automatically validate the target program against its initial high-level program using automated theorem proving, translation validation and relational approaches to reasoning about programs. Li *et al.* [7] defines new bisimulation relations to improve the method by reducing the number of query of automated theorem prover. But when the control structure of input behavior is modified by a path based scheduler [8], this method will fail. A dual-rail symbolic simulation of the input and output design representations of a transformation is presented in [9]. The method explores all paths of the source and target program using symbolic execution. It can automatically certify most of transformations applied by a behavioral synthesis tool. Li *et al.* [10] applied translation validation method to designs modeled using FSMD [1]. However, this method has to iterate over the loop to find a fixpoint when validating designs with loop structure, while such process does not always terminate. Karfa [11] proposed an equivalence checking method for scheduling verification. This method can solve the problem of the modified control structure of the input behaviour by the scheduler. The work reported in [14] has identified some false-negative cases of

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

the algorithm in [11] and proposed an algorithm to overcome those limitations. A formal verification method for checking correctness of code motion techniques is presented in [12]. The method can verify both uniform and nonuniform code motion techniques by identifying the properties needed to be checked during equivalence checking and validating them using the model checking tool NuSMV [13].

All these methods can not deal with loop invariant code motion and code motion across loops, since the definition of a path cover will not allow a path extended beyond a loop. A state of the art equivalence checking method based on value propagation for verification of code motion techniques was presented in [15]. Different with many other reported methods, this method can deal with uniform, non-uniform code motion and code motions across loops. This is achieved by repeated propagation of the mismatched values to subsequent paths until the final path segments are traversed without finding a match or the values match.

However, when some loop invariant operation *op* is moved before or after the loop and there is a guarantee the loop will execute, the VP based method will give false negative results. To solve the false negative problem, we propose a DSS based equivalence checking method to exclude the false paths of the programs during equivalence checking and establish the equivalence. Experimental results present the effectiveness of the proposed method.

The remainder of this paper is organized as follows. The definition of FSMD and DSS are presented in section II. The motivation of our method is given in Section III. The definition of equivalence of paths is presented in section IV. The definition of equivalence of FSMD is presented in section V. The equivalence checking algorithm is described in section VI. Section VII presents an example illustrating our method. The correctness of our method is presented in section VIII. Section IX gives the experimental results. The conclusions and future work are presented in section X.

II. FSMD AND DEEP STATE SEQUENCE

A. FINITE STATE MACHINES WITH DATA PATHS (FSMD)

In this paper, the original and the transformed programs are modeled using FSMD. An FSMD is a Finite State Machine (FSM) with datapaths of which each transition is associated with a condition over the data variables. When the transition is traversed, a set of operation will be executed to transform the variable values. We extended the definition of traditional FSMD by defining a final state of FSMD for the generation of DSS. All the incoming edges of the reset state are all altered to point to the final state. And the final state will loop to reset state. The FSMD is defined as an ordered tuple $< Q, q_0, q_f, I, O, V, f, h >$ in our method, where

1) $Q = \{q_0, q_1, q_2, \ldots, q_n\}$ is the finite set of states,
2) $q_0 \in Q$ is the reset state,
3) $q_f \in Q$ is the final state,
4) I is the finite set of primary inputs,
5) O is the finite set of primary outputs,
6) V is the finite set of storage variables,
7) $f : Q \times 2^S \rightarrow Q$ is the state transition function, where S is the set of status expression including predicates over $I \cup V$,
8) $h : Q \times 2^S \rightarrow U$ is the update function of outputs and storage variables, where U is a set of storage variable assignment or outputs statements, $U = \{x = e \mid x \in O \cup V, e$ is arithmetic expression over $I \cup V\}$

The second argument of the function f (or h) among the members of the set of status expressions is conjunction and parallel edges between two states are disjunction of status expressions. The FSMD remains deterministic, thus for any state q, $f(q, s_1) = f(q, s_2)$ and $h(q, s_1) = h(q, s_2)$ if status expression $s_1 = s_2$. The label on each transition edge in Figure 1 is of the form s/h, $s \in 2^S$, for example the label "$i \leq n/x = 5, y = y + i$" on the transition from A_1 to A_2 in Figure 1(a).

The FSMD model M_0 for the original specification shown in Figure 1 (a) is as follows,

1) $M_0 =< Q, q_0, q_f, I, O, V, f, h >$.
2) $Q = \{A_0, A_1, A_2, A_3, A_4\}$, $q_0 = A_0$, $q_f = A_4$, $V = \{x, y, i\}$, $I = \{n\}$, $O = \{out\}$.
3) $U = \{y = 0, x = 0, i = 0, x = 5, y = y + 1, i = i + 1, out = x + y, out = -1\}$.
4) $S = \{n \neq 0, i \leq n, i > n, n < 0\}$.
5) f and h as defined in the transition graph shown in Figure 1 (a).
6) Some typical expressions of f and h are shown bellow:

- $f(A_1, i \leq n) = A_2$
- $f(A_0, n < 0) = A_3$
- $h(A_1, i \leq n) = \{x = 5, y = y + 1\}$
- $h(A_0, n < 0) = \{out = -1\}$

B. DEEP STATE SEQUENCE

Definition 1: Path: A finite path p from q_i to q_j, where q_i, $q_j \in Q$, is a finite transition sequence of states of the form $\langle q_i = q_1 \overrightarrow{c_1} q_2 \overrightarrow{c_2} \ldots \overrightarrow{c_{n-1}} q_n = q_j \rangle$ such that $\forall l$, $1 \leq l \leq n-1$, $\exists \overrightarrow{c_l} \in 2^S$ such that $f(q_l, \overrightarrow{c_l}) = q_{l+1}$.

Definition 2: Deep State Sequences: For a given FSMD M, p is a finite path composed of states $\langle q_i, q_{i+1}, q_{i+2} \ldots q_n \rangle$. If the path p starts in initial state ($q_i = q_0$), ends in final state ($q_n = q_f$) and does not have any path repetition, it is called a deep state sequence.

Definition 3: False path: A path of an FSMD is called false path if it never executes (can not be activated by any input vector).

An example of deep state sequences is shown in Figure 2. The node q_1 has two branches: q_1 to q_4 and q_1 to q_2. Two DSS will be generated in this situation. At first, the branch q_1 to q_2 is selected, and when the path goes back to q_1, q_1 to q_4 is the only choice for the reason that the branch q_1 to q_2 has been visited and DSS will not have repeated paths. According to definition 2, two DSS are geneated in the figure. One is $q_0 \rightarrow q_1 \rightarrow q_4 \rightarrow q_5$ depicted in blue dotted line and the

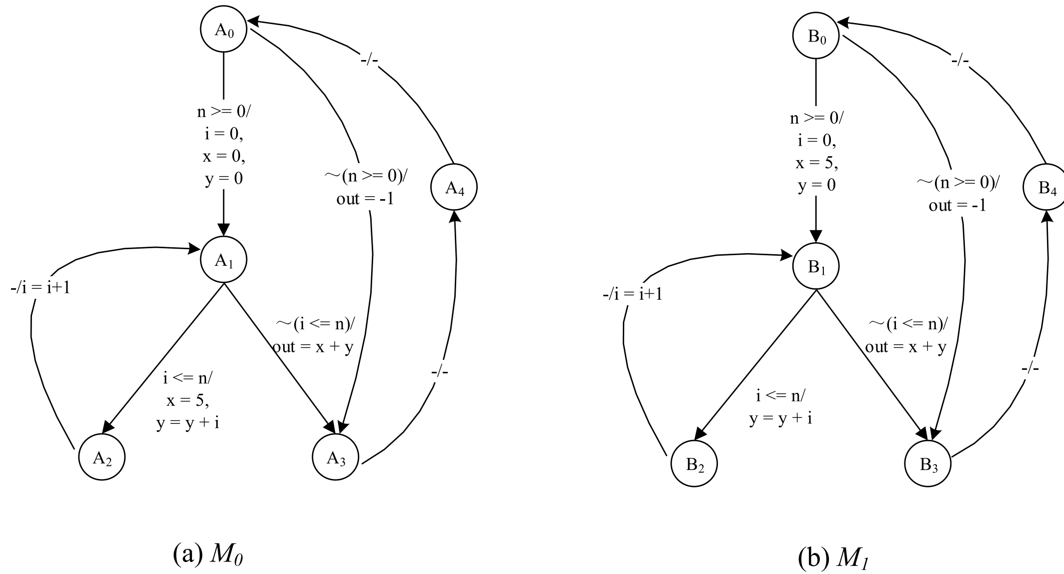


FIGURE 1. (a) M_0, the original FSMD (b) M_1, the transformed FSMD.

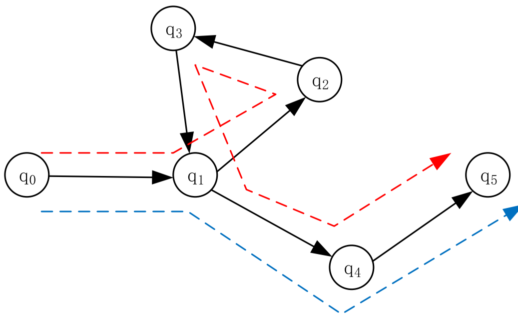


FIGURE 2. Example of deep state sequences [16].

other is $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_1 \rightarrow q_4 \rightarrow q_5$ depicted in red dotted line.

Since the path in the existing path-based method and the DSS in our method are both generated in a syntactic manner. Some paths or DSS will not be executed in practice because no inputs can satisfy the path condition, which will cause false negative results. To avoid these false paths results, our method excludes the DSS that can not be satisfied by the inputs using ATVG technique.

III. MOTIVATIONS

Loop invariant statements consists of statements or expressions inside a loop body which are not dependent on loop iterations. In other words, these statements produce the same result each time the loop is executed. When this code are moved outside the loop body, they will not change the program semantics. By reducing the number of times loop invariant expressions executed, loop invariant code motion can improve overall program execution time by a factor equal to the loop size.

Let us consider the original and the transformed FSMD in Figure 1. In this example, the operation $x = 5$ is a loop invariant statement for FSMD M_0 in Figure 1(a). It is moved out of the loop in the transformed FSMD M_1 in Figure 1(b). There are 3 possible paths in this FSMD, $p_1 = A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i>n} A_3 \rightarrow A_4$, $p_2 = A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i\leq n} A_2 \rightarrow A_1 \xrightarrow{i>n} A_3 \rightarrow A_4$, $p_3 = A_0 \xrightarrow{n<0} A_3 \rightarrow A_4$. The path p_1 executes if the loop condition $n \geq 0$ and $i > n$ are satisfied. The path p_2 executes if the loop condition $n \geq 0$ and $i \leq n$ are satisfied. The path p_3 executes for the condition $n < 0$. In this example, n is always greater than or equal to 0 and i is equal to 0 when the state A_1 is reached for the first time. Therefore, the loop will execute at least once for all possible $n \geq 0$ and $i = 0$. This means the path p_1 is a false path which will never execute.

For equivalence checking of these two programs using the VP based method, it will try to prove all possible paths in the FSMDs equivalent. Thus, the equivalence checker will try to find the equivalence of paths p_1, p_2 and p_3 in the other FSMD M_1. It finds that the paths p_2 and p_3 of FSMD M_0 are equivalent to the paths $q_2 = B_0 \xrightarrow{n\geq 0} B_1 \xrightarrow{i\leq n} B_2 \rightarrow B_1 \xrightarrow{i>n} B_3 \rightarrow B_4$ and $q_3 = B_0 \xrightarrow{n<0} B_3 \rightarrow B_4$ of FSMD M_1, respectively. However the equivalence checking method finds that the path p_1 of FSMD M_0 is not equivalent to the path $q_1 = B_0 \xrightarrow{n\geq 0} B_1 \xrightarrow{i>n} B_3 \rightarrow B_4$ of FSMD M_1, since the final value of the variable x is different. It may be found that the final values of x are 0 and 5 after the execution of path p_1 in M_0 and path $q_1 = B_0 \xrightarrow{n\geq 0} B_1 \xrightarrow{i>n} B_3 \rightarrow B_4$ in M_1. In this example, as described above, the path p_1 will never execute. This equivalence checking method will report non-equivalence of the FSMDs due to this false path. If we can recognize this false path before validation and exclude it during equivalence checking, we can prove the

equivalence between these two behaviors. The preliminary results and the observations motivate us to propose the novel approach.

IV. EQUIVALENCE OF PATHS

Definition 4: Condition of execution of the path (R_α): the condition of the path α is a logical expression over the inputs in I and the variables in V such that R_α is satisfied by the initial data state of the path iff the path α is traversed. Thus, R_α is the weakest precondition of the path α [18].

Definition 5: Data transformation of path (r_α): the data transformation of the path α over V, denoted as r_α, is the tuple $\langle s_\alpha, O_\alpha \rangle$; the first member s_α, termed as storage (variable) transformation of α, is an ordered tuple $\langle e_i \rangle$ of algebraic expressions over the inputs in I and the variables in V such that the expression e_i represents the value of the variable v_i after the execution of the path in terms of the initial data state of the path; the second member $O_\alpha = [OUT(P_{i_1}, e_1), OUT(P_{i_2}, e_2), \ldots]$ represents the output list along the path α [12].

The characteristic formula $\tau_\alpha(\bar{v}, \bar{v}_f, O)$ of the path α is $R_\alpha(\bar{v}) \wedge (\bar{v}_f = s_\alpha(\bar{v})) \wedge (O = O_\alpha(\bar{v}))$, where O_α is the output list in the path α, s_α is the data transformation, $\bar{v}_f$ represents a vector of variables of V and $\bar{v}$ represents a vector of variables of $I \cup V$. The formula captures the following: If the condition of execution R_α of the path α is satisfied by the (initial) vector $\bar{v}$ at the beginning of the path, then the path is executed, and after execution, the output $O_\alpha(\bar{v})$ is produced [12], and the final vector $\bar{v}_f$ of variable values becomes $s_\alpha(\bar{v})$.

Definition 6: Equivalence of paths. A path p_1 of an FSMD M_0 is equivalent to a path p_2 of another FSMD M_1 if $R_{p_1} \equiv R_{p_2}$ and $O_{p_1} \equiv O_{p_2}$, where R_{p_1} and R_{p_2} represent the conditions of execution of p_1 and p_2, respectively and O_{p_1} and O_{p_2} are the output lists of p_1 and p_2, respectively. The equivalence of paths p_1 and p_2 is denoted as $p_1 \equiv p_2$.

V. EQUIVALENCE OF FSMD

Let FSMD $M_0 = \langle Q_0, q_{00}, q_{0f}, I_0, O_0, V_0, f_0, h_0 \rangle$ and $M_1 = \langle Q_1, q_{10}, q_{1f}, I_1, O_1, V_1, f_1, h_1 \rangle$ represent the behavior of the original and the transformed programs, respectively. The main goal of our paper is to validate the equivalence of M_0 and M_1, which means that the execution traces of M_0 and M_1 produce the same outputs on each output port for all possible inputs. A DSS of an FSMD is a finite path from the reset state to final state without having repeated paths. So, a DSS represents one possible execution of an FSMD under an input sequence and produces an output sequence. We define the equivalence of FSMD as follows:

Definition 7: Equivalence of DSS. A DSS c_1 of an FSMD M_0 is equivalent to a DSS c_2 of another FSMD M_1 if $R_{c_1} \equiv R_{c_2}$ and $O_{c_1} \equiv O_{c_2}$, where R_{c_1} and R_{c_2} represent the conditions of execution of c_1 and c_2, and O_{c_1} and O_{c_2} represent the output lists of c_1 and c_2, respectively. The equivalence of DSS c_1 and DSS c_2 is denoted as $c_1 \equiv c_2$.

Definition 8: Containment of FSMDs: If for any DSS c_0 of FSMD M_0, there exists a DSS c_1 of FSMD M_1 such that $c_0 \equiv c_1$, the FSMD M_0 is said to be contained in an FSMD M_1, symbolically $M_0 \sqsubseteq M_1$.

Definition 9: Equivalence of FSMDs: If two FSMDs M_0 and M_1 such that $M_0 \sqsubseteq M_1$ and $M_0 \sqsupseteq M_1$, then M_0 and M_1 are said to be equivalent, symbolically $M_0 \equiv M_1$.

Algorithm 1 EquivalenceChecker(FSMD M_0, FSMD M_1)

1: $F_0 \leftarrow$ *Ex_FSMD*(M_0)
2: $DSS_0 \leftarrow$ *Ex_DSS*(F_0)
3: *Ins_State*(M_1)
4: **while** $DSS_0 \neq \emptyset$ **do**
5: $dss_0 \leftarrow$ *Sel_DSS*(DSS_0)
6: $i_0 \leftarrow$ *Gen_test*(dss_0)
7: $dss_1 \leftarrow$ *Sim_Model*(M_1, i_0)
8: $r \leftarrow$ *Comp_DSS*(dss_0, dss_1)
9: **if** $r ==$ *'sat'* **then**
10: *Error and Exit('Not Equivalent')*
11: **else**
12: *Remove_DSS*(dss_0, DSS_0)
13: **end if**
14: **end while**
15: **if** M_0 *and* M_1 *have not interchanged* **then**
16: *Goto step* 1 *with* M_0 *and* M_1 *interchanged*
17: **else**
18: *Exit('Equivalent')*
19: **end if**

VI. EQUIVALENCE CHECKING ALGORITHM

Algorithm 1 presents our proposed equivalence checking algorithm. Two programs M_0 and M_1 are taken as the inputs representing the original and the transformed programs. The algorithm recognizes the corresponding DSS-pairs and excludes the false paths during verification. And then it checks the equivalence of all the generated DSS-pairs of the two programs. Supposing the scheduling will not change the names of storage variables, inputs and outputs. The following subsections describes the details of the algorithm.

A. GENERATING FSMD FROM THE ORIGINAL PROGRAM

The procedure **Ex_FSMD** can automatically generate the FSMD from the original program before scheduling. We use the tool Pycparser [20] to generate an abstract syntax tree (AST) of the original program. The procedure traverses through the AST and creates FSMD nodes based on the type of syntax tree node. The combination of the following three basic constructs can represent any sequential behavior:

1) Basic Blocks (BBs): sequences of statements without any bifurcation of control flow
2) Control Blocks (CBs): *if-else* constructs or *switch-case* constructs
3) Loop Blocks (LBs): *while*, *do while* or *for* constructs

Therefore, our algorithm can effectively represent any sequential behavior as an FSMD if it captures these three

constructs in a program. We construct FSMD in the following way:

1) For start node: Before the first statement of the program, a node is created as the start node.
2) For BBs: Create a node before a CB or LB.
3) For CBs: *if(c) then* BB_1 *else* BB_2 *endif*. In this branch case, the FSMDs M_1 and M_2 of BB_1 and BB_2 are constructed first. Second, the start states of the two FSMDs are merged into one start state and the end states of the two FSMDs are merged into one end state. Third, we placed the condition c as the condition of the transition from the start node to the FSMD M_1 corresponding to BB_1 and placed the condition $\neg c$ as the condition of the transition from the start node to the FSMD M_2 corresponding to BB_2.
4) For LBs: *while(c) BB endwhile*. In this case, the FSMD M of BB is constructed first. Second, the start and the end state of M are merged into one state, q_0 say, and the condition c is placed on the transition from q_0 to the FSMD M corresponding to BB. A transition from q_0 with condition $\neg c$ is added as the exit path from the loop in the FSMD.
5) For end node: Before the return statement of the program, a node is created as the end node.

Taking Figure 1(a) as an example, node A_0 is created as the start node before the first statement of the program, A_4 is created as the end node before the return statement. And nodes A_1, A_2 and A_3 are created as the LB construct, the conditioin $i \leq n$ is placed on the transition from state A_1 to state A_2 and the conditioin $i > n$ is placed on the transition from state A_1 to state A_3. The FSMD construction algorithm in our method is more efficient than the method in [12]. Because we only create one state for a BB, while the method in [12] may create several states for a BB according to a dependence graph that is a directed acyclic graph. The condition associated with each transition between the states of the BB is "true", hence, we do not need to create so many states and one state is enough.

B. GENERATING DSS FROM THE FSMD

Procedure *Ex_DSS* can automatically generate all the DSS from the FSMD after the FSMD of the design is obtained. To deal with the FSMD, the extracted FSMD is first converted to a link list structure. The list structure of the Figure 1 is shown in Figure 3. Each node in the link list represents a state in the FSMD and the number of the node is the state number of the state in FSMD. If there is a transition from state a to state b, there is a link from node a to node b in the link list.

Then, the link list is traversed from the start node in a depth-first search manner, and all the state sequences without repeated paths from the reset state to the final state are generated. The algorithm for DSS extraction is presented in Algorithm 2. We use recursive method to generate the DSS.

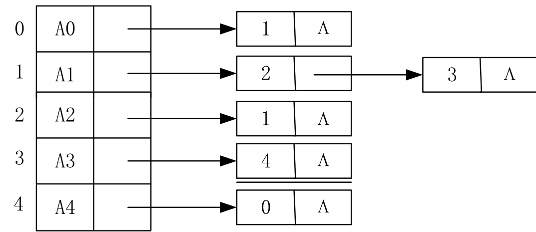


FIGURE 3. The converted list of FSMD in Figure 1.

Algorithm 2 DSSGenerator(seq, num, $list$, p, j)

```
1:  Seq[num][j] ← p.vertex
2:  if InSeq(p.vertex, Seq[num], j − 1) then
3:     p_n ← list[p.vertex].next
4:     while p_n ≠ Null do
5:        if InSeq(p.vertex, Seq[num], j − 1) then
6:           p_n ← 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 ← list[p.vertex].next
13:    if p_n ≠ Null then
14:       Generate_Seq(p_n, Seq[num], list, j + 1)
15:    end if
16: end if
17: p ← p.next
18: while p ≠ Null do
19:    if InSeq(p.vertex, Seq[num], j − 1) then
20:       p ← p.next
21:    else
22:       num ← num + 1
23:       Generate_Seq(p, Seq[num], list, j)
24:    end if
25: end while
26: Return Seq
```

During the recursion process, the algorithm will determine whether the path is repeated. The unrepeated paths are automatically saved in a list. The variable *Seq*, *num*, *list*, *p*, *j* are used to save the generated DSS, the number of generated DSS, the converted FSMD list, the current FSMD node and the number of node in current DSS. The function *InSeq()* determines if node *p* is in the first *j-1* nodes of *Seq*, which can determine whether the path is repeated. When a node is visited, the algorithm will visit the subsequent unvisited nodes in a depth-first search manner and add them to the DSS list. For example, in Figure 1 (a) if A_1 is encountered, A_1 is appended to A_0. The algorithm will determine whether the subsequent node A_2, A_3 of A_1 have been visited. And then it will visit the unvisited node. The process will continue in a depth first search manner until it reaches the final state. For example, the path $A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i\leq n} A_2 \rightarrow A_1 \xrightarrow{i>n} A_3 \rightarrow A_4$ is a generated DSS from the FSMD.

C. GENERATING CORRESPONDING DSS IN THE TRANSFORMED PROGRAM

1) TEST GENERATION

Procedure **Sel_DSS** selects a dss_0 from the DSS set if the obtained DSS set is not empty. Then the procedure **Gen_test** generates direct test for the selected dss_0 by using ATVG technique [21]. The procedure first simulates the selected dss_0 symbolically and then feeds the generated expression to a SMT solver. The SMT solver will output satisfied input vector for the dss_0 or output unsatisfied which means there is no inputs that can satisfy the DSS. The ATVG procedure can automatically identify the false path by checking whether the path will actually execute. If there is no input vector that can satisfy the path execution, our method will recognize the path as a false path and ignore it during equivalence checking.

2) CORRESPONDING DSS GENERATION

The corresponding potential equivalent DSS are generated in our proposed algorithm by using simulation technique. The output statements are inserted automatically into the source code of transformed program by traversing the generated AST. Figure 4 shows the inserted output statements for the program in Figure 1 (such as "//output A0" or "//output B0").

(a) Original Program	(b) Transformed Program

//Output A0
if(n≥0)
{
x=0;
y=0;
//Output A1
for(i=0;i≤n;i++)
{
y=y+i;
x=5;
//Output A2
}
out=x+y;
}
else
out = -1;
//Output A3
//Output A4
	//Output B0
	if(n≥0)
	{
	x=5;
	y=0;
	//Output B1
	for(i=0;i≤n;i++)
	{
	y=y+i;
	//Output B2
	}
	out=x+y;
	}
	else
	out = -1;
	//Output B3
	//Output B4

FIGURE 4. The inserted output statements in the code of Figure 1.

The generated tests in the previous step (Test Generation) are applied to simulate the transformed program by procedure **Sim_Model**. When the simulation ends, the procedure will output corresponding deep state sequence according to the inserted output statements. The two *dss* from the original program and the transformed program under the same test is a corresponding potential equivalent DSS-pair. Taking $dss_0 = A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i\leq n} A_2 \rightarrow A_1 \xrightarrow{i>n} A_3 \rightarrow A_4$ in M_0 and the generated $dss_1 = B_0 \xrightarrow{n\geq 0} B_1 \xrightarrow{i\leq n} B_2 \rightarrow B_1 \xrightarrow{i>n} B_3 \rightarrow B_4$ in M_1 in Figure 1 (a) as an example, dss_0 and dss_1 constitute one corresponding potential equivalent DSS-pair, where dss_1 is obtained in the transformed program by using the test generated from dss_0.

D. COMPARING THE CORRESPONDING POTENTIAL EQUIVALENT DSS-PAIRS

Procedure **Comp_DSS** will compare the DSS-pairs using word-level symbolic simulation technique after recognizing the corresponding potential equivalent DSS-pairs. Before symbolic simulation, the expression of each statements should be transformed to static single assignment (SSA) [17]. And then symbolic simulation conjuncts the expression of SSA of the DSS-pairs. Next, we convert the conjunction SSA expression to SMT format. Finally, an SMT solver is used to check the equivalence of the SMT format of the corresponding DSS-pairs. The corresponding DSS-pair is not equivalent when the SMT solver returns "sat". Our algorithm exits and outputs "Not Equivalent". The corresponding DSS-pair is equivalent when the SMT solver returns "unsat". We remove the dss_0 from DSS set and continue the checking process until the DSS set is empty. If all the DSS-pairs are equivalent, it shows the FSMD M_0 is contained in FSMD M_1. Next, we exchange the two FSMDs and repeat the equivalence checking process from the first step. If FSMD M_1 is also contained in FSMD M_0, it shows the two FSMDs are equivalent according to definition 9.

VII. RUNNING EXAMPLE

The working of our algorithm for loop invariant code motion is briefly discussed with the example shown in Figure 1. The scheduling will not change the names of storage variables, inputs and outputs. The iterations of the equivalence checking method are as follows.

First, procedure **Ex_FSMD** extracts the FSMD from the original program as shown below. A_i represents the state name and the state transition condition e_1 and update function e_2 in the form e_1/e_2 are separated by '/'. Each assignment statement is separated by a comma. '−' represents *true* for transition condition in e_1 or no operations in update function e_2.

FSMD of original program:

$A_0 : n \geq 0/y = 0, i = 0, x = 0, A_1; !(n \geq 0)/out = -1, A_3;$
$A_1 : i \leq n/y = y + i, x = 5, A_2; !(i \leq n)/out = x + y, A_3;$
$A_2 : -/i = i + 1, A_1;$
$A_3 : -/-, A_4;$
$A_4 : -/-, A_0;$

Next, the FSMD is converted to a link list as shown in Figure 3. And procedure **Ex_DSS** generates all the DSS from the FSMD as shown bellow. The procedure generates three DSS from the generated FSMD. "A_i" represents the state name. '→' represents the transition between states and the expression on the transition represents transition condition, such as '$n \geq 0$' and '$i > n$'.

DSS of original program:

1. $A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i\leq n} A_2 \rightarrow A_1 \xrightarrow{i>n} A_3 \rightarrow A_4$
2. $A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i>0} A_3 \rightarrow A_4$
3. $A_0 \xrightarrow{n<0} A_3 \rightarrow A_4$

Next, procedure **Gen_test** generates input vectors for all the generated DSS by using ATVG technique.

First, the expression of each statement is transformed to SSA expression. The subscript of a variable is initialized to 0 and it should increase by 1 when the variable is assigned a value (such as $i_1 = i_0 + 1$). Second, symbolic simulation technique is used to generate the symbolic expression of each DSS. Symbolic expression is the conjunctions of the conditions and operations of the program through each DSS. The generated symbolic expressions are shown below.

Symbolic simulation

1. $A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i\leq n} A_2 \rightarrow A_1 \xrightarrow{i>n} A_3 \rightarrow A_4 : (n_0 \geq 0) \wedge (i_0 = 0) \wedge (x_0 = 0) \wedge (y_0 = 0) \wedge (i_0 \leq n_0) \wedge (x_1 = 5) \wedge (y_1 = y_0 + i) \wedge (i_1 = i_0 + 1) \wedge (i_1 \geq n) \wedge (out_0 = x_1 + y_1)$
2. $A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i>0} A_3 \rightarrow A_4 : (n_0 \geq 0) \wedge (i_0 = 0) \wedge (x_0 = 0) \wedge (y_0 = 0) \wedge (i_0 \geq n) \wedge (out_0 = x_0 + y_0)$
3. $A_0 \xrightarrow{n<0} A_3 \rightarrow A_4 : (n_0 \geq 0) \wedge (out_0 = -1)$

The obtained symbolic expressions are fed into an SMT solver to generate the directed tests. In this case, we find that the path $A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i>0} A_3 \rightarrow A_4$ can not be satisfied by any input vector while the other two paths can. This shows that the path $A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i>0} A_3 \rightarrow A_4$ will not be executed in practice, which means it is a false path. Therefore, our method excludes this unsatisfied path and ignores it during equivalence checking. But the VP based method in [15] can not recognize and exclude the false path and it will fail to establish the equivalence.

Next, output statements are inserted into the transformed program by procedure **Ins_State** as shown in Figure 4, such as "//output A0" or "//output B0". And then the generated tests are applied to simulate the program by procedure **Sim_Model**. After simulation, the corresponding potential equivalent DSS are obtained in the transformed program as shown below.

DSS of transformed program:

1. $B_0 \xrightarrow{n\geq 0} B_1 \xrightarrow{i\leq n} B_2 \rightarrow B_1 \xrightarrow{i>n} B_3 \rightarrow B_4$
2. $B_0 \xrightarrow{n<0} B_3 \rightarrow B_4$

Finally, procedure **Com_DSS** verifies the corresponding potential equivalent DSS-pairs between the original and transformed program by using symbolic simulation and a SMT solver. First, the expression of each statement is transformed to SSA expression. Second, the symbolic expressions of the DSS-pair are generated and the two symbolic expressions are conjuncted. Third, for all the corresponding inputs, we conjunct the formula $\wedge(I_{0i} == I_{1i})$ to the SSA expression. Finally, for all the corresponding outputs, we conjunct the disjunction of all such formulas (such as $(O_{0j} \neq O_{1j})$) to the SSA expression. We take the DSS-pair $A_0 \xrightarrow{n\geq 0} A_1 \xrightarrow{i\leq n} A_2 \rightarrow A_1 \xrightarrow{i>n} A_3 \rightarrow A_4$ and $B_0 \xrightarrow{n\geq 0} B_1 \xrightarrow{i\leq n} B_2 \rightarrow B_1 \xrightarrow{i>n} B_3 \rightarrow B_4$ as an example. The resulting symbolic expression is $(n_{00} \geq 0) \wedge (i_{00} = 0) \wedge (x_{00} = 0) \wedge (y_{00} = 0) \wedge (i_{00} \leq n_{00}) \wedge (x_{01} = 5) \wedge (y_{01} = y_{00} + i) \wedge (i_{01} = i_{00} + 1) \wedge (i_{01} \geq n_{00}) \wedge (out_{00} = x_{01} + y_{01}) \wedge (n_{10} \geq 0) \wedge (i_{10} = 0) \wedge (x_{10} = 0) \wedge (y_{10} = 0) \wedge (i_{10} \leq n_{10}) \wedge (x_{11} = 5) \wedge (y_{11} = y_{10} + i) \wedge (i_{11} = i_{10} + 1) \wedge (i_{10} \geq n_{10}) \wedge (out_{10} = x_{11} + y_{11}) \wedge (n_{00} == n_{10}) \wedge (out_{00} \neq out_{10})$. The SMT solver outputs 'unsat' for the expression, which means the corresponding DSS-pair is equivalent.

In this example, the two valid corresponding potential equivalent DSS-pairs are proved to be equivalent. According to definition 8, the original program is contained in the transformed program, denoted as $M_0 \sqsubseteq M_1$. Next, we exchange the two FSMDs and repeat the verification process. And we can prove that the transformed program is also contained in the original program, denoted as $M_0 \sqsupseteq M_1$. Hence, the two FSMDs are equivalent according to definition 9, denoted as $M_0 \equiv M_1$.

VIII. CORRECTNESS

Theorem 1: The generated DSS in the transformed program under the test generated from the DSS in the original program is the only corresponding potential equivalent DSS in the transformed program.

Proof: Assuming the DSS set of the original program is DSS_0, DSS set of the transformed program is DSS_1 and the tests set I_0 are generated from DSS_0 using ATVG technique.

Since the FSMD in our method is deterministic, one input test activates only one DSS. $\forall i_0 \in I_0$ generated from a $dss_0 \in DSS_0$, it can only activate one corresponding $dss_1 \in DSS_1$. Other $dss_1 \in DSS_1$ cannot recognize the input test. Meanwhile the equivalent DSS-pair must recognize the same test. Therefore, the DSS generated by simulation from the transformed program is the only corresponding potential equivalent DSS needed to be considered. □

Theorem 2: When our algorithm terminates, the equivalence between the original and transformed programs can be checked.

Proof: Assuming the DSS set of the original program is DSS_0, DSS set of the transformed program is DSS_1, I_0 is the test set generated from DSS_0 using ATVG technique, original program is M_0 and transformed program is M_1. $\forall dss_0 \in DSS_0, i_0 \in I_0$ is the generated test from dss_0. dss_1 is the obtained DSS from transformed program using test i_0.

If dss_0 in DSS_0 can not be satisfied by any input vector using ATVG technique, it means dss_0 is a false path. We exclude it from the set DSS_0. The false path problem will be solved by ignoring the unsatisfied dss_0 during the equivalence checking process.

If dss_0 can be satisfied and $dss_0 \neq dss_1$(not equivalent), then we have found a test i_0 which makes the outputs of the M_0 and M_1 different. It means the original and transformed programs are not equivalent, symbolically $M_0 \neq M_1$. Our algorithm terminates with the output 'not equivalent'.

If dss_0 can be satisfied and $dss_0 \equiv dss_1$, then a dss_1 in M_1 equivalent with the dss_0 in M_0 has been found. The verified dss_0 is removed from DSS_0 and another $dss_0 \in DSS_0$ is selected. The process will repeat until DSS_0 is empty. When all the dss_0 in the original program have found an equivalent dss_1 in transformed program, according to definition 8, the original program is contained in the transformed

TABLE 1. Experimental results.

Benchmarks	#States	#Loop	VP [15]		Our Method	
			Equivalent	Time(ms)	Equivalent	Time(ms)
ASSORT	10	2	Yes	84	Yes	86
DIFFEQ	6	1	Yes	24	Yes	28
MODN	6	1	Yes	28	Yes	32
PERFECT	9	1	Yes	20	Yes	20
QRS	16	0	Yes	232	Yes	185
ASSORT-1	10	2	No	32	No	54
DIFFEQ-1	6	1	No	100	No	160
MODN-1	6	1	No	40	No	62
PERFECT-1	9	1	No	32	No	40
QRS-1	16	0	No	220	No	180

TABLE 2. Experimental results.

Benchmarks	#States	#Loop	VP [15]		Our Method	
			Equivalent	Time(ms)	Equivalent	Time(ms)
Test 1	5	1	No	8	Yes	14
Test 2	8	1	No	10	Yes	14
Test 3	12	2	No	16	Yes	26
Test 4	6	1	No	10	Yes	14
Test 5	10	2	No	20	Yes	26

program, denoted as $M_0 \sqsubseteq M_1$. Next, we exchange the two FSMDs and repeat the verification process. If we can prove that the transformed program is also contained in the original program, denoted as $M_0 \sqsupseteq M_1$, the two FSMDs are equivalent according to definition 9, denoted as $M_0 \equiv M_1$. Our algorithm terminates with the output 'equivalent'. □

IX. EXPERIMENTAL RESULTS

The equivalence checking algorithm described in this paper has been implemented in python and C and has been run on a 2.5 GHz Intel i5 duo processor with 8G RAM. The synthesizer used in our experiments is SPARK [19]. The SPARK scheduler is allowed to perform uniform, non-uniform and code motion across loop. The symbolic simulator and FSMD extractor are implemented based on Pycparser [20]. The DSS extractor is implemented in C and Z3 [22] is used as our SMT solver. Tables 1 and 2 show the results of the experiments. Table 1 tabulates the comparison of the execution time required by the VP based equivalence checking method [15] and our DSS-based method for the benchmarks. The VP-based tool is available at [23]. The first column is the names of the benchmarks. The second and the third columns show the number of states in the original FSMD and loops in the original program. For each benchmark, the obtained runtime (in milliseconds (ms)) and equivalence result by both tools are recorded. The equivalent scenarios are listed in rows 1-5 and the inequivalent scenarios with some manually introduced faults for the benchmarks listed in rows 1-5 are listed in rows 6-10. Both tools are able to establish the equivalence for the benchmarks in rows 1-5 and report non-equivalence for the benchmarks in rows 6-10.

Some test cases where the VP based method fails to establish the equivalence are presented in Table 2. We created these test cases manually. In these test cases, some loop invariant operations are moved before or after the loop from inside it and there is a guarantee the loop will execute at least once, which makes some paths become false paths in the FSMD. Table 2 shows our DSS-based method can establish the equivalence for the cases with false paths, but the VP based method fails. The results in Tables 1 and 2 show that 1) first our method can handle all the cases which can be handled by the VP based equivalence checking method; 2) second our method can solve the false negative problems in the method [15].

X. CONCLUSION

An equivalence checking method presented in this paper verifies loop invariant code transformations with false paths. For each DSS, our method automatically symbolically simulates it and uses a SMT solver Z3 to validate the satisfaction of the DSS. If the DSS can not be satisfied, our method ignores the false path during equivalence checking. Using our proposed method, we can solve the false-negative problem in VP based method.

A limitation of the present work is the scalability of our method. Because the DSS is generated in a syntactic way, our algorithm will generate all the possible DSS in the FSMD. If a program has too many branches, the number of the DSS will be large. It will be time-consuming to generate all the tests for a large number of DSS and compare the DSS-pairs. Enhancing the equivalence checking to encompass the limitation seems to be a promising future endeavor. Investigating some heuristic path extension method (such as machina learning [24]) to generate less number of DSS can be useful in this regard.

REFERENCES

[1] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Norwell, MA, USA: Kluwer, 1999.

[2] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Using global code motions to improve the quality of results for high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 2, pp. 302–312, Feb. 2004.

[3] L. C. V. Dos Santos and J. Jress, "A reordering technique for efficient code motion," in *Proc. 36th ACM/IEEE Design Automat. Conf. (DAC)*. New York, NY, USA: ACM, 1999, pp. 296–299.

[4] G. Lakshminarayana, A. Raghunatharn, and N. Jha, "Incorporating speculative execution into scheduling of control-flow-intensive designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 3, pp. 308–324, Mar. 2000.

[5] L. C. V. Dos Santos, M. J. M. Heijligers, C. A. J. Van Eijkvan, J. Van Eijnhoven, and J. A. G. Jess, "A code-motion pruning technique for global scheduling," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 1, pp. 1–38, 2000.

[6] S. Kundu, S. Lerner, and R. K. Gupta, "Translation validation of high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 4, pp. 566–579, Apr. 2010.

[7] T. Li, Y. Guo, and W. Liu, "Efficient translation validation of high-level synthesis," in *Proc. IEEE Int. Symp. Qual. Electron. Design*, Mar. 2013, pp. 516–523.

[8] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 10, no. 1, pp. 85–93, Jan. 1991.

[9] Z. Yang, K. Hao, K. Cong, S. Ray, and F. Xie, "Equivalence checking for compiler transformations in behavioral synthesis," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Asheville, NC, USA, Oct. 2013, pp. 491–494.

[10] T. Li, Y. Guo, W. Liu, and M. Tang, "Translation validation of scheduling in high level synthesis," in *Proc. 23rd ACM Int. Conf. Great Lakes Symp. VLSI (GLSVLSI)*, 2013, pp. 101–106.

[11] C. Karfa, D. Sarkar, P. Kumar, and C. Mandal, "An equivalence-checking method for scheduling verification in high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 3, pp. 556–569, Mar. 2008.

[12] C. Karfa, C. Mandal, and D. Sarkar, "Formal verification of code motion techniques using data-flow-driven equivalence checking," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 17, no. 3, Jul. 2012, Art. no. 30.

[13] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model checker," *Int. J. Softw. Tools Technol. Transf.*, vol. 2, no. 4, pp. 410–425.

[14] C. H. Lee, C. H. Shih, and J. D. Huang, "Equivalence checking of scheduling with speculative code transformations in high-level synthesis," in *Proc. 16th Asia–South Pacific Design Autom. Conf.*, 2011, pp. 497–502.

[15] K. Banerjee, C. Karfa, and D. Sarkar, "Verification of code motion techniques using value propagation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 33, no. 8, pp. 1180–1193, Aug. 2014.

[16] J. Hu, T. Li, and S. Li, "Formal equivalence checking between SLM and RTL descriptions," in *Proc. 28th IEEE Int. Syst. Chip Conf. (SOCC)*, Beijing, China, Sep. 2015, pp. 131–136.

[17] R. Cytron, J. Ferrante, K. B. Rosen, N. M. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[18] Z. Manna, *Mathematical Theory of Computation*. New York, NY, USA: McGrawHill, 1974.

[19] S. Gupta, N. Dutt, and R. Gupta, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proc. Int. Conf. VLSI Design, IEEE Comput. Soc.*, Jan. 2003, pp. 461–466, 2003.

[20] Accessed: Jun. 2019. [Online]. Available: http://pypi.python.org/pypi/pycparser

[21] T. Li, Y. Guo, G. Liu, and S. Li, "Functional vectors generation for RT-level verilog descriptions based on path enumeration and constraint logic programming," in *Proc. 8th Euromicro Conf. Digit. Syst. Design (DSD)*, 2005, pp. 17–23.

[22] Accessed: Jun. 2019. [Online]. Available: http://z3.codeplex.com/

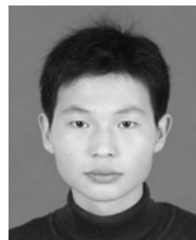
[23] Accessed: Jun. 2019. [Online]. Available: http://cse.iitkgp.ac.in/%7Echitta/pubs

[24] C. Bishop, *Pattern Recognition and Machine Learning*. Berlin, Germany: Springer, 2008.

JIAN HU received the B.S., M.S., and Ph.D. degrees in computer science from the College of Computer, National University of Defense Technology, in 2009, 2012, and 2016, respectively. He is currently an Assistant Professor with The 63rd Research Institute, National University of Defense Technology. He has published more than 20 articles in electronic design automation and computer aided design fields, which are accepted by FCS, JCSC, DAC, SOCC, ISQED, and GLSVLSI. His research interests include computer aided design, formal verification, and high-level equivalence checking.

GUANWU WANG received the B.S. degree in computer science from the College of Computer, Sichuan University, in 2008, and the M.S. and Ph.D. degrees in computer science from the College of Computer, National University of Defense Technology, in 2011 and 2015, respectively. He is currently an Assistant Professor with The 63rd Research Institute, National University of Defense Technology. He has published more than ten scientific articles in computer architecture and compiler, which are indexed by SCI and EI. His research interests include coarse grained reconfigurable architecture, high-level compiler, and microelectronics.

GUILIN CHEN received the M.S. degree in computer science from the College of Computer, National University of Defense Technology, in 2018. He is currently an Assistant Engineer with The 63rd Research Institute, National University of Defense Technology. He has published several articles in CNN accelerations and microarchitecture, which are indexed by SCI and EI. His research interests include microarchitecture, adversarial machine learning, and neural network accelerator.



XIANGLIN WEI received the bachelor's degree from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2007, and the Ph.D. degree from the PLA University of Science and Technology, Nanjing, China, in 2012. He is currently working as a Researcher with The 63rd Research Institute, National University of Defense Technology, Nanjing. His research interests include mobile edge computing, wireless network optimization, and the Internet of Things. He has served as an Editorial Member of many international journals and a TPC member of a number of international conferences. He has also organized a few special issues for many reputed journals.