Test Input Data Generation for Choiceless Program Nets

Biao Wu[1] and **Qi-Wei Ge**[2]

ABSTRACT

Software testing is an important problem in designing a large software system and this problem is difficult to solve due to its computational complexity. Generating test input data is an effective way to approach this problem, and we try to use program net to find test input data. In our previous work, as the first step towards solving software testing problem, we have proposed algorithms to divide a whole program net into subnets that can structurally cover the original one based on a divide-and-conquer method. This paper aims to solve the remaining task of our approach, which is how to find test input data for each subnet to be called choiceless program net. First, definitions of program nets are extended and the properties of choiceless program nets are analysed. Then, polynomial algorithms are proposed to get all constraint conditions of any given choiceless program nets. Finally, a method to generate test input data under the obtained constraint conditions is proposed by adopting an SMT (Satisfiability Modulo Theory) solver called *Z*3 prover. An example is given to show the usefulness of our method.

Keywords: Program Net, Software Testing, Constraint Condition, SMT Solver

1. INTRODUCTION

Software testing is done to find bugs, defects, or errors in a software program [1] and is indispensable for all software development. It is a critical element of software quality assurance and represents the final review of specification, design, and coding [2]. Path testing, an important aspect of software testing, searches for suitable test data that covers every possible path in the software under test. Generally, software testing takes a great deal of computational time and is an NP-complete problem [3]. Among so many testing activities, test data generation is one of the most intellectually demanding tasks and also is one of the most critical ones, since it has a strong impact on the effectiveness and efficiency of the whole testing process [4]. It is not surprising that a great amount of research effort in the past decades has been spent on test data generation. As a result, a good number of different techniques of test data generation have been advanced and investigated intensively [4]: such as (a) structural testing using symbolic execution, (b) model-based testing, (c) combinatorial testing, (d) random testing and its variant of adaptive random testing, and (e) search-based testing. In this paper, we develop a new method for test data generation by adopting a so-called Data-Flow Program Net [5].

As a kind of graph, a data-flow program net (program net or net for short) is an important way to study data-flow program [5]. A program net can be expressed by a directed graph and can be specially tuned for data-flows through arithmetic and logical operations. Figure 1 shows a general program net. In a program net, a node represents an operator, fork, switch, and so on. An edge represents a communication channel of tokens between nodes. A token represents an abstract datum whose structure varies from edge to edge [6].

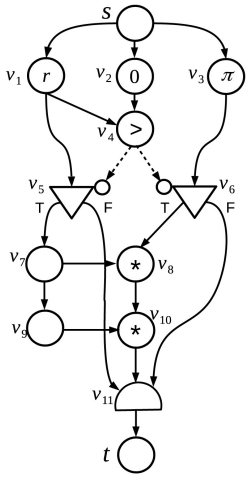

Fig.1: *An example of program net to calculate the area πr^2 of circle, where, r is the radius of circle.*

Treating a program as a program net, we try to apply program nets to our software testing problem theoretically through two steps: (1) to divide a program net into subnets, which can together structurally cover all the nodes of the program net, and

Manuscript received on June 26, 2019 ; revised on September 4, 2019.
Final manuscript received on October 26, 2019.
[1] The author is Graduate School of East Asian Studies, Yamaguchi University, Yamaguchi-shi, 753-8514 Japan., E-mail: w501sn@yamaguchi-u.ac.jp
[2] The author is Faculty of Education, Yamaguchi University, Yamaguchi-shi, 753-8513 Japan., E-mail: gqw@yamaguchi-u.ac.jp
DOI 10.37936/ecti-cit.2020141.197859

(2) for each subnet, generate test input data through analysis of the properties of the subnet. All of the paths of a program may be tested if each subnet can execute under the obtained test input data in turn. Step (1) has been accomplished in our previous work [7] which can (i) treat a program as a program net, (ii) find and delete back edges to construct an acyclic program net, and (iii) generate a set of subnets that cover all of the nodes of the acyclic program net. This paper aims to do the remaining task (Step (2)), which is how to find test input data for each subnet. Specifically, the expected outcome in this paper is to find a set of certain variable values for each obtained subnet under which all parts of the subnet can be executed. In a subnet obtained in our previous work [7], a SWITCH-node representing a switch operator is fixed only in one of the two states, $True$ and $False$. Thus we call these subnets *Choiceless Program Nets* (Choiceless Nets for short) in this paper. Since finding test input data is time-consuming and often technically difficult to perform [8] directly for a whole program (program net), this paper focuses on such choiceless program nets that have test input data.

This paper is organized as follows: Section 2 introduces definitions of general program nets and the subnets generated in our previous works and gives a brief introduction to the Satisfiability Modulo Theories used in this paper. In Section 3, the properties of choiceless program nets are analyzed and algorithms are shown to obtain the constraint conditions from a choicless program net. Then a method is presented to find test input data by using a tool, the $Z3$ SMT solver, for a choicless program net. In Section 4, an example is given to show how to find test input data by using the provided algorithms and the $Z3$ SMT solver.

2. PRELIMINARY

2.1 Program Net

A program net [9] is denoted by $PN=(V, E, \alpha, \beta)$, where V is a set of nodes consisting of AND-nodes ($\bigcirc$), OR-nodes ($\triangle$ or a semicircle), SWITCH-nodes ($^{\circ}\bigtriangledown$), and E is a set of directed edges between nodes. The token ($\bullet$) represents a single datum and token distribution $d^{\tau}=(d^{\tau}_{e_1}, d^{\tau}_{e_2}, \cdots, d^{\tau}_{e_{|E|}})$ that expresses token numbers on each edge e_i at time τ. α is the token threshold of node firing on the input edges. β is the token threshold of node firing on the output edges. If a program net PN is given with an initial token distribution d^0 onto edges, then it is called a marked program net and denoted by $MPN=(PN, d^0)$. PN is called acyclic if there are no directed circuits and PN is called SWITCH-less if there are no SWITCH-nodes.

Definition 1 [10]: Let v be a node, with e_{ip} and e_{op} as its data-flow input edge and output edge, respectively.

(i) An AND-node v_{AND} is firable with respect to d^{τ} if each input edge e_{ip} satisfies $d^{\tau}_{e_{ip}} \geq \alpha_{e_{ip}}$. If a firable node is fired, $\alpha_{e_{ip}}$ tokens are removed from each input edge e_{ip} and $\beta_{e_{op}}>0$ tokens are placed on each output edge e_{op}. The start node s fires only once.

(ii) An OR-node v_{OR} is firable with respect to d^{τ} if one input edge e_{ip} satisfies $d^{\tau}_{e_{ip}} \geq \alpha_{e_{ip}}$. When an OR-node is fired, $\alpha_{e_{ip}}$ tokens are removed from an arbitrarily selected input edge e_{ip} but not from the other, and $\beta_{e_{op}}$ tokens are placed on the output edge e_{op}.

(iii) A SWITCH-node v_{SW} is firable with respect to d^{τ} if its input data-flow edge e_{ip} satisfies $d^{\tau}_{e_{ip}} \geq \alpha_{e_{ip}}$ and its control-flow edge e_{ctrl} satisfies $d^{\tau}_{e_{ctrl}} \geq 1$. If the value of the control token is True (or False), then the token on the data input is directed to the output terminal True (or False) and the control token is removed. □

Definition 2 [11]: Let MPN be a marked program net.

(i) If node v is firable with respect to token distribution d then v is called d-firable.

(ii) A node sequence $\sigma=v_1 v_2 \cdots v_k$ is a firing sequence if and only if (iff for short) v_i is d^{i-1}-firable and d^i results from d^{i-1} by firing v_i.

(iii) An MPN is called terminating iff all of its firing sequences are of finite length $k<\infty$. □

Definition 3 [11]: An MPN is token self-cleaning (or self-cleaning for short) iff the following two conditions hold.

(i) MPN is terminating;

(ii) There is no token remaining on the edges after execution of any terminating firing sequence. □

Definition 4 [12]: Let v_1 and v_2 be two nodes of PN.

(i) If there is a directed path from v_1 to v_2, then v_1 is *predecessor* of v_2 and v_2 is *successor* of v_1. The sets of *predecessors* and *successors* of node v are denoted by $Pre(v)$ and $Suc(v)$, respectively.

(ii) If $(v_1, v_2) \in E$, then v_1 is an *immediate predecessor* of v_2 and v_2 is an *immediate successor* of v_1. The sets of *immediate predecessors* and *immediate successors* of node v are denoted by $IP(v)$ and $IS(v)$, respectively. □

In this paper, we assume that all marked program nets are self-cleaning program nets with $d^0=\mathbf{0}$ as in our previous work [7]. We have designed algorithms to generate subnets (denoted by $\{\widehat{PN}_i\}$) that can structurally cover all the nodes of a given program net [7]. These algorithms mainly carry out the following two steps: (1) making the given program net acyclic, and (2) generating subnets based on the acyclic program net. For example, for the program net shown in Figure 1, we get the subnets set shown in Figure 2.

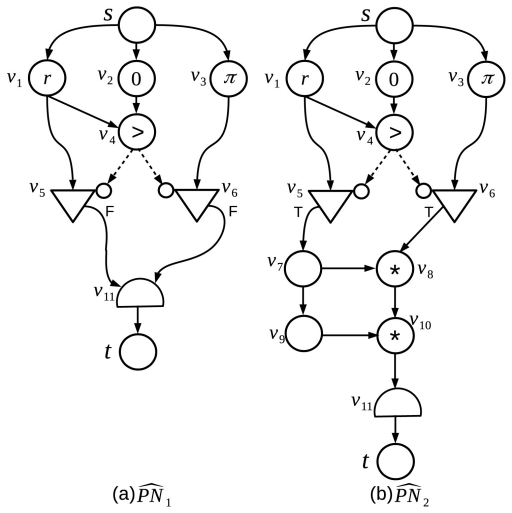

Fig.2: *Generated subnets of Figure 1.*

2.2 SMT Problem

Satisfiability Modulo Theory (SMT for short) is one of the central problems with both theoretical and practical interests in computer science. It is the problem to determine whether a formula expressing constraint conditions has a solution [13]. For example, a solution of the following formula is a set of variable values, e.g. (x, y, z)=$(1, 2, 1)$, which makes the formula satisfiable.

$$x{+}y{\geq}0{\wedge}(x{=}z{\Rightarrow}x{+}z{>}1){\wedge}{\neg}(x{>}y)$$

In recent years, there has been enormous progress in dealing with a large scale of problems that can be solved by SMT solvers [13], such as $Z3$, Yices, and so on [14, 15]. If the constraint conditions of test input data for each subnet can be obtained, we need only to use SMT solvers to find the test input data, which means our software testing problem can be solved. Therefore, we try to generate the constraint conditions and apply SMT solvers to find test input data in this paper.

3. CONSTRAINT CONDITIONS GENERATION

In this section, we provide algorithms to generate constraint conditions for the choiceless program nets obtained in Ref. [7] by analyzing the nets.

3.1 Analysis of Choiceless Program Net

We have structurally constructed a set of subnets that can cover all nodes for a given program net in our previous work [7]. The program nets have been represented around the structure so far without paying much attention to the detailed behavior of the nodes. However, in order to generate test input data, we must mine how the data is concretely operated upon and how it flows in the subnet. Since the OR-nodes and SWITCH-nodes function as fixed operations, merging input data and switching data to True or False terminals respectively, we need only to clarify the detailed description of the operations for the AND-nodes.

The operation at an AND node v_{AND} can be regarded as one of three operation forms: (a) handling logical-operation, (b) executing arithmetic-operation, or (c) duplicating its input data. Logical and arithmetic operators are generally those included in the list, O=$(<, >, \leq, \geq, !, !=, \&\&, ||, \&, |, \hat{}, =, +, -, *, /, \%, \ll, \gg, ++, --, ==, \sim)$, as shown in Table 1. This operator list can be further divided into logical and arithmetic operator sublists, O_l=$(<, >, \leq, \geq, !, !=, \&\&, ||, ==)$ and O_a=$(\&, |, \hat{}, =, +, -, *, /, \%, \ll, \gg, ++, --, \sim)$ respectively.

To obtain constraint conditions, we need to express operation results for all the AND-nodes. Hence it is essential (i) to indicate from which input edge the input data comes, and (ii) to clearly specify whether the input data should be placed before or after the operator at the AND-node. Although an AND-node generally may contain three or more input edges, we limit any AND-node to one that possesses at most two input edges in this paper in order to conveniently express constraint conditions. We can do so due to the fact that any AND-node with three or more input edges can be simply transformed to ones with two input edges as shown in Figure 3.

Table 1: *Common operator list O.*

Index	Operator	Description
0	$<$	less than
1	$>$	greater than
2	$\leq$	less than or equal to
3	$\geq$	greater than or equal to
4	!	logical not
5	!=	not equal to
6	&&	logical and
7	$\|\|$	logical or
8	&	bitwise and
9	$\|$	bitwise or
10	$\hat{}$	bitwise xor
11	$=$	assignment
12	$+$	addition
13	$-$	subtraction
14	$*$	multiplication
15	$/$	division
16	%	modulus
17	$\ll$	left shift
18	$\gg$	right shift
19	$++$	increment
20	$--$	decrement
21	$==$	equal to
22	$\sim$	bitwise not

Based on the previous discussions, we extend an ordinary program net to an *Exhaustive Program Net* in the following definition.

Definition 5: A program net is called an Exhaustive Program Net and denoted by EN=$(V, E, g, o, r, \alpha, \beta)$ if each AND-node possesses at most two input edges

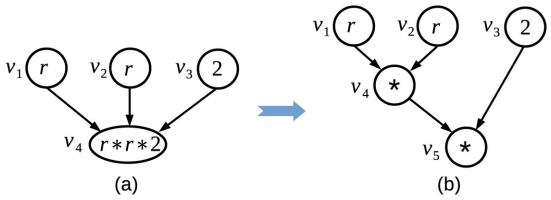

Fig.3: *Transform operation.*

and furthermore the following conditions are satisfied:

(i) V is a set of nodes consisting of AND-nodes, OR-nodes and SWITCH-nodes;

(ii) E is a set of directed edges between nodes. The solid arrow ($\longrightarrow$) represents a *dataflow-edge* and the hidden arrow ($\dashrightarrow$) shows a *controlflow-edge*;

(iii) $g(v)$ expresses the operation result at node v;

(iv) $o(v)$ expresses the operator at node v as follows:

$$o(v)=\begin{cases} op\in O, & v \text{ is an AND-node with logical} \\ & \text{or arithmetic operator} \\ NULL, & \text{otherwise} \end{cases}$$

(v) $r(e)$ is marked on an input edge e of v as follows:

$$r(e)=\begin{cases} ① \text{ or } ②, & v \text{ is an AND-node with logical} \\ & \text{or arithmetic operator} \\ NULL, & \text{otherwise} \end{cases}$$

Suppose $e=(v', v)$ and $g(v')$ are the operation result of v' flowing through e to v. (i) If $r(e)$=①, $g(v')$ is placed before $o(v)$ (i.e., "$g(v')o(v)$" is operated at v); (ii) If $r(e)$=②, $g(v')$ is placed after $o(v)$; (iii) If $r(e)=NULL$, $g(v')$ just passes through v;

(vi) α is token threshold of node firing on the input edges and β is token threshold of node firing on the output edges. □

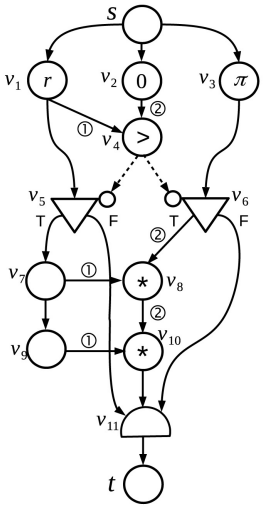

Fig.4: *Exhaustive program net EN of Figure 1.*

Hereafter, a set of subnets $\{\widehat{PN}_i\}$ (choiceless nets) obtained in [7] is expressed by a set of exhaustive nets and denoted by $\{\widetilde{EN}_i\}$. Figure 4 shows the exhaustive net of Figure 1 and Figure 5 shows the choiceless exhaustive subnets of Figure 4, $\widetilde{EN}_1$ and $\widetilde{EN}_2$. In the following discussions, when we say choiceless net denoted by $\widetilde{EN}$, we mean a choiceless exhaustive subnet.

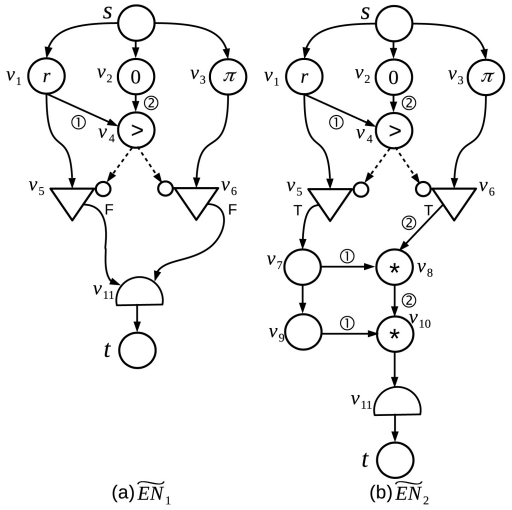

Fig.5: *Choiceless nets $\widetilde{EN}_1$ and $\widetilde{EN}_2$ of Figure 4.*

Let's see how to use $o(v)$ and $r(e)$ to express operation results of the nodes in Figure 6. v_3 has two input edges, e_1 and e_2 marked with ① and ② respectively, and hence the operation result of v_3 is $g(v_3)=g(v_1)o(v_3)g(v_2)=g(v_1)>g(v_2)$ according to Definition 5 (v). For Figure 6 (b), $g(v_5)=$ $g(v_4)o(v_5)=g(v_4)$++ holds, since $r(e_3)$=①. Also $g(v_7)=o(v_7)g(v_6)=!g(v_6)$ due to $r(e_4)$=②. In this way, it is possible to get a series of expressions of operation results that are what we want to find, the constraint conditions. Particularly, constraint conditions for a choiceless net $\widetilde{EN}$ are the expressions of operation results of control-flow input nodes, which must meet the T or F states of each of the related SWITCH-nodes in a choiceless net $\widetilde{EN}$. For $\widetilde{EN}_2$ shown in Figure 5 (b), the operation results of control-flow input nodes, v_4, are $g(v_4)=g(v_1)o(v_4)g(v_2)=(r>0)$. Since all the SWITCH-nodes, v_5 and v_6, have only T terminals, $(r>0)$ is the constraint condition.

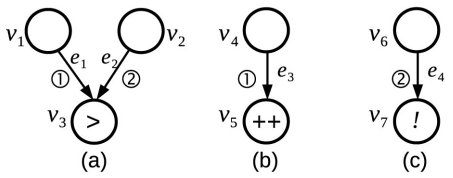

Fig.6: *Example AND-nodes for expression of operation result.*

Choiceless nets have the properties shown in the following propositions.

Proposition 1: Let $\widetilde{EN}$ be a choiceless net. If all the SWITCH-nodes fire and further output data to their terminals that exist in $\widetilde{EN}$ under a set of input data, then all the nodes of $\widetilde{EN}$ can fire. □

Proof: Since each SWITCH-node fires and outputs data to its T or F terminal that is included in $\widetilde{EN}$, we can fire each node in such a way: (i) firing the start node and deleting it, and then (ii) recursively firing the source nodes and deleting them. Finally all the nodes will be deleted. In other words, all the nodes can fire, since there is no directed circuit in $\widetilde{EN}$. □

Proposition 2: Let v_{SW} be a SWITCH-node in a choiceless net $\widetilde{EN}$ and $Suc(v_{SW})$ be a set of its successor nodes. If $Suc(v_{SW})$ contains no SWITCH-nodes, then there exist no constraint conditions in the subnet induced by $Suc(v_{SW})$. □

This proposition obviously holds, so we can simply delete such subnets induced by $Suc(v_{SW})$ in generating constraint conditions.

From the previous discussion, we know that each SWITCH-node corresponds to one constraint condition, and hence our next work is to obtain such all constraint conditions for each SWITCH-node existing in a choiceless net $\widetilde{EN}$. In order to facilitate the generation of constraint conditions, we construct an adjacency matrix A to express all of the information of a choiceless net $\widetilde{EN}$. The following adjacency matrix A is constructed to represent the net of Figure 5 (b).

$$A=\begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 2 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -14 & 0 & 2 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -14 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

We now give a detailed explanation of how to construct the adjacency matrix A. Suppose $i\neq j$ holds below. First, diagonal values $\{a_{ii}\}$ of A express the types and operations of nodes as follows:

$$\begin{cases}
a_{ii}<0,\ v_i \text{ is AND-node with logical or arithmetic operator and } o(v_i)=O[-a_{ii}]\\
a_{ii}=1,\ v_i \text{ is other AND-node}\\
a_{ii}=2,\ v_i \text{ is OR-node}\\
a_{ii}=3,\ v_i \text{ is SWITCH-node}
\end{cases}$$

Then non-diagonal values $\{a_{ij}\}$ express the types of edges between nodes.

$$\begin{cases}
a_{ij}=-1, & \text{edge } (v_i, v_j) \text{ is control-flow edge}\\
a_{ij}>0, & \text{edge } (v_i, v_j) \text{ is data-flow edge}
\end{cases}$$

Furthermore, when $a_{ij}>0$: (i) if $a_{jj}<0$, a_{jj} expresses the operation of the node v_j; (ii) if $a_{jj}>0$, a_{ij} that means there is an edge (v_i, v_j) such that:

$$\begin{cases}
a_{ij}=r(e), & \text{edge } e=(v_i, v_j) \text{ and } a_{jj}<0\\
a_{ij}=1, & \text{edge } e=(v_i, v_j) \text{ exists and } a_{jj}>0
\end{cases}$$

Finally, we represent the state of SWITCH-node (T or F) in A. The following rule is applied only when node v_i is a SWITCH-node ($a_{ii}=3$).

$$\begin{cases}
a_{ij}=r(e)\times 10, & \text{edge } e=(v_i, v_j) \text{ is } F \text{ terminal of } v_i\\
a_{ij}=r(e), & \text{edge } e=(v_i, v_j) \text{ is } T \text{ terminal of } v_i
\end{cases}$$

Note that the values of $r(e)$, ① and ②, are replaced by 1 and 2 in the matrix A.

3.2 Generation of Constraint Conditions

The entire process of applying program nets to solving our software testing problem is shown in Figure 7. Among them, (a) has been solved by the polynomial algorithms in our previous work [7]. What we need to do next is (b) and (c), which are generating constraint conditions and finding test input data.

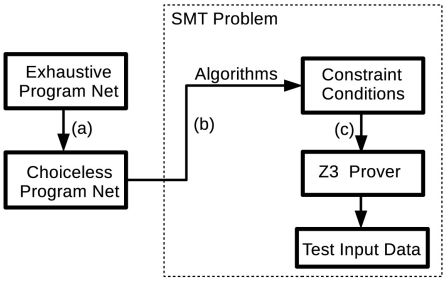

Fig.7: *The procedures of generating test input data.*

Since each SWITCH-node corresponds to one constraint condition in a choiceless net $\widetilde{EN}$, we can find a way to obtain such all constraint conditions by doing two steps: (1) determining the boolean value of the corresponding constraint condition of each SWITCH-node, and then deleting sink nodes iteratively until all the sink nodes become SWITCH-nodes, which can be done according to Proposition 2; (2) designing algorithms to obtain constraint conditions from the net obtained in step (1). Algorithm 1 is used for step (1).

Algorithm 1 Construction of Simplified Program Net

Require: $\widetilde{EN}=(V, E, o, g, r, \alpha, \beta)$, A of $\widetilde{EN}$
Ensure: Simplified Program Net G

1: **while** $V \neq \phi$ **do**
2: take out a node v_x from V
3: **if** v_x is a SWITCH-node (a_{xx}=3) **then**
4: **if** a_{xy}<10 where $x \neq y$ **then**
5: $q(v_j) \leftarrow true$ where $a_{jx}=-1$
6: **else**
7: $q(v_j) \leftarrow false$ where $a_{jx}=-1$
8: **end if**
9: **end if**
10: **end while**
11: matrix $B \leftarrow A$
12: $G \leftarrow \widetilde{EN}$
13: **while** a sink node v_x that is not a SWITCH-node ($b_{xx} \neq 3$) exists in G **do**
14: $G \leftarrow G - \{v_x\}$
15: update B (delete v_x-related row and column)
16: **end while**
17: **Output** G

Theorem 1: Let $\widetilde{EN}$ be a choiceless net and G be the simplified net obtained by Algorithm 1.
(1) All sink nodes in G are SWITCH-nodes.
(2) The computational time complexity is $O(|V|^2)$. □
Proof: (1) It is obvious from lines 13-16 of the algorithm. (2) The execution of lines 1-10 takes $O(|V|^2)$, since lines 1-10 execute $|V|$ iterations and each iteration takes $|V|$ times to search for A. Lines 13-16 execute at most $|V|$ iterations, and each iteration takes at most $|V|$ times to search a sink node and $2|V|$ times to delete the row and the column, thus the execution of lines 13-16 also takes $O(|V|^2)$. Therefore, the total computational time complexity is $O(|V|^2)$. □

Algorithm 2 is used for step (2).

Algorithm 2 Generation of Constraint Conditions

Require: G, adjacency matrix B of G
Ensure: Constraint conditions set CCS
1: mark all nodes "unused"
2: queue $Q \leftarrow \phi$
3: enqueue s (start node of G) to Q
4: **while** $Q \neq \phi$ **do**
5: dequeue an element v_i from Q
6: **if** there exists an output "unused" node v_j from v_i **then**
7: **if** $b_{jj} > 0$ **then**
8: **if** b_{jj}=1 **then**
9: **if** v_j has no "unused" input node **then**
10: $g(v_j) \leftarrow g(v_i)$
11: mark v_j "used"
12: enqueue v_j to Q
13: **end if**
14: **else if** v_j is OR-node (b_{jj}=2) **then**
15: **if** v_j is "unused" **then**
16: $g(v_j) \leftarrow g(v_i)$
17: mark v_j "used"
18: enqueue v_j to Q
19: **end if**
20: **else if** v_j is SWITCH-node (b_{jj}=3) **then**
21: **if** v_j is "unused" **then**
22: $g(v_j) \leftarrow g(v_x)$ where $b_{xj} = 1$
23: mark v_j "used"
24: enqueue v_j to Q
25: **end if**
26: **end if**
27: **else if** b_{jj}<0 **then**
28: **if** v_j has no "unused" input node **then**
29: **if** b_{ij}=1 **then**
30: $g(v_j) \leftarrow g(v_i)O[-b_{jj}]$
31: **else if** b_{ij}=2 **then**
32: $g(v_j) \leftarrow O[-b_{jj}]g(v_i)$
33: **else if** b_{ij}=1 and b_{xj}=2 **then**
34: $g(v_j) \leftarrow g(v_i)O[-b_{jj}]g(v_x)$
35: **else if** b_{ij}=2 and b_{xj}=1 **then**
36: $g(v_j) \leftarrow g(v_x)O[-b_{jj}]g(v_i)$
37: **end if**
38: enqueue v_j to Q
39: **if** v_j has control output edge (b_{jx}=−1) **then**
40: **if** $q(v_j) = true$ **then**
41: $CCS \leftarrow CCS \cup \{g(v_j)\}$
42: **else if** $q(v_j) = false$ **then**
43: $CCS \leftarrow CCS \cup \{\neg g(v_j)\}$
44: **end if**
45: **end if**
46: **end if**
47: **end if**
48: **end if**
49: **end while**
50: **Output** CCS

Theorem 2: Let $\widetilde{EN}$ be a choiceless program net, G be the simplified net constructed from $\widetilde{EN}$ by Algorithm 1, and CCS be the set of constraint conditions obtained by Algorithm 2.
(1) If there exists a set of test input data satisfying CCS, then all the nodes of $\widetilde{EN}$ can fire.
(2) The computational time complexity is $O(|V|^2)$. □
Proof: (1) If there exists a set of test input data satisfying CCS, then each SWITCH-node fires to output data to its T or F terminal included in $\widetilde{EN}$. According to Proposition 1, all the nodes can fire. (2) The execution of lines 4-49 takes $O(|V|^2)$, since lines 4-49 execute at most $|V|$ iterations and each iteration takes at most $|V|$ times to calculate the operation result at lines 6-26. It takes at most $2|V|$ times to generate constraint condition at lines 27-37 when searching in B. Therefore, the computational time complexity is $O(|V|^2)$. □

By using Algorithm 2, we can obtain all constraint conditions of $\widetilde{EN}_2$ of Figure 5 (b). As a result, the constraint conditions can be synthesized as F: $r>0$. In the following discussions, we explain how to find a solution satisfying such F.

4. TEST INPUT DATA GENERATION

In this section, we aim to concretely generate test input data from the constraint conditions obtained in Section 3.

Till now, many SMT solvers have been developed to find a solution satisfying a constraint condition. Well-known SMT solvers include $Z3$ prover, Yices, SMT-RAT, and so on. The $Z3$ prover is a high performance theorem prover developed by Microsoft Research and can be used to check the satisfiability of logical formulas over one or more theories [15].

$Z3$ prover is such a useful and convenient tool because (i) it can handle decimals, (ii) it can handle multiplication and division between variables, (iii) one needs only to write constraint conditions without solving the equation, and (iv) it has been developed

as a package for the Python programing language and it can be used conveniently. Therefore, we chose the $Z3$ prover for generating test input data for choiceless nets.

```
File  Edit  Format  Run  Options  Window  Help
from Z3 import *
import math
s=Solver()

r=Int('r')

s.add(r>0)

print(s.model())
```

Fig.8: *The program for solving F using the $Z3$ prover.*

Figure 8 shows a short program to solve F: $r>0$ given in the last section. It is solved by using the $Z3$ prover in Python IDLE (Integrated Development and Learning Environment). The test input data $r=1$ can be obtained by executing this program for the choiceless net of Figure 5 (b). Similarly for the net of Figure 5 (a) with constraint conditions, $\neg(r>0)$, the test input data $r=0$ can be obtained.

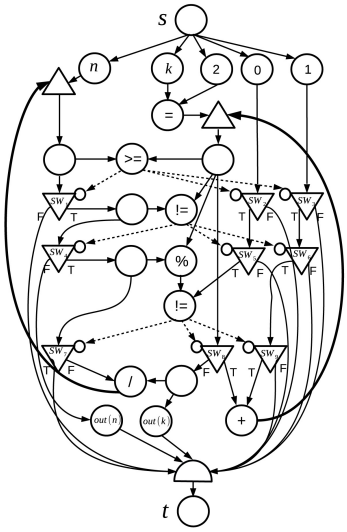

Fig.9: *A program net PN to decompose a positive integer into prime factors.*

Here, we show an example by applying our algorithms and the $Z3$ prover to generate a set of test input data for the program net PN shown in Figure 9. First, applying our previously documented method [7], we get the acyclic net $\widehat{PN}$ shown in Figure 10 whose corresponding exhaustive net EN is shown in Figure 11, together with four subnets, $\widetilde{EN}_1 \sim \widetilde{EN}_4$ that structurally cover the original net of Figure 9.

Second, the simplified nets $\widetilde{EN}_1' \sim \widetilde{EN}_4'$ of $\widetilde{EN}_1 \sim \widetilde{EN}_4$ are constructed by executing Algorithm 1, as shown in Figure 13, in which all of the sink nodes are SWITCH-nodes. For these nets, $\widetilde{EN}_1' \sim \widetilde{EN}_4'$, we can get their constraint conditions with Algorithm 2 as shown in Table 2.

Finally, applying the $Z3$ prover to the obtained constraint conditions, we get the four test input data values shown in the right column of Table 2. As a result, "$n=1, 2, 4, 5$" can be used as the test input data to check all the possible paths of the original net shown in Figure 9.

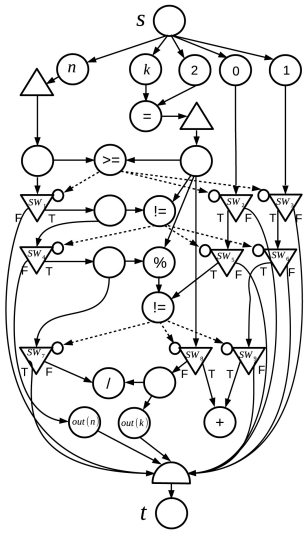

Fig.10: *Acyclic program net $\widehat{PN}$ of Figure 9.*

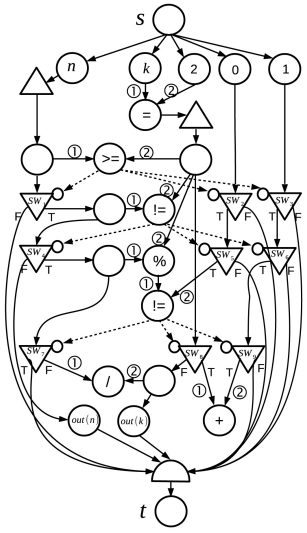

Fig.11: *Exhaustive program net EN of Figure 10.*

Table 2: *Constraint conditions and test input data.*

Nets	Constraint Conditions	Test Input Data
$\widetilde{EN}_1'$	$\neg(n\geq 2)$	$n=1$
$\widetilde{EN}_2'$	$(n\geq 2)\wedge(\neg(n!=2))$	$n=2$
$\widetilde{EN}_3'$	$(n\geq 2)\wedge(n!=2)\wedge(\neg(n\%2!=0))$	$n=4$
$\widetilde{EN}_4'$	$(n\geq 2)\wedge(n!=2)\wedge(n\%2!=0)$	$n=5$

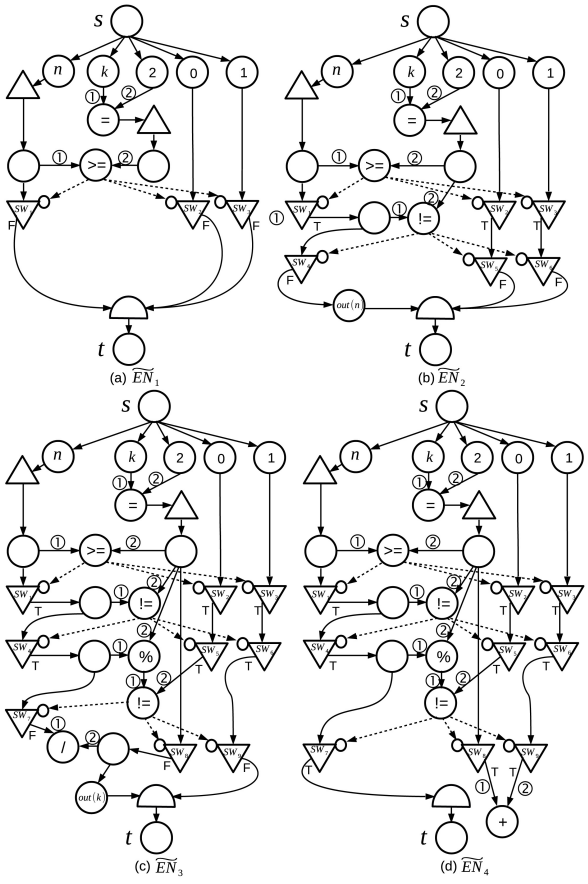

Fig.12: *Generated choiceless nets of Figure 11.*

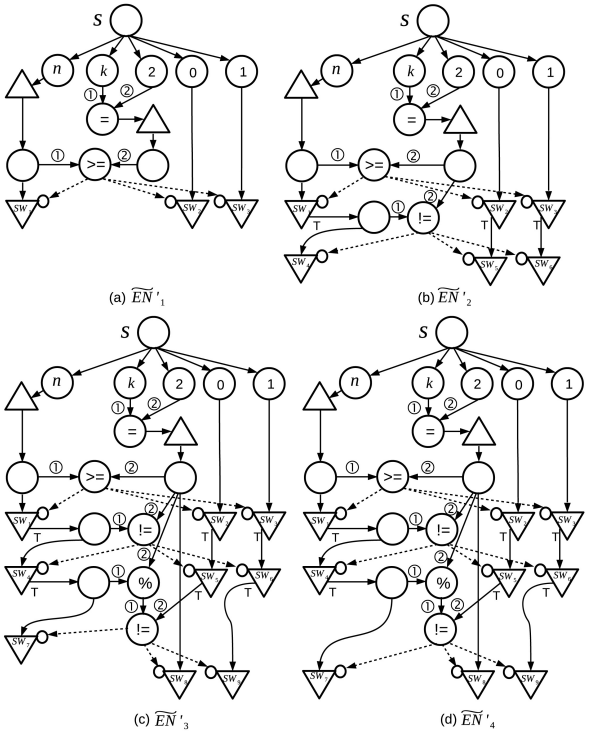

Fig.13: *Simplified program nets of Figure 12.*

From the above results, it is obvious that we can always find test input data for each subnet, which ensures all parts of each subnet executed. As mentioned in Section 1, software testing takes much computational time due to its NP-completeness, and generating test input data is an effective way to approach it. That is why we apply program nets to find proper test input data. Although these input data values are generated by the SMT solver that may take considerable computational time, our algorithms are all polynomial. This means that our method is feasible and effective to practically approach the software testing problem.

5. CONCLUDING REMARKS

Following our previous work [7] towards solving software testing problem by using program nets, we have created a method to (i) generate constraint conditions for the subnets that have been previously obtained in [7], and that can structurally cover the whole original program net, and (ii) it can find test input data based on the constraint conditions for each subnet by using the $Z3$ SMT solver. Algorithms have been designed for (i) with polynomial computational time. An example has been presented showing how to find test input data, clearly demonstrating the usefulness of our algorithms.

Including our previous work, we have basically presented a solution to a software testing problem related to finding suitable test input data by adopting the $Z3$ SMT solvers. Although all of our designed algorithms are polynomial, the computational time is mainly dependent on SMT solvers, since the software testing problem is generally NP-complete. Therefore, an efficient SMT solver is necessary when applying our method.

There is still some related future work to do: (1) to find such subnets that all definitely have input data making all the SWITCH-nodes firable, and (2) to improve the algorithms in [7] to find an optimal set of subnets for a given program net.

ACKNOWLEDGMENTS

The authors would like to thank Professor Xiaoan Bao and Associate Professor Na Zhang, Zhejiang Sci-Tech University, China, for supporting this work. We are also grateful to Mr. Hiromu Morita for his assistance in coding programs when he was studying at Yamaguchi University. Finally, we are greatful for the assistance given by Professor Mitsuru Nakata, Yamaguchi University, whose his expertise greatly assisted our work.

References

[1] B. Hetzel, *The Complete Guide to Software Testing*, John Wiley & Sons, Inc., 1993.

[2] B.T. Abreu, E. Martins, and F.L. Sousa, "Automatic test data generation for path testing using a new stochastic algorithm," *Proc. of the 19th Brazilian Symp. on Software Engineering*, vol.19, pp.247-262, 2005.

[3] M. Alzabidi, A. Kumar, and A.D. Shaligram, "Automatic software structural testing by using evolutionary algorithms for test data generations," *IJCSNS International Journal of Computer Science and Network Security*, vol.9, no.4, 2009.

[4] S. Anand, E.K. Burke, T.Y. Chen, J. Clark, M.B. Cohen, W. Grieskamp, M.Harman, M.J. Harrold, P. McMinn, A. Bertolino, J.J. Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol.86, no.8, pp.1978-2001, 2013.

[5] Q.W. Ge, T. Watanabe, and K. Onaga, "Execution termination and computation determinacy of data-flow program nets," *J. Franklin Institute*, vol.328, no.1, pp.123-141, 1991.

[6] Q.W. Ge, T. Watanabe, and K. Onaga, "Topological analysis of firing activities of data-flow program nets," *IEICE Trans. Fundamentals*, vol.E73, no.7, pp.1215-1224, 1990.

[7] B. Wu, X. Bao, N. Zhang, H. Morita, M. Nakata, and Q.W. Ge, "Subnets generation of program nets and its application to software testing," *IEICE Trans. Fundamentals*, vol.E102, no.9, pp.1303-1311, 2019.

[8] J. Mei, S.Y. Wang, "An improved genetic algorithm for test cases generation oriented paths," *Chinese journal of electronics*, vol.23, no.3, pp.494-498, 2014.

[9] Q.W Ge, T. Watanabe, and K. Onaga, "Computation of minimum firing time for general self-cleaning SWITCH-less program nets," *IEICE Trans. Fundamentals*, vol.E81, no.6, pp.1072-1078, 1998.

[10] S. Yamaguchi, T. Takai, T. Watanabe, Q.W. Ge, and M. Tanaka, "Complexity and a heuristic algorithm of computing parallel degree for program nets with SWITCH-nodes," *IEICE Trans. Fundamentals*, vol.E89, no.11, pp.3207-3215, 2006.

[11] Q.W. Ge, and K. Onaga, "On verification of token self-cleanness of data-flow program nets," *IEICE Trans. Fundamentals*, vol.E79, no.6, pp.812-817, 1996.

[12] Q.W. Ge, C. Li, and M. Nakata, "Performance evaluation of a two-processor scheduling method for acyclic SWITCH-less program nets," *IEICE Trans. Fundamentals*, vol.E88, no.6, pp.1502-1506, 2005.

[13] H. Wang, J. Xing, Q. Yang, W. Song, and X.W. Zhang, "Generating effective test cases based on satisfiability modulo theory solvers for service-oriented workflow applications," *Software Testing, Verification and Reliability*, vol.26, no.2, pp.149-169, 2016.

[14] B. Dutertre and L.D. Moura, "A fast linear-arithmetic solver for DPLL(T)," *Proc. of the 16th International Conference on Computer Aided Verification*, vol. 4144, pp.81-94, 2006.

[15] L.D Moura and N. Bjϕrner, "Z3: An efficient SMT solver," *International conference on Tools and Algorithms for the Construction and Analysis of Systems.*, Springer, Berlin, Heidelberg, pp.337-340, 2008.

Biao Wu was born in 1989. He received his B.S. from Hubei University of Technology in 2013, and an M.E. from Zhejiang Sci-Tech University in 2016, both in the People's Republic of China. He is currently a Ph.D. candidate in the Graduate School of East Asian Studies, Yamaguchi University, Japan. His main research interests include software testing and program net theory.

Qi-Wei Ge received his B.E. from Fudan University, China, in 1983, and his M.E. and Ph.D. degrees from Hiroshima University, Japan, in 1987 and 1991, respectively. He was with Fujitsu Ten Limited from 1991 to 1993. He was an Associate Professor at Yamaguchi University, Japan, from 1993 to 2004. Since April of 2004, he has been a Professor at Yamaguchi University, Japan. His research interest includes Petri nets, program net theory and combinatorics. He is a member of the Institute of Information Processing Society of Japan (IPSJ) and the Institute of Electrical and Electronics Engineers (IEEE).