山口大学大学院東アジア研究科
博士論文

A Study on Test Data Generation for Software Testing by Applying Program Nets

2021 年 3 月

呉　彪

Abstract

Title of Thesis: A Study on Test Data Generation for Software Testing by Applying Program Nets

Name of Degree Candidate: Biao Wu
Asian Education Course
The Graduate School of East Asian Studies
Yamaguhci University, Japan

Degree and Year: Doctor of Philosophy, 2021

Dissertation directed by: Dr. Qi-Wei Ge
Professor
The Graduate School of East Asian Studies
Yamaguhci University, Japan

Along with the development of information technology, more and more software products have been applied to most aspects of society and facilitated people's lives. Consequently, the problems of software quality and security have been paid more attention. Software testing is an important means to ensure software quality and reliability, which is to find bugs, defects, or errors in a software program and is indispensable for all software development since it is a critical element of software quality assurance and represents the final review of specification, design, and coding.

Path testing is an important measure of general software testing, which searches specific test input data that covers every possible path in the software program. Among testing activities, test data generation is one of the most intellectually demanding tasks and also one of the most critical ones, since it has a strong impact on the effectiveness and efficiency of the whole testing process. However, with the increasing of complexity and

scale of software, a software program may contain an infinite number of paths when the program has loops. In addition, the number of paths is also exponential to the number of branches of the program. As a result, test input data generation takes much computational time and is an NP-complete problem, which makes input data generation more complex.

Graphs can capture complex data dependencies and be widely used to model and analyze data-flow programs. As a kind of graph, data-flow program net (program net or net for short) is an important way to study data-flow programs. Program net is specially tuned to data-flows through arithmetic and logical operations. This dissertation discusses how to apply program nets to generate test input data for approaching the software testing problem theoretically. This dissertation is organized as follows: Chapter 1 gives an introduction of the research background and presents the motivation and the target of this dissertation.

Chapter 2 presents the basic definitions of program nets and proposes a necessary extension of program nets so-called exhaustive program nets to describe the dynamic behaviour information of program nets. The known basic properties and the whole process of applying program nets to test input data generation are also given.

Chapter 3 presents a proposal to generate such required subnets' set that can cover all nodes of a given program net and gives its corresponding algorithms to (1) construct the layer net of the given program net, (2) construct the acyclic program net from the given program net with directed circuits, (3) obtain the initial subnet, and (4) generate the remaining subnets so that the obtained subnets set containing the initial subnet can cover all nodes of the given program net in order to finally find a specific test data used in software testing.

Chapter 4 presents a method to find a specific test input data for such subnets that may possess input data based on the analysis result of the behaviours of all the subnets. Firstly, a brief introduction is given to Satisfiability Modulo Theories (SMT for short) that can be used to find test input data. Then, discussion is done on how to equivalently transform the subnets obtained in Chapter 3 into exhaustive subnets. After that, an algorithm is designed to obtain all constraint conditions from a given exhaustive subnet. Finally, we introduce an SMT solver, $Z3$ Prover, which can find the specific test input data according to the constraint conditions.

Chapter 5 gives a case study that shows how to generate required subnets for three actual Java programs by using the algorithms proposed in Chapter 3, as well as to indicate how to find a set of test input data for the subnets by using the SMT solver introduced in Chapter 4. Also, the experimental results are shown and discussed for the Java programs.

Chapter 6 concludes the results obtained in this dissertation and discusses future research works remaining to be solved.

学位論文内容要旨

学位論文題目：プログラムネットを適用したソフトウェアテスティング
　　　　　　　のためのテストデータ生成に関する研究

指導教員：葛　崎偉　教授

申請者名：呉　彪 (ゴ　ヒョウ)

平成 28 年度入学

山口大学大学院 東アジア研究科 (博士後期課程)
アジア教育開発コース

情報技術の発展に伴い, コンピュータプログラムであるソフトウェアが社会のあらゆる場面で活用され, 豊かな情報化社会の基盤となっている. それに従って, ソフトウェアの品質や信頼性, 及び, セキュリティの問題がこれまで以上に注目されつつある. ソフトウェアテスティングは, ソフトウェアの品質と信頼性を確保するための重要な手段であり, ソフトウェアのバグや脆弱性の発見, 仕様への適合性の確認などに広く用いられている. また, ソフトウェアの仕様設計, コーディングおよび最終レビューなどのソフトウェア開発の多くの過程において繰り返し実施される必要不可欠なプロセスである.

広く用いられているソフトウェアテスティング手法として, パステスティングが挙げられる. これは, ソフトウェアの処理経路 (パス) を網羅的に実行して正しく動作するか否かを確認する手法である. この手法の有効性と効率性は, テストの際に入力するデータ (テスト入力データ) の生成に大きな影響をうける. そのためパステスティングにおいて, テスト入力データの生成は緻密さと効率の両者を要求される最重要のタスクといえる. しかし, ソフトウェアの複雑さと規模が増大するにつれて, パスの数は爆発的に増えていく. 例えば, プログラム内の分岐 (ブランチ) の数に対してパスの数は指数関数的に増えるし, プログラムがループ構造を含む場合

はパスのループが生じる可能性もある. それ故に, 一般的にパステスティングはテスト入力データの生成に多大な計算時間がかかる NP 完全問題として知られている.

本論文では, パステスティングの有効性と効率の向上を目的として, データフロープログラムネットを活用したテスト入力データ生成手法を提案する. データフロープログラムネット (略してプログラムネットまたはネット) とは, 様々な現象を表現·分析するために広く用いられているグラフの一種であり, プログラムにおける算術演算や論理演算等によるデータの流れ (データフロー) を表す. 本論文は次のように構成されている.

第 1 章では, 研究の背景を紹介し, 本論文の動機と目標を示す.

第 2 章では, プログラムネットの基本的な定義を示し, プログラムネットの動的な情報の記述に必要なプログラムネットの拡張 (Exhaustive Program Net) について論じる. また, 既知の基本的なプログラムネットの性質と, プログラムネットを利用した入力データの生成に関するプロセス全体の考え方について説明する.

第 3 章では, プログラムネットに含まれるすべてのノードをカバーするサブネットセットを生成するために, 次の方法に関連するアルゴリズムを提案する. (1) 特定のプログラムネットのレイヤーネットを構築する方法, (2) 有向回路を含む特定のプログラムネットから非巡回プログラムネットを構築する方法, (3) 初期サブネットを取得する方法, (4) ソフトウェアテスティングで使用される特定のテストデータを最終的に見つけるために, 初期サブネットを含む取得されたサブネットセットが特定のプログラムネットのすべてのノードをカバーできるように、残りのサブネットを生成する方法.

第 4 章では, すべてのサブネットの動作の分析結果に基づいて, 入力データをもつ可能性のあるサブネットの特定のテスト入力データを見つける方法を示す. 最初に, テスト入力データを見つけるために Satisfiability Modulo Theories (SMT) について簡単に紹介する. 次に, 第 3 章で得られたサブネットを等価な Exhaustive Subnet に変換する方法について説明し, Exhaustive Subnet に関する制約条件を導き出すためのアルゴリズムを設計する. また, 制約条件に基づいて特定のテスト入力データを見つけるこ

とのできる SMT ソルバーである Z3 Prover を紹介する.

第 5 章では, 第 3 章と第 4 章で提案した方法を利用したケーススタディを示す. まずは, 第 3 章で提案したアルゴリズムを活用して 3 つの実際の Java プログラムを対象とするサブネットを生成する. 次に, 第 4 章で導入された SMT ソルバーを使ってサブネットのテスト入力データのセットを見つける. また, 3 つの Java プログラムに対する実験結果を示し, その結果に関する議論を行う.

第 6 章では, 本論文で得られた結果をまとめ, 今後の課題について述べる.

A STUDY ON TEST DATA GENERATION FOR SOFTWARE TESTING BY APPLYING PROGRAM NETS

by

BIAO WU

Dissertation submitted to the Graduate School of East Asian Studies of Yamaguchi University,
in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Advisory Committee:

Professor Qi-Wei GE, Chairman/Advisor
Professor Ryo TAKAOKA
Professor Mitsuru NAKATA
Associate Professor Chisato KITAZAWA

©Yamaguchi University. All rights reserved.

Acknowledgements

Time flies, and in a flash, four and a half years' education career in Japan is coming to an end. On the completion of this Ph.D. dissertation, I would like to express my deep sincere appreciation and best wishes to those who have cared, helped and supported me in the past years.

First and foremost, I would like to express my sincere appreciation to my supervisor Professor Qi-Wei Ge. Thanks to Professor Ge for his guidance, continuous support and encouragement, I was able to complete my doctoral degree. I have been fortunate to study in a foreign country under his warm care. I have been fortunate to learn his rigorous scientific attitude and approachable character. All I can say is that without him, I would not be where I am today.

I am very grateful to the other members of the examination committee for this dissertation: Professor Ryo Takaoka, Professor Mitsuru Nakata, and Associate Professor Chisato Kitazawa. Their careful reading and precious comments led to substantial improvement in the final product.

I would like to thank Associate Professor Rie Tanaka and Professor Mitsuru Nakata for their helpful advices. I am also grateful to others in the same laboratory, Mr. Quan Gan, Mr. Zhenyu An, and master student Mr. Hiromu Morita of the same research group when he studied at Yamaguchi University, etc., who have made my stay in Japan an enjoyable experience.

And finally, I express my grateful thanks to my family, without their love and great upbringing I would never have been able to accomplish this dissertation.

Notation

$x{\in}X$:	x is an element of set X.
$x{\notin}X$:	x is not an element of set X.
$X{\subset}Y$:	Set X is contained in set Y.
$X{\cup}Y$:	Union of sets X and Y.
$X{\cap}Y$:	Intersection of sets X and Y.
$X{-}Y$:	Difference of sets X and Y.
$X{-}x$:	Delete the element x from X.
$max\ X$:	The maximum element of set X.
$\sum x_i$:	The summation of all x_i.
ϕ :	Empty set.
N :	The set of natural numbers.
Z :	The set of integers.
$x{\leftarrow}y$:	Assign a value of y to varibale x.
$X{\leftarrow}X{\cup}\{x_1, x_2, \cdots, x_n\}$:	Add elements $x_1, x_2, \cdots, x_n$ to set X.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Background and Motivation

Computer software (or simply software) is defined as a collection of data or computer instructions that tell the computer how to work. Software is generally divided into system software, application software, and middleware between them. In computer science and software engineering, a software includes all information processed by computer systems, programs and data [1].

With the development of economic globalization, the software industry has also ushered in its best development opportunities [2]. With the reorganization of international production factors and the transfer of industries, the international software industry is undergoing a global transformation of production regions [3, 4]. As the cradle of the early software industry and the world's software power, United States of America is a global leader in software product development and basic research [5]. However, in recent years, the Asian software market has sprung up. Due to its obvious production cost advantage and broad development prospects, it is mainly represented by China, India, and Japan, occupying the largest share of global software outsourcing [5, 6]. As a result, businesses are more than ever willing to invest in bespoke software development. Furthermore, the centre of global software development is slowly shifting to Asia [7, 8].

In the context of the rapid development of Asian software industry, the scale of software is getting larger and larger, and the complexity is getting higher and higher [9]. Therefore, ensuring software quality has become an urgent research issue. Software quality is very important, especially for commercial and system software like Microsoft Office, Microsoft Windows, and Linux, furthermore, if a software is faulty (buggy), it can delete a person's work, crash the computer and do other unexpected things [1]. An example of a programming error that led to multiple deaths is discussed in Dr. Leveson's paper [10]. There are other more serious incidents caused by software failure.

The Internet service company-Yahoo reported two major user account data leakage incidents during the second half of 2016 [11]. The first announced breach was reported in September 2016, which had occurred sometime in late 2014 and affected over 500 million Yahoo user accounts [11, 12, 13]. In May 2015, Airbus issued an alert for urgently checking its A400M aircraft when a report detected a software bug that had caused a fatal crash earlier in Spain [14]. Prior to this alert, a test flight in Seville had caused the death of four air force crew members and two were left injured [14, 15].

Therefore, ensuring software quality is so important that testing the software before its release is essential. Software testing is very important because of the following reasons [16, 17]:

- Software testing is really required to point out the defects and errors that were made during the development phases.
- It is essential since it makes sure that the customer finds the organization reliable and their satisfaction in the application is maintained.
- Quality product delivered to the customers helps in gaining their confidence.
- In order to provide convenience to customers, such as delivering high-quality products or software applications, they need to be tested with lower maintenance costs to obtain more accurate, consistent and reliable results.
- Testing is required for an effective performance of software application or product.
- It is important to ensure that the application should not result into any failures because it can be very expensive in the future or in the later stages of the development.

Software testing is to find bugs, defects or errors in a software program and the purpose is to verify whether the software meets the required specifications [18]. Therefore, many researchers paying more and more attention to software testing. Broadly speaking, there are four levels of testing: unit testing, integration testing, system testing and acceptance testing [19]. Among them, unit testing refers to the tests that verify the functionality of a specific section of code. Integration testing is a type of software testing that aims to verify the interface between components based on the software design. System testing will test the fully integrated system to verify if the system meets its requirements. Acceptance testing's purpose is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery. Currently, many testing approaches, such as black box testing, structural testing, gray box testing, etc., have been developed for these four levels of testing. For instance, black box testing [20, 21] and structural testing [22] can be applied to all these four levels of software testing, gray box testing [23] is usually used in the level of integration testing, alpha and beta testing [24] are often used in the level of acceptance testing. Among these approaches, structural testing is the most commonly used approach, which is defined as the testing of a software solution's internal structure, design, and coding. The main focus of this testing method is on investigating the internal logic and structure of source code [25, 26]. In structural testing, the code is visible to the tester, moreover, it focuses primarily on verifying the flow of inputs and outputs through the application, improving design and usability, and strengthening security [27]. It is most important and necessary for a tester to have complete knowledge of source code in using the test technique. Next, the structural testing is described in detail below.

A) Structural Testing

Structural Testing is an approach of testing to test the internal structure of a program. It is also known as White Box testing or Glass Box testing [28, 29]. This type of testing requires knowledge of the code, and most of it is done by the developers. It is more concerned with how the system works, rather than the function of the system. It provides more coverage for testing. For example, to test certain error messages in an application, we need to test the trigger conditions for them, but there must be many trigger conditions. When testing the requirements drafted in the software requirements specification, one may be missed. However, with structural testing, since it aims to cover all nodes and paths in the code structure, it is most likely to cover all trigger conditions.

Structural Testing is complementary to functional testing. Using this technique the test data drafted according to system requirements can be first analyzed and then more test data can be added to increase the coverage. While it can be used on different levels such as unit testing, component testing, integration testing, system testing, it is usually done at the unit testing. It helps in performing a thorough testing on software and it is mostly automated.

There are different criteria that are used to perform such testing. Before moving on to those, it is important to understand some terms.

Control Flow Graph: It is a graphical representation of the program depicting all the paths that may be transverse during the execution. It contains,

(1) Basic Block or Node: It is a sequence of statements such that control can only have one entry point and can exit the block only when all the statements have been executed.

(2) Edge: It shows the control flow.

For example, Figure 1.1 is a control flow graph. N_1, N_2, N_3, N_4, N_5 and N_6 represents Basic Blocks or Nodes and E_1, E_2, E_3, E_4, E_5 and N_6 represents Edges. We can also observe that N_2, N_3, N_4, N_5 and N_6 together represent an IF structure.

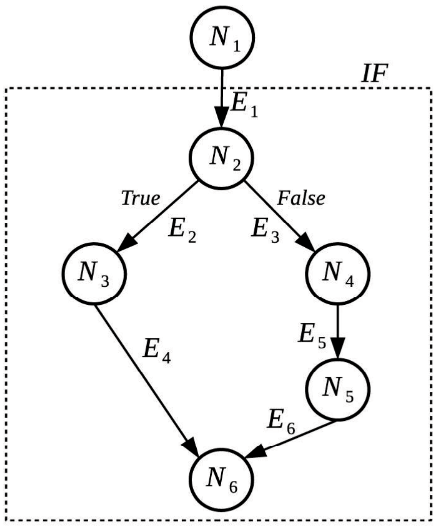


Figure 1.1: An example of control flow graph.

(3) Adequacy Criterion: This term refers to the coverage provided by a test suite. Such as, Statement Coverage (It aims to test all the statements present in the program.), Branch Coverage (It aims to test all the branches or edges at least once in the test suite or to test each branch from a decision point at least once.), Condition Coverage (It aims to test individual conditions with possible different combination of boolean input for the expression.) and Path Coverage (It aims to test the different path from entry to the exit of the program, which is a combination of different decisions taken in the sequence.).

As a general software testing method, Structural Testing has some advantages over other methods, such as it can (a) force test developer to reason carefully about implementation; (b) reveal errors in "hidden" code; (c) spot the dead code or other issues with respect to best programming practices.

In general, there are four methods developed in Structural Testing, such as path testing, data flow testing, slice-based testing and mutation testing. Among them, path testing is most commonly used, which searches suitable test data that covers every possible path in the software under test. A program may contain an infinite number of paths when the program has loops [30]. Although a loop can be tested by executing it only once [31], the number of paths in a program is exponential to the number of branches in it [32]. Moreover, the number of test data is too large, since each path can be covered by several test data [33]. For these reasons, test data generation is much more complicated, so the test data generation problem in path testing is an NP-complete problem [33, 34, 35]. Other types of structural testing methods all have to find test data to cover the paths and hence they face the same problems as path testing [36, 37, 38]. Consequently, the problem of test data generation is an NP-complete problem.

Although the problem of test data generation in Structural Testing cannot be completely solved due to its NP-completeness, many techniques have been developed to generate test data, such as (1) random testing technique [39]; (2) intelligent optimization algorithm [40]; (3) model-based testing method [41]; and (4) search-based testing method [42]; To decrease the computation time, many techniques have also been developed to decompose a system model into suitable subsystem models [43, 44]. This dissertation tries to propose a new method to approach test data generation problem by using the method of structural testing. Our method is to apply program nets to test data generation, which not only includes data flow and control flow, but also directly simulates paths' execution of a given program. It includes the benefits of the four structural testing methods and is expected to solve a variety of other problems in software testing.

B) Program Nets

A data-flow computer is based on the principle of data-driven processing and runs data-flow program nets (program net or net for short) as machine codes, which explicitly express data dependency existing in a program in graph form so that parallelism can be maximally exploited and hence high speed computation may be realized if message communication overhead is manageable [45].

In the past decade, experimental data-flow computers have been investigated: Dennis [46, 47] and Rumbaugh [48] proposed data-flow architectures, Gurd and Watson [49] and Terada and Nishikawa [50] designed experimental machines, and the ETL group of Japan [51, 52], headed by Shimada and Yuba, constructed a prototype machine SIGMA-1, which is currently under run-tests for performance evaluation and developing language DFCII and its compiler [53]. With increasing maturity of data-flow computer technology, modelling and analysis of a data-flow program net have become ever more important.

A data-flow program net of Rodriguez [54], Dennis [46, 47] and Davis and Keller [55], which is radically different from a flow chart, is a variation of the Petri nets of Petri [56], especially tuned to flows of data through arithmetic and logical operations. It can be seen as a simple case of GAN and GERT of Elmaphraby [57] and Pritsker and Happ [58] in operations research.

Graphs can capture complex data dependencies and be widely used to model and analyze data-flow programs [59]. As a kind of graphs, data-flow program net [60] is an important way to study data-flow program.

In a data-flow program net, a node represents an operation of a fixed single valued function. An edge represents a transmission channel of tokens between nodes and serves also as a FIFO queue of holding tokens. A token represents a single datum [61]. An execution of a program is expressed by a flow of tokens through the net where tokens are transferred across a node from input edges to output edges by node firing, which takes tokens from its input edges and places tokens onto its output edges. A data-flow program net has an equivalent transformation into a Petri net [56]. Program net is radically different from the data-flow chart and specially tuned to data-flows through arithmetic and logical operations. A program net can be expressed by a directed graph, as shown in Figure 1.2.

Treating a program as a program net, we try to apply program nets to software testing problem theoretically in such a way that, (i) to divide a program net into subnets, of which each can be entirely executed when some specific input data is given; (ii) for each subnet, to find out input data through analysis of the structure of the subnet. So that all the paths of a program can be tested if executing it under the obtained input data in turns.

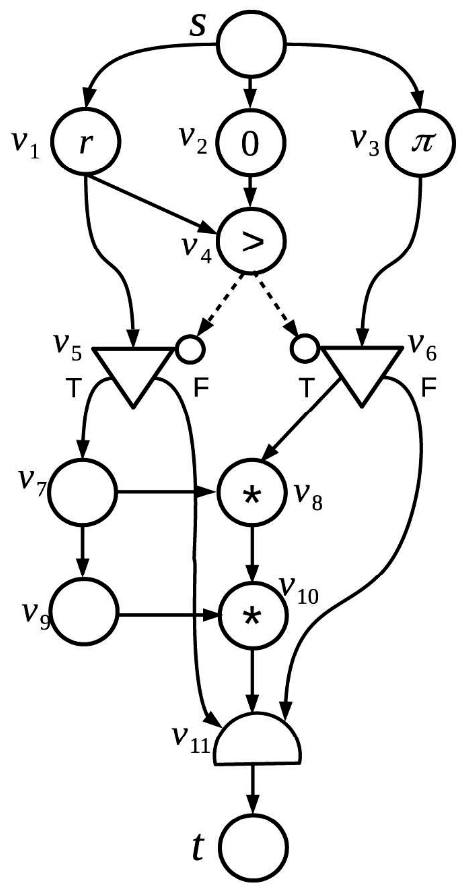


Figure 1.2: An example of program net to calculate the area πr^2 of circle: r is the radius of circle, s is the start node and t is the end node.

1.2 Dissertation Overview

The rest of this dissertation is organized as follows.

Chapter 2 starts with a review on the definitions of Program Nets. The chapter then moves on introducing some properties about Program Nets.

Chapter 3 presents a method of generating subnets for a given program net in order to finally find a set of test data used in software testing. A subnet will be such that all the nodes can fire for certain given input test data, which means the test data can cover the subnet. Owing to generating subnets, acyclic program net are constructed and then a series of polynomial algorithms are proposed to generate subnets based on acyclic program net.

Chapter 4 presents a method to find a specific test input data for such subnets that may possess input data based on the analysis result of the behaviours of all the subnets. Firstly, a brief introduction is given to Satisfiability Modulo Theories (SMT for short) that can be used to find test input data. Then, discussion is done on how to equivalently transform the subnets obtained in Chapter 3 into exhaustive subnets. After that, an algorithm is designed to obtain all constraint conditions from a given exhaustive subnet. Finally, we introduce an SMT solver, $Z3$ Prover, which can find the specific test input data according to the constraint conditions.

Chapter 5 gives an example to show how to find subnets by using the proposed algorithms, as well as to indicate how to find the input test data of the program net in order to do the related software testing.

Finally, Chapter 6 summarizes this dissertation's works and shows our future works on solving software testing problem by using program nets.

Chapter 2

Program Nets and Software Testing

This chapter presents the basic definitions of General Program Nets, such as three types of nodes in program nets, and their firing rules. Simultaneously, in order to describe the dynamic behavior information of program nets, the definition of exhaustive program network as a extension of general program nets is also introduced. Then the basic properties of program nets used in this dissertation are also given, such as self-cleanness. In addition, the problems of software testing and the whole process of applying program nets to software testing will also be introduced.

2.1 Definitions and Properties of Program Nets

2.1.1 General Program Nets

A program net [62] is denoted by $PN{=}(V, E, \alpha, \beta)$, where V is a set of nodes consisting of AND-node (◯), OR-node (△ or a semicircle) and SWITCH-node (°▽). The three types of nodes are shown in Table 2.1. E is a set of directed edges consisting of data-flow edge and control-flow edge written in solid arrow ($\longrightarrow$) and dotted arrow ($\dashrightarrow$) respectively. The token (●) represents a single datum and token distribution $d^\tau{=}(d^\tau_{e_1}, d^\tau_{e_2}, \cdots, d^\tau_{e_{|E|}})$ expresses token numbers on each edge e_i at time τ. α is token threshold of node firing on the input edges, β is token threshold of node firing on the output edges.

Table 2.1: Three Node types and their shapes, where n_{e_i} is the number of input edges.

Node types		Shapes
AND-node		○
OR-node	$n_{e_i}{=}2$	△
	$n_{e_i}{>}2$	⌓
SWITCH-node		°▽

A node v_i represents an operation specified by a fixed single-valued function of data (called tokens) on the input edges entering into v_i. An edge e_i represents a transmission channel of data and serves as a FIFO queue. A SWITCH-node is always accompanied by a control-flow edge while others are not.

If a program net PN is given with an initial token distribution d^0 onto edges, then it is called marked program net and denoted by $MPN{=}(PN, d^0)$.

Three types of nodes and their firing rules are categorized into following classes:

(i) AND-node: The class of AND-nodes includes start, termination, data-base, logical-operation, predicate-evaluation, basic arithmetic-operation including vector-operation, duplication, synchronization, buffer, joint and distributor. An AND-node becomes firable if each input edge x possesses no less tokens than the individually prescribed number $\alpha_x{>}0$. If a firable node is fired, α_x tokens are removed from each input edge x and $\beta_y{>}0$ tokens are placed on each output edge y. Especially, start node s fires only once.

(ii) OR-node: An OR-node represents a merging of two token flows and becomes firable if one of its input edge x possesses more tokens than the prescribed number α_x. In general data-flow program graphs [46, 47, 55], an OR-node is always accompanied by a control-flow edge while it is not in this paper. Hence an OR-node is less restricted here. An OR-node is fired, $\alpha_x{>}0$ tokens are removed from an arbitrarily selected input edge x but not from the other and $\beta_y{>}0$ tokens are placed on the output edge y.

(iii) SWITCH-node: A SWITCH-node represents an decision condition clause and becomes firable if the data input and control input have at least one token on each. 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. We actually have $\alpha{\equiv}\beta{\equiv}1$.

The firing rules just mentioned are summarized ($\alpha{\equiv}\beta{\equiv}1$ for simplicity) in Figure 2.1. Note that a node without input edges is always firable. For simplicity we choose to use as the net elements only those tabulated, since a node with multiple inputs or multiple outputs can be constructed from those tabulated.

The program nets dealt with in this dissertation are based on the assumptions that (1) there is only one start node s (without input edge), and one termination node t (without output edge), and (2) for any node v_i, there exists at least one directed path from start node s to termination node t that passes v_i.

[**Definition 2.1**] A node sequence $\sigma{=}v_1v_2\cdots v_k$ of length k is called a firing sequence of length k with respect to a given token distribution d^0 on edges if:

(i) the first node v_1 is the start node s which appears only once in σ.

(ii) node $v_i(i = 1, 2, \cdots, k)$ is firable with respect to token distribution d^{i-1}, where d^i is the token distribution resulted from d^{i-1} by firing node v_i. □

[**Definition 2.2**] Let $MPN{=}(PN, d^0)$ be a marked program net. A finite firing sequence σ of MPN is called terminating if it satisfies the criterion that: any node x is not firable with respect to the final token distribution d^k. □

[**Definition 2.3**] A marked program net MPN is called terminating if and only if (iff for short) all of its firing sequences are of finite length $k{<}\infty$. □

[**Definition 2.4**] A program net PN is d^0-terminating iff $MPN{=}(PN, d^0)$ is terminating. PN is called structurally terminating if and only if for any initial token distribution d, PN is d-terminating. □

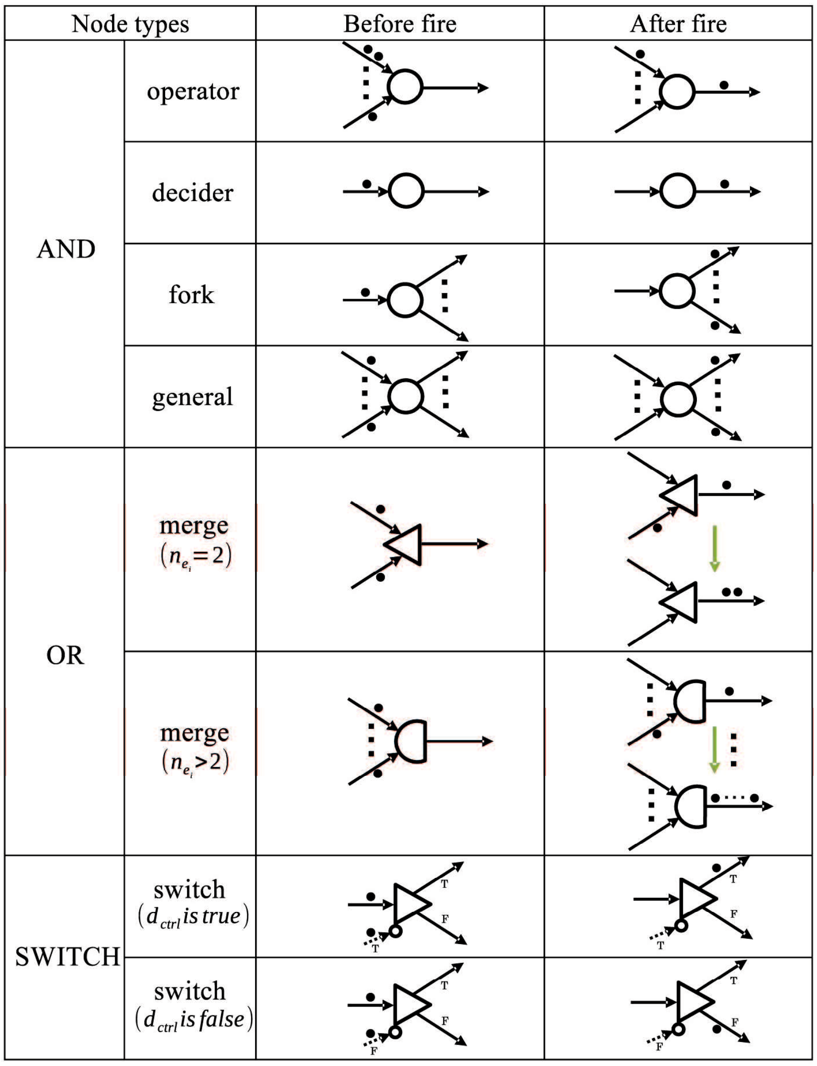


Figure 2.1: Node types and firing rules, where n_{e_i} is the number of input edges and d_{ctrl} is the token value of control-flow.

[**Definition 2.5**] An AND-node-path is a directed path consisting of edges and AND-nodes. The most simple AND-node-path consists of a single edge or a single AND-node. □

[**Definition 2.6**] PN is called SWITCH-less or OR-less iff there is no SWITCH-node or OR-node respectively. □

[**Definition 2.7**] Let v be a node, e_{ip} and e_{op} be its data-flow input edges and output edges respectively. If the required operand of node v is satisfied, v can fire, that is to say v is firable.

(i) An AND-node v_{AND} is firable with respect to d^{τ} if its 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}. Especially, 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 $d^{\tau}_{e_{ip}} \geq \alpha_{e_{ip}}$. 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 2.8**] 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. □

2.1.2 Exhaustive Program Nets

In the general program net, 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; (c) duplicating its input data. Logical and arithmetic operators are generally those included in the list, $O{=}(<, >, \leq, \geq, !, !{=}, \&\&, ||, \&, |, \hat{}, =, +, -, *, /, \%, \ll, \gg, ++, --)$, as shown in Table 2.2. This operator list can be further divided into logical and arithmetic operator sublists, $O_l{=}(<, >, \leq, \geq, ==, !, !{=}, \&\&, ||)$ and $O_a{=}(\&, |, \hat{}, =, +, -, *, /, \%, \ll, \gg, ++, --)$ respectively.

[**Definition 2.9**] A program net is called Exhaustive Program Net and denoted by $EN{=}(V, E, g, o, r, \alpha, \beta)$ if each AND-node possesses at most two input edges and further the following conditions are satisfied:

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

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

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

Table 2.2: 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	$==$	equal to
5	!	logical not
6	!=	not equal to
7	&&	logical and
8	$\|\|$	logical or
9	&	bitwise and
10	$\|$	bitwise or
11	ˆ	bitwise xor
12	$=$	assignment
13	$+$	addition
14	$-$	subtraction
15	$*$	multiplication
16	$/$	division
17	%	modulus
18	$\ll$	left shift
19	$\gg$	right shift
20	$++$	increment
21	$--$	decrement

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

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

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

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

Suppose $e=(v', v)$ and $g(v')$ be 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. □

Figure 2.2 shows the exhaustive program net of Figure 1.2.

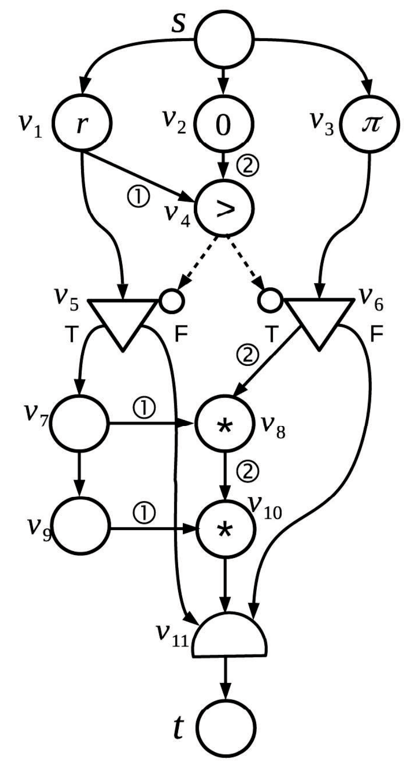


Figure 2.2: The exhaustive program net *EN* of Figure 1.2.

2.1.3 Basic Properties of Program Nets

In this dissertation, we assume that all marked program nets are self-cleaning net with d^0=$\mathbf{0}$. PN is called acyclic if there is no directed circuit. If there exists a directed circuit consisting of AND-nodes and OR-nodes only (or AND-nodes and SWITCH-nodes only, or AND-nodes only), then token will never go out of the circuit once a token entered the circuit (or never enter the circuit), which implies tokens will remain inside the circuit (or at the entrances of the circuit) [63].

Figure 2.3 (a) shows a circuit which contains AND-node and OR-node only, Figure 2.3 (b) shows a circuit which contains AND-node only, Figure 2.3 (c) shows a circuit which contains AND-node and SWITCH-node only and Figure 2.3 (d) shows a circuit which contains OR-node and SWITCH-node. The following propositions are satisfied [63].

[**Proposition 1**] A self-cleaning SWITCH-less program net contains no directed circuits. □

[**Proposition 2**] A self-cleaning OR-less program net contains no directed circuits. □

[**Proposition 3**] Each directed circuit of a self-cleaning program net contains at least one OR-node and one SWITCH-node. □

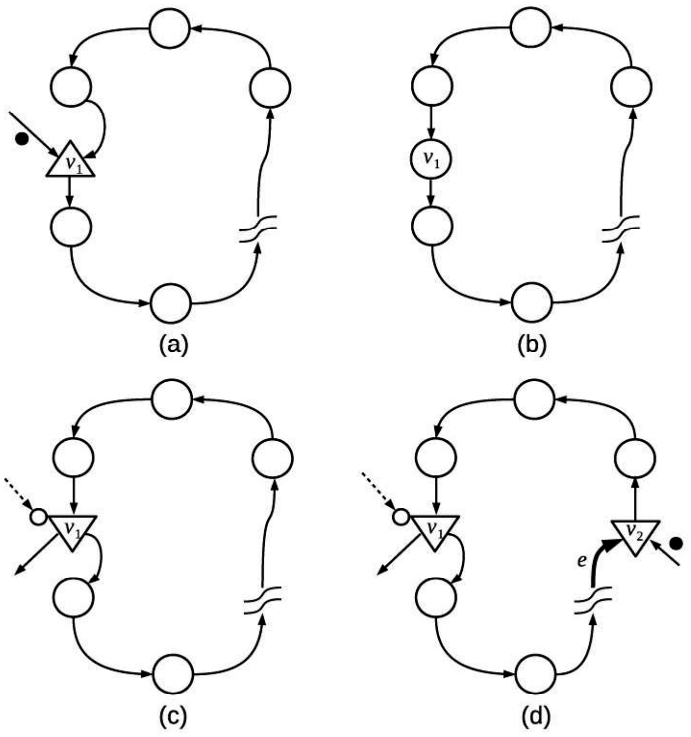


Figure 2.3: Four types of directed circuit.

[**Definition 2.10**] 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 *predecessor* and *successor* of node v are denoted by $Pre(v)$ and $Suc(v)$, respectively.

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

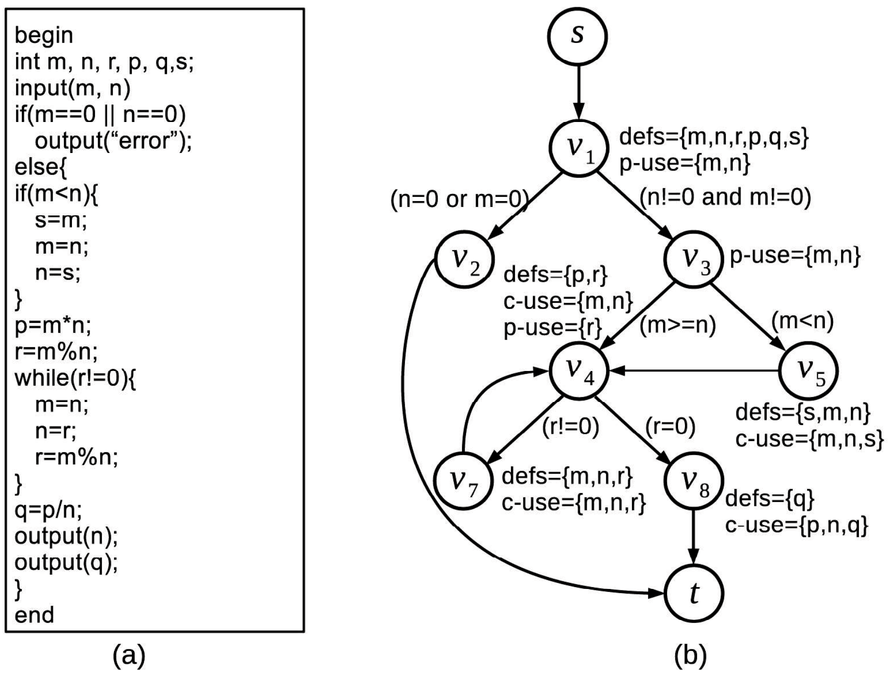
(iii) If a directed circuit includes $e=(v_1, v_2) \in E$ and v_2 is an OR-node, e is called *back edge* (i.e. e in Figure 2.3 (d)). □

2.2 Software Testing

2.2.1 Data Flow Chart

As described in Introduction, structural testing is widely adopted as a type of software testing in reality. Because program graphs can visually view structure of a program, researchers usually convert a program under test into a program graph in structural testing, and then directly test the program graph.

Data flow testing method can be considered as a structural testing technique, which uses program graphs to represent programs under test. A program graph is usually called a Data Flow Chart (DFC for short) [64]. DFC not only contains control-flow but also data-flow. Figure 2.4 (a) is an example program that calculates Greatest Common Divisor and Lowest Common Multiple. Figure 2.4 (b) shows a DFC of the example program.



```
begin
int m, n, r, p, q,s;
input(m, n)
if(m==0 || n==0)
  output("error");
else{
if(m<n){
  s=m;
  m=n;
  n=s;
}
p=m*n;
r=m%n;
while(r!=0){
  m=n;
  n=r;
  r=m%n;
}
q=p/n;
output(n);
output(q);
}
end
```

Figure 2.4: A program and transfered to data-flow chart.

The nodes in DFC possess the information of variable definition ("def" for short) and reference (denoted by "use") except start node and end node. There are two major types of use nodes as follows:

- P-use: represents predicate use and this variable is used when making a decision (e.g. if b>6).

- C-use: represents computation use and this variable is used in a computation (e.g. b=3+d with respect to the variable d).

In software testing, a testing path of DFC is a directed path that represents the interaction between variable definition and the reference in a program. Therefore, we have the following Definition.

[**Definition 2.11**] A path p=<$v_1, v_2, \cdots, v_j, v_k$> in DFC is a DU-path with respect to a variable v, if v is defined at node v_1 and either:

(i) there is a c-use of v at node v_k and v has no variable re-definition at any other node in p, or

(ii) there is a p-use of v at edge (v_j, v_k) and v has no variable re-definition at any other node in p with no loop. □

In data flow testing, test data have to be generated according to the test adequacy criterion, which is considered to be a stopping rule that determines whether sufficient testing has been done and provides measurements of test quality. Several dataflow-based test adequacy criteria have been proposed, the most popular family of test adequacy criterion is Rapps and Weyuker's test adequacy criteria system architecture [65], which is shown in Figure 2.5.

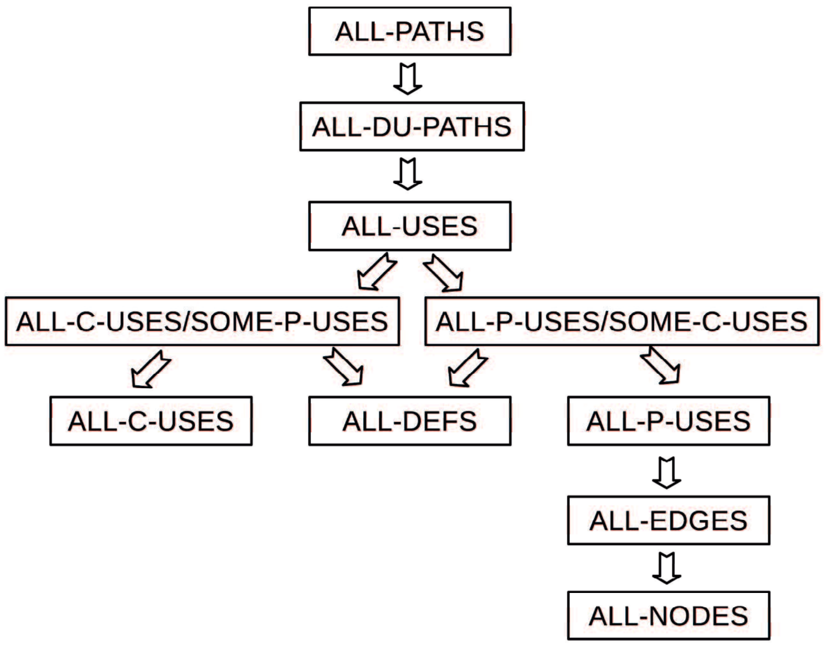


Figure 2.5: Rapps and Weyuker's test adequacy criteria system architecture [65].

The test adequacy criteria system architecture is clearly derived from a set of criteria defined for the control flow graphs introduced in Chapter 1 and originates from ALL-PATHS, ALL-EDGES and ALL-NODES criteria. Rapps and Weyuker extended the set of criteria by defining ALL-DEFS, ALL-USES, ALL-C-USES/SOME-P-USES, ALL-P-USES/SOME-C-USES, ALL-P-USES and finally All-DU-PATHS. Single ALL-C-USES (not including SOME-P-USES) criterion was added later on in the list of the after mentioned criteria. The whole set is based on definitions and uses of variables and uses are distinguished by two terms, c-uses and p-uses. The first term is used to define a use in a computation (e.g. the right side $x+y$ of assignment $z=x+y$) and the second to define the use of the variable as a predicate in a Boolean calculation. Data flow criteria were examined by different researchers from time to time, aiming at defining a partial order between all the criteria or revealing the weaknesses and strengths of each criterion. Rapps and Weyuker also provided the "hierarchy" of the criteria with a robust proof in Figure 2.5 and have defined their strongest criterion, ALL-DU-PATHS [66].

2.2.2 Software Testing Problem

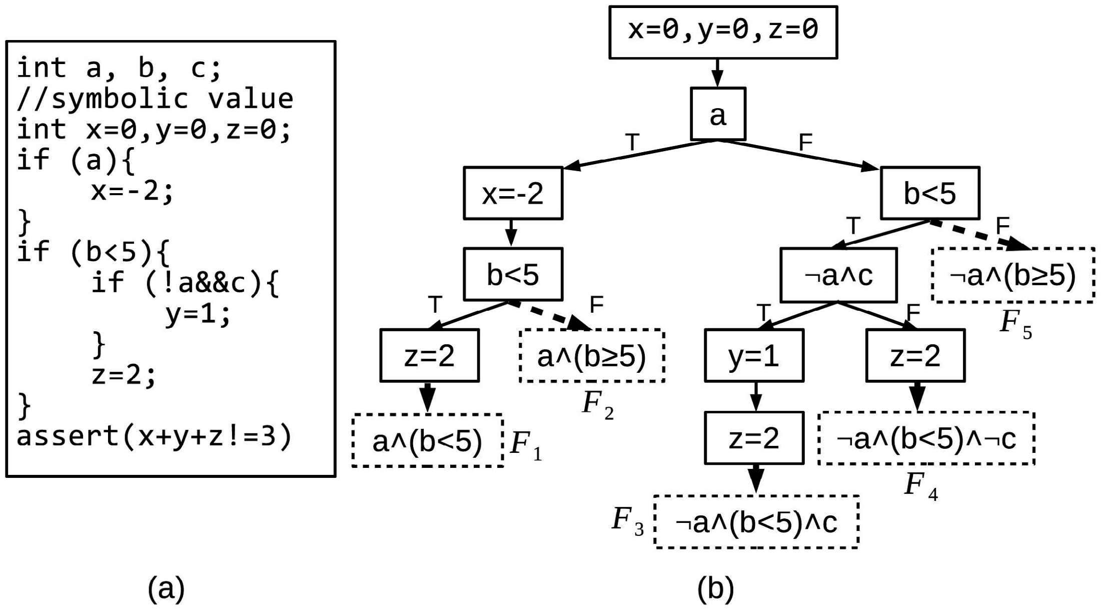
One of software testing problems is to find a set of test data that can cover all the DU-paths. This means generating DU-paths is an important problem in software testing. In fact, if a DFC contains loop, then the number of DU-paths would become infinite. The loop problem can be dealt by executing it once in software testing [31], but the number of paths in a program is exponential to the number of branches in it, which makes the problem of testing path generation much complex. In Figure 2.4, variable m is defined at v_1, v_5, v_7, then DU-paths set of m is $P(m)=\{<v_1, v_2>,<v_1, v_3>,<v_1, v_3, v_4>, <v_1, v_3, v_5>,<v_5, v_4>,<v_5, v_4, v_7>,<v_7, v_4>\}$. In fact $<v_1, v_3>$ and $<v_5, v_4>$ are included in other paths, and hence only $P'(m)=\{<v_1, v_2>, <v_1, v_3, v_4>, <v_1, v_3, v_5>, <v_5, v_4, v_7>, <v_7, v_4>\}$ need to be checked in software testing.

In this dissertation, instead of DU-paths, we are to generate a set of subnets for a given program net. The reasons are that (i) a program net is equivalent to a data-flow chart in the sense of expressing the data-flow of a program and each DU-path of a data-flow chart is also included in the program net; (ii) if a set of subnets cover all nodes as well as T and F terminals of all SWITCH-nodes then all the DU-paths to be checked can be investigated by individually giving input data for each subnet; (iii) finding a set of input test data for a subnet is rather easier than that for a whole program net and hence the total computational time can be decreased.

2.3 Symbolic Execution

In computer science, symbolic execution is a means of analyzing a program to determine which input data can make each part of the program executable. Symbolic execution technique was originally proposed by King in Reference [67] in 1976, and has attracted people's interest since then. The main reasons are that constraint solver on which the symbolic execution technique depends has been improved and the performance of computers has also been improved.

Symbolic execution is a well-known program analysis technique that uses symbolic values (such as a, b and c in Figure 2.6) instead of concrete data when initializing program variables, and also it expresses logical expressions with symbolic values [68]. As a result, the output calculated by a program is expressed as an expression composed of symbol values. A symbolic execution tree represents execution paths followed during the symbolic execution of a program, in which nodes represent program states and arcs represent transitions between states [69]. Figure 2.6 shows a program fragment and its symbolic execution tree. F_1~F_5 are boolean formulas over the symbolic values of input variables.



```
int a, b, c;
//symbolic value
int x=0,y=0,z=0;
if (a){
    x=-2;
}
if (b<5){
    if (!a&&c){
        y=1;
    }
    z=2;
}
assert(x+y+z!=3)
```

Figure 2.6: A program fragment and its symbolic execution tree.

2.3.1 SMT Problem Description

The earliest development of Satisfiability Modulus Theory (SMT for short) can be traced back to Nielsen's doctoral dissertation [70] from the late 1970s to the early 1980s. In the past few years, we have witnessed important improvements in this technology, and SMT is still very popular even now. It is used in various fields, such as hardware design, software verification, model checking, symbolic execution, test case generation, etc.

The SMT issue is one of the central issues of theoretical and practical interest in computer science, which is a problem to determine whether the formula expressing constraint conditions has a solution [71].

A formula ϕ_i $(i{\in}N)$ can be a propositional variable p, a negation $\neg\phi_0$, a conjunction $\phi_0{\wedge}\phi_1$, a disjunction $\phi_0{\vee}\phi_1$, or a logical implication $\phi_0{\Rightarrow}\phi_1$. When there is only one letter, it is either a propositional variable p or its negation $\neg p$. Then the negation of a letter p is $\neg p$, and the negation of $\neg p$ is just p. $\phi_1{\wedge}\cdots{\wedge}\phi_n$ is a conjunctive normal form, where each element ϕ_i $(1{\leq}i{\leq}n)$ is a formula, and this conjunctive normal form is true or satisfiable only if every element is true. $\phi_1{\vee}\cdots{\vee}\phi_m$ is a disjunction normal form, in which each element ϕ_i $(1{\leq}i{\leq}m)$ is a formula, and this disjunction normal form is true or satisfiable as long as one of all elements is true. For convenience, we call disjunction normal form and disjunction normal form constraint conditions in this dissertation. A normal form has solution if and only if the normal form is true, and therefore a normal form may have many solutions, and we only need one of them.

For instance, a solution of the following constraint conditions is a set of variable values, e.g. $(x, y, z){=}(1, 2, 1)$, which makes the constraint conditions satisfiable.

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

2.3.2 SMT Solvers

An SMT solver is a powerful type of automatic theorem prover that can solve SMT problems. In addition, it has been used in a variety of other application domains including verification of deep neural networks, program synthesis, static analysis, scheduling, etc. [72, 73].

To efficiently solve complex real world problems, many state-of-the-art SMT solvers such as $Z3$ [74], Yices [75], CVC4 [76], MathSAT5 [77], Boolector [78], are developed. These SMT solvers have been developed using different methods including manual heuristic combination methods, satisfiability search techniques, or other methods.

Table 2.3 summarizes some of features of several available SMT solvers. The column "SMT-LIB" indicates compatibility with SMT-LIB language. Many systems marked "yes" may support only older versions of SMT-LIB, or offer only partial support for SMT-LIB language.

Table 2.3: SMT solvers.

Platform			Features			
Name	OS	License	SMT-LIB	Built in theories	API	SMT-COMP
Boolector	Linux	MIT	v1.2	bitvectors, arrays	C	2009
CVC4	Linux, Mac OS, Windows, FreeBSD	BSD	Yes	rational and integer linear arithmetic, arrays, tuples, records, inductive data types, bitvectors, strings, and equality over uninterpreted function symbols	C++	2010
MathSAT	Linux, Mac OS, Windows	Proprietary	Yes	empty theory, linear arithmetic, nonlinear arithmetic, bitvectors, arrays	C/C++, Python, Java	2010
Yices	Linux, Mac OS, Windows, FreeBSD	GPLv3	v2.0	rational and integer linear arithmetic, bitvectors, arrays, and equality over uninterpreted function symbols	C	2014
Z3	Linux, Mac OS, Windows, FreeBSD	MIT	v2.0	empty theory, linear arithmetic, nonlinear arithmetic, bitvectors, arrays, datatypes, quantifiers, strings	C/C++, .NET, OCaml, Python, Java, Haskell	2011

2.4 Overview of Applying Program Nets to Software Testing

Treating a program as a program net, we are to apply program nets to software testing problem theoretically. Figure 2.7 shows the entire process of applying a program net to address software testing problems based on the above description, in which steps (a) - (d) are to be discussed in the following chapters.

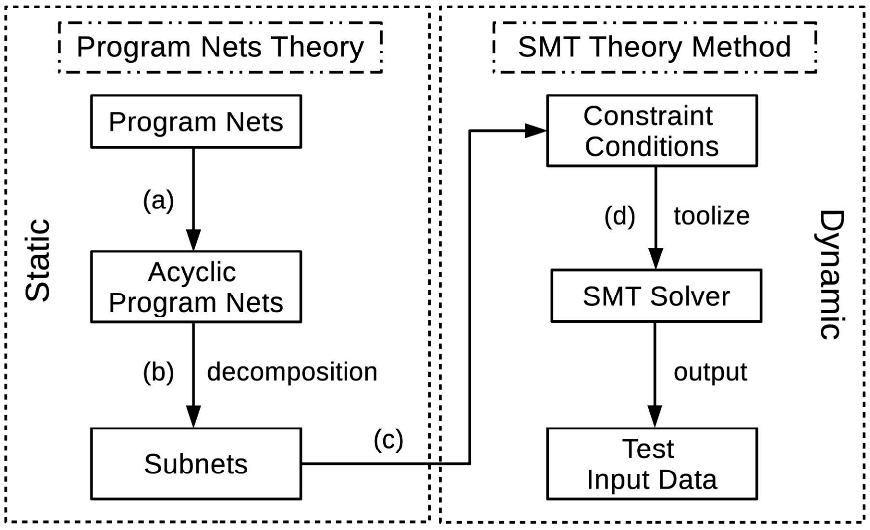


Figure 2.7: The procedures of generating test input data.

Concretely, steps (a) and (b) are approached in Chapter 3 by dividing a program net into subnets, of which each can be entirely executed when some specific input data is given. Steps (c) and (d) are dealt with in Chapter 4 by finding out input data for each subnet through analysis of its structure.

Chapter 3

Subnets Generation of Program Nets

In this chapter, we are to propose a method of generating subnets for a given program net in order to finally find a set of test data used in software testing. A subnet will be such that all the nodes can fire for certain given input test data, which means the test data can cover the subnet. Furthermore, if a set of input test data can make all the subnets firable, then this data set is what we want to find in software testing.

3.1 Basic Consideration

In data flow testing, the first important step is to generate test paths. According to All-DU-PATHS test coverage criteria, the test paths here are DU-paths introduced in Chapter 2. The general process of test path generation for a given program is shown in Figure 3.1. Specifically, it consists of (1) dividing the program into basic blocks, (2) constructing a data flow chart, and (3) obtaining all DU-paths.

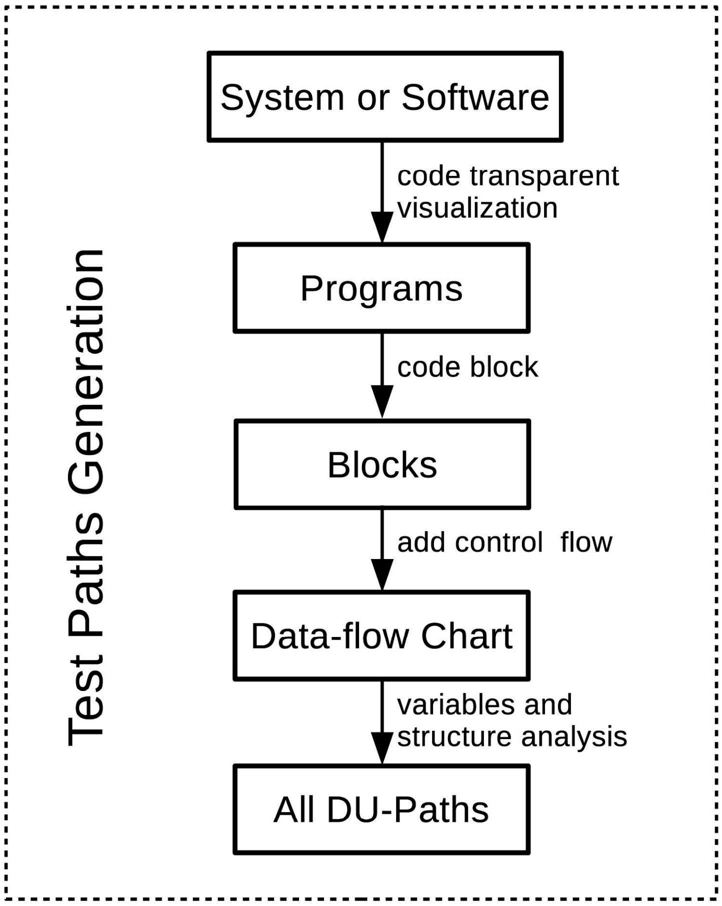


Figure 3.1: Test paths generation in data flow testing.

As mentioned in Chapter 2, if the number of branches of a program increases exponentially, it will lead to an explosive increase in the number of test paths. It will undoubtedly consume a lot of time and cost to obtain all test paths. Therefore, we hope to use a method of generating subnets from a given program net to solve the explosive growth problem of the number of test paths. Moreover, the given program net retains a program's data flow and control flow information.

As the first major step of using program nets to solve the software testing problem, the process of subnets generation is shown in Figure 3.2. The concrete steps for subnet generation are as follows:

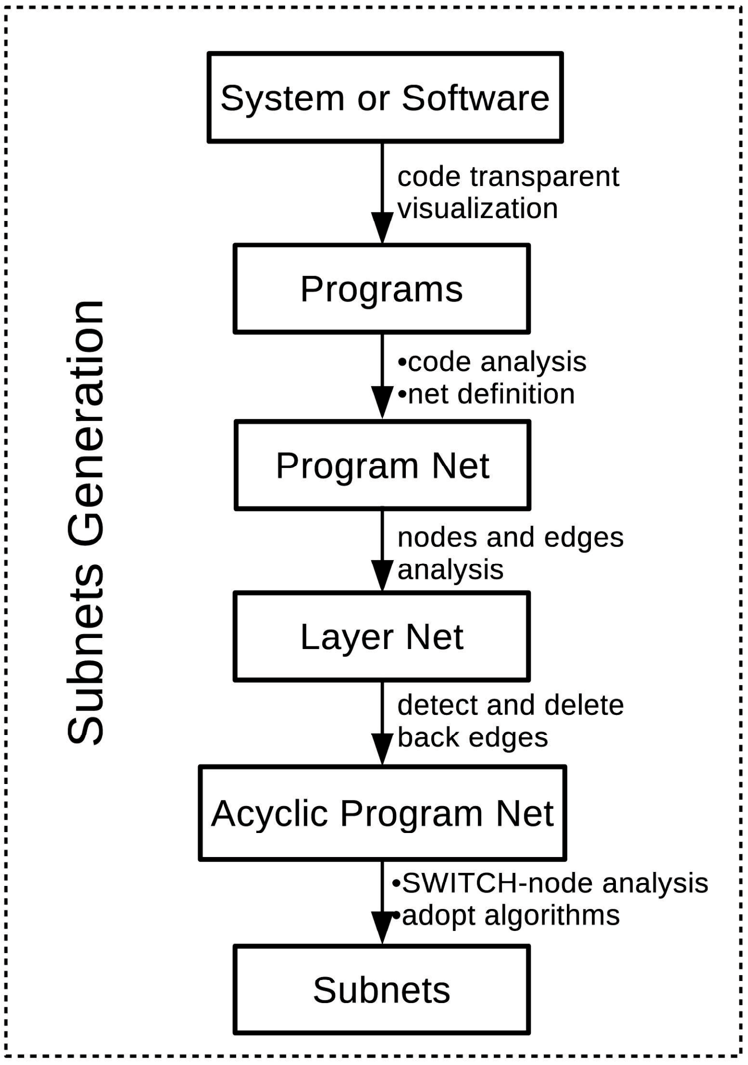


Figure 3.2: Subnets generation in program nets.

(1) To transform a program to a program net according to general definition of program nets;

(2) To construct a layer net of the program based on different firing rules and firing order of nodes, in which each layer contains one or more nodes with same firing order;

(3) To analyze the layer net to detect whether there are directed circuits in the program net, and if the program net contains directed circuit then delete some back

edges to construct an acyclic program net.

(4) For the acyclic program net, to generate an initial subnet and then generate other subnets.

In generating subnets, the following two important problems are mainly to be solved:

(i) Generally, a large scale program net may contains directed circuits. Since in software testing, any loop (directed circuit) needs only to be executed once [31], it is the first problem how to find and delete some back edges to make a given program net acyclic.

(ii) For the acyclic program net, how to divide it into a set of subnets that cover all the nodes of the original program net PN. Since each SWITCH-node of PN has the behaviors that its input tokens may flow to both its T and F terminals, the set of subnets should also include both T and F terminals of all the SWITCH-nodes.

3.2 Acyclic Program Net Construction

As mentioned in Basic Consideration, the first problem to be solved in generation of subnets is construction of an acyclic program net. In this section, we are to design algorithms to delete back edges in order to make a given program net acyclic by the following steps: (1) construct a layer net to possibly find back edges; (2) delete certain back edges to construct an acyclic program net.

3.2.1 Construction of Layer Net

A program net is a variant of petri net, it possess three types of nodes, namely AND-node, OR-node and SWITCH-node. Each type of nodes can fire under different firing rules showed in Figure 2.1. And also they have different firing order in execution of a program net. For example, in Figure 3.3, based on the firing rules of three types of nodes we know that SWITCH-node v_5 must fire after v_1 and v_4.

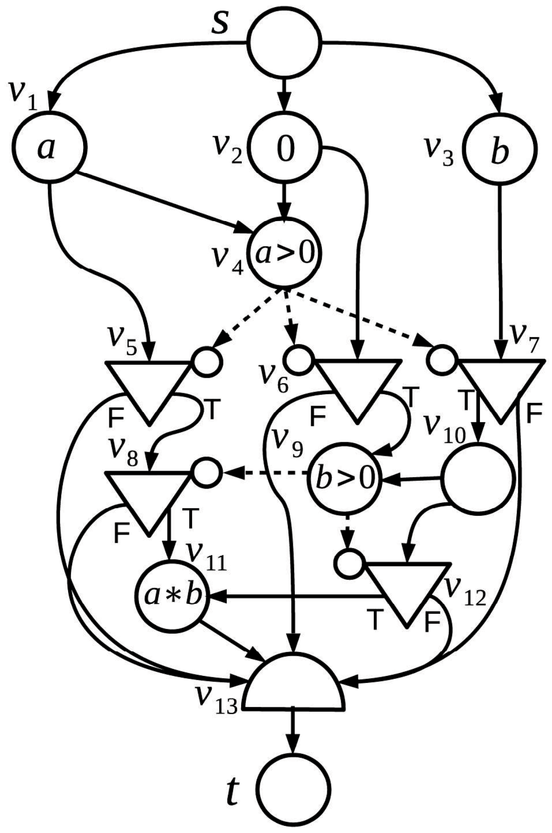


Figure 3.3: An example of program net.

In a program net, due to the number of input edges and the type of nodes, there is always a situation that firing of a node may depend on other nodes. In order to express such situation concretely, we summarized firing dependencies of three types of nodes as shown in Figure 3.4.

In Figure 3.4, assuming that nodes v_1, v_4 and v_7 have been fired and two tokens are placed on their output edges, respectively. According to the type of nodes v_3, v_6 and v_9, we can find out that v_3 should fire after v_2 firing and v_6 should fire after v_5. However, v_9 can fire directly after v_7 because of the firing rules of OR-node. Thus, we put the nodes with same firing order into a same layer. Figure 3.5 shows the layers of these nodes.

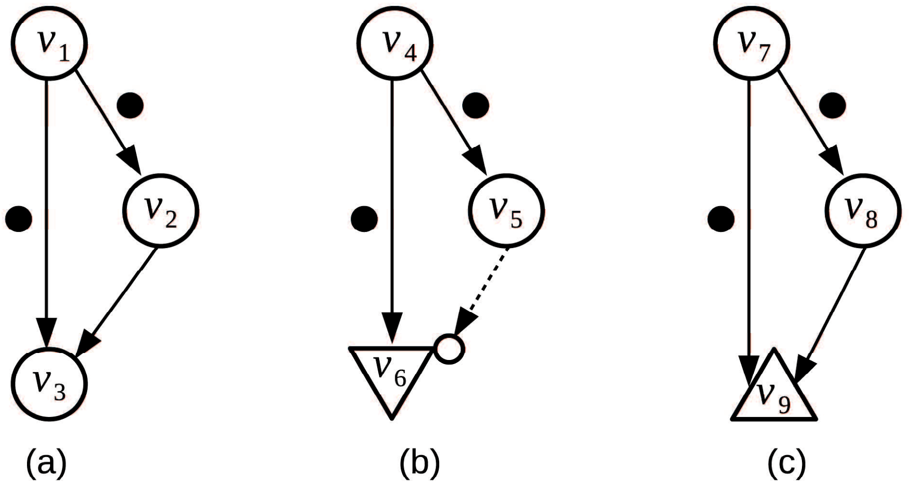


Figure 3.4: Firing dependencies of three types of nodes.

Based on the above discussion, we define layer nets in the following definition.
[**Definition 3.1**] A layer net L_{PN}=(V, E, α, β, l) is a program net that has its vertexes partitioned into a sequence of layers, and its edges connect vertexes between layers, where V is the node set, E is the edge set and l expresses the number of layers for the nodes obtained by ≪Algorithm 1 Construction of Layer Net≫. □

Algorithm 1 Construction of Layer Net

```
Require: A program net $PN = (V, E, \alpha, \beta)$
Ensure: A layer net $L_{PN} = (V, E, \alpha, \beta, l)$
1: $G \leftarrow PN$
2: queue $Q \leftarrow \phi$
3: $level = 1$
```

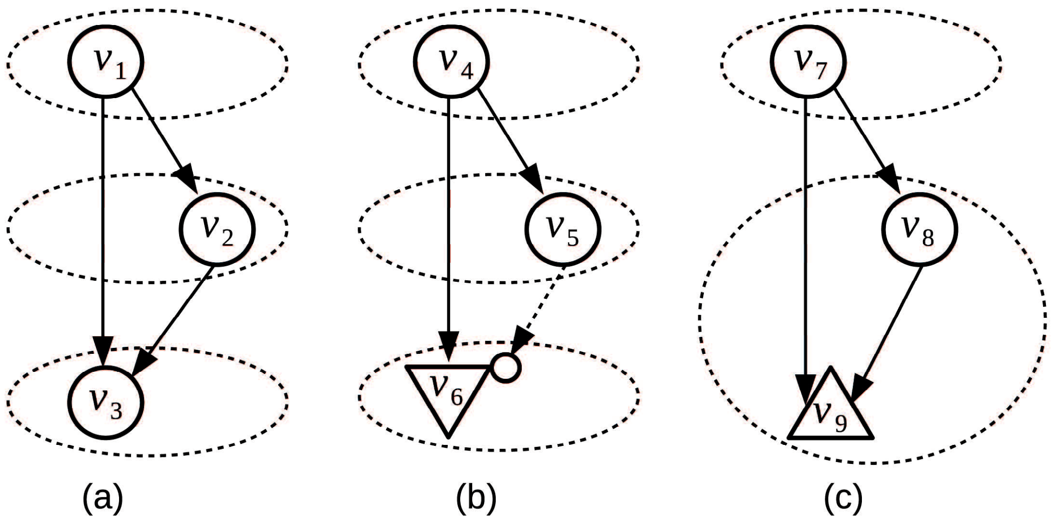


Figure 3.5: Layered diagrams of three types of nodes.

```
4: mark each $v \in V$ "unvisited"
5: let $\rho$ be a virtual node
6: enqueue $s$ (start node) and $\rho$ to $Q$
7: while true do
8:     dequeue an element $w$ from $Q$
9:     if $w = \rho$ then
10:        if $Q = \phi$ then
11:            break
12:        else if $Q \neq \phi$ then
13:            enqueue $\rho$ to $Q$
14:            do $level = level + 1$
15:        end if
16:    else if $w \neq \rho$ then
17:        do $l(w) =$ level
18:        for each "unvisited" output node $c$ of w do
19:            if $c$ is AND-node or SWITCH-node then
20:                if all input nodes of $s$ are "visited" then
21:                    enqueue $c$ to $Q$
22:                    mark $c$ "visited"
23:                end if
24:            else if $c$ is OR-node then
```

```
25:                 enqueue $c$ to $Q$
26:                 mark $c$ “visited”
27:             **end if**
28:         **end for**
29:     **end if**
30: **end while**
31: make $L_{PN}$ from $PN$ and $l$
```

Algorithm 1 can output not only layer numbers of nodes but also a layer net L_{PN} for a given program net PN and its computational time complexity is $O(|V|+|E|)$. Figure 3.6 shows a simple layer net constructed by Algorithm 1 for the program net shown in Figure 3.3.

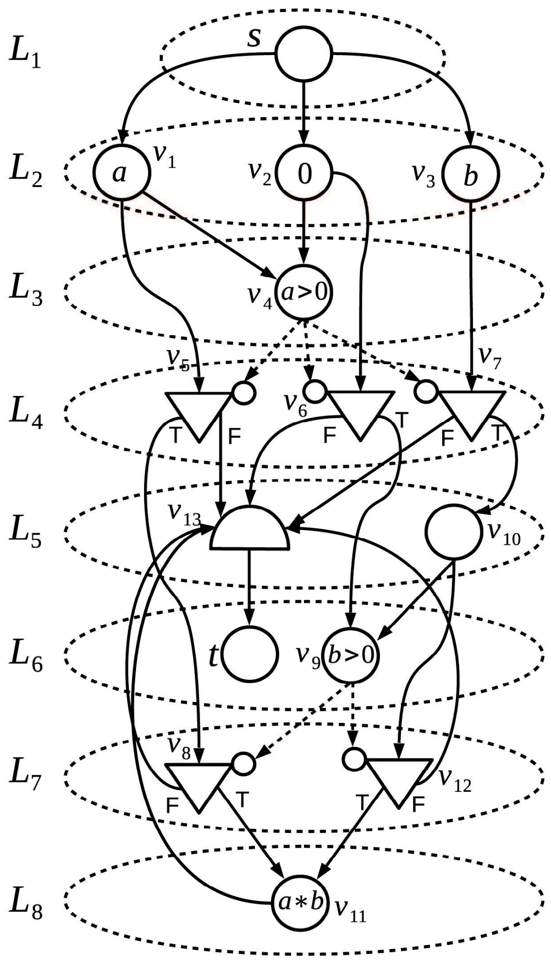


Figure 3.6: The layer net of Figure 3.3.

3.2.2 Properties of Layer Nets

In the following discussions, we use L_i to express L_{PN}'s i-th layer consisting of a set of nodes, in which each node v of L_i satisfies $l(v)=i$. Figure 3.7 is an example of layer net which contains directed circuits.

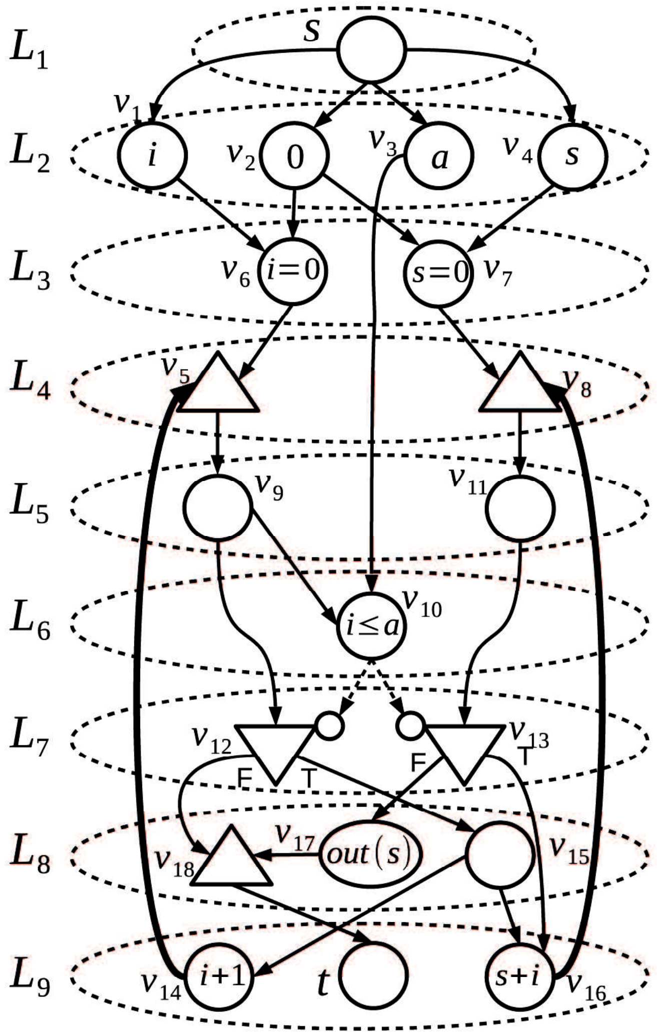


Figure 3.7: A layer net that calculates the sum of integer additions from 0 to a, where the red edges are back edges.

Proposition 4: Let (v, u) be a directed edge in L_{PN}. If $l(u) \leq l(v)$, u is an OR-node. □

Proof: Figures 3.8 (a) and (b) show the case of $l(u) \leq l(v)$, from which it is obvious that u contains at least two input edges. If u is an AND-node or a SWITCH-node, u must be included in the layer with level greater than $l(v)$ according to the lines 19∼23 in the Algorithm 1. This means $l(u) > l(v)$ and contradicts the condition "$l(u) \leq l(v)$", and thus u must be an OR-node. □

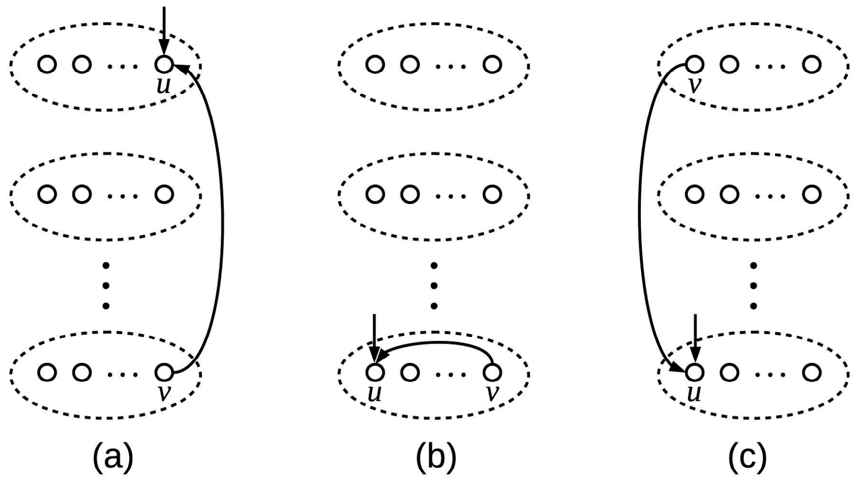


Figure 3.8: Three types of edges in layer net.

Proposition 5: Let (v, u) be a directed edge in L_{PN}. If $l(u) > l(v)+1$, u is AND-node or SWITCH-node. □

Proof: From Fig.3.8 (c), if u is an OR-node, then $l(u)=l(v)+1$ must hold according to the lines 24∼27 in the Algorithm 1. Therefore, u must be AND-node or one SWITCH-node when $l(u) > l(v)+1$. □

Proposition 6: Let $MPN = (PN, d^0)$ be a self-cleaning marked program net and L_{PN} be a layer net obtained from PN. If any two nodes $v, u \in V$ with $(v, u) \in E$ satisfy $l(u) \geq l(v)$, then there exists no directed circuit in PN. □

Proof: This proposition obviously holds if there exist no such two nodes $v, u \in V$ with $l(u)=l(v)$ and further with both $(v, u) \in E$ and $(u, v) \in E$ that compose a directed circuit. If such $v, u \in V$ exist, then both v and u must be OR-nodes from Proposition 4, which contradicts Proposition 3 (since MPN is self-cleaning). □

It is obviously that, if L_{PN} contains directed circuits, there must exist $v, u \in V$ with $(v, u) \in E$ and $l(u) < l(v)$ from Proposition 6; and such u must be OR-node from Proposition 4. That is, (v, u) is a back edge if a path from u to v exists, like the node v_5 and v_8 in the Figure 3.7, the edges (v_{14}, v_5) and (v_{16}, v_8) are back edges. Therefore, to obtain an acyclic net L_{PN}, we need to (1) find the OR-node u with $(v, u) \in E$ and $l(u) < l(v)$; (2) check if there is a directed path from u to v; (3) if there is ((v, u) is a back edge) delete it. Note that, such edges (v_8, u_{13}), (v_{11}, u_{13}), (v_{12}, u_{13}) shown in Figure 3.6 are not back edges, since there is no path from u_{13} to v_8, v_{11} and v_{12}.

3.2.3 Construction of Acyclic Program Net

Usually, users of software or system sometimes need to repeat an action or a function, such as delete, join, and so on. This makes software developers inevitably adopt a loop structure when designing a software, which increases the complexity of software testing. As mentioned before, a loop needs only to be tested once, and hence for a program net with directed circuits, we can delete these directed circuits through removing back edges to make the program net acyclic according to Propositions 4∼6 discussed in the previous section.

Since any directed circuit in a self-cleaning program net must contain at least one OR-node in accordance with Propositions 1∼3, if we delete certain related back edges then a program net becomes acyclic. Therefore, the method of deleting back edges to make a program net with directed circuit acyclic is feasible.

Consequently, we design the following algorithm to construct an acyclic program net (to be denoted by $\widehat{PN}$) from layer net L_{PN} according to above discussion.

Algorithm 2 Construction of Acyclic Program Net

```
Require: A layer net $L_{PN} = (V, E, \alpha, \beta, l)$
Ensure: $\widehat{PN}$
 1: $G \leftarrow L_{PN}$
 2: let $L_1, L_2, \cdots, L_{max}$ be the layers of $L_{PN}$
 3: $i = 1$
 4: while $i \neq max$ do
 5:     $E_u \leftarrow \phi$
 6:     for each $u \in L_i$ do
 7:         if u is OR-node then
 8:             $E_u \leftarrow \{(v,u) \mid (v,u) \in E, l(u) < l(v)\}$
 9:         end if
10:     end for
11:     for each $e = (v,u) \in E_u$ do
12:         if there exists a directed path from $u$ to $v$ in G then
13:             delete edge $e$ from G
14:         end if
15:     end for
16:     i=i+1
17: end while
18: $\widehat{PN} \leftarrow G$
```

From Propositions 2-6, we have the following theorem.

Theorem 1: Let MPN=(PN, d^0) be a marked self-cleaning program net and $\widehat{PN}$ be a program net obtained by ≪Algorithm 2 Construction of Acyclic Program Net≫.
(1) $\widehat{PN}$ is acyclic and connected.
(2) The computational time complexity of ≪Algorithm 2 Construction of Acyclic Program Net≫ is $O(|E|(|V|+|E|))$. □

Proof: (1) Any directed circuit possesses at least one OR-node u from Proposition 3 and its input edge (v, u) included in the circuit satisfies $l(u)<l(v)$ from Propositions 4-6. Thus, (v, u) is an back edge, and further deleting all such edges as done in the lines 5∼15 in Algorithm 2 makes a program net acyclic and meanwhile keeps the connection of u and v since there is a path from u to v. Therefore the obtained $\widehat{PN}$ is acyclic and connected.

(2) Finding a directed path in the lines 11∼15 takes computational time $O(|V|+|E|)$ and this operation would be iteratively executed $O(|E|)$ times. Therefore the computational time complexity is totally $O(|E|(|V|+|E|))$. □

Figure 3.9 shows the acyclic program net constructed by Algorithm 2 to delete two back edges ((v_{14}, v_5) and (v_{16}, v_8)) of the layer net in Figure 3.7.

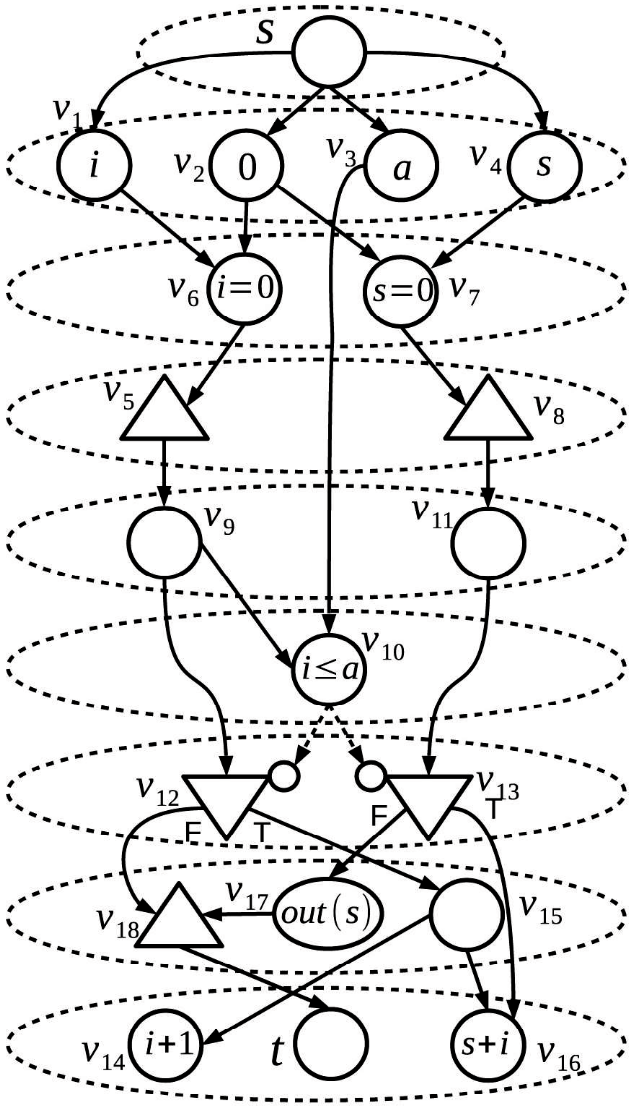
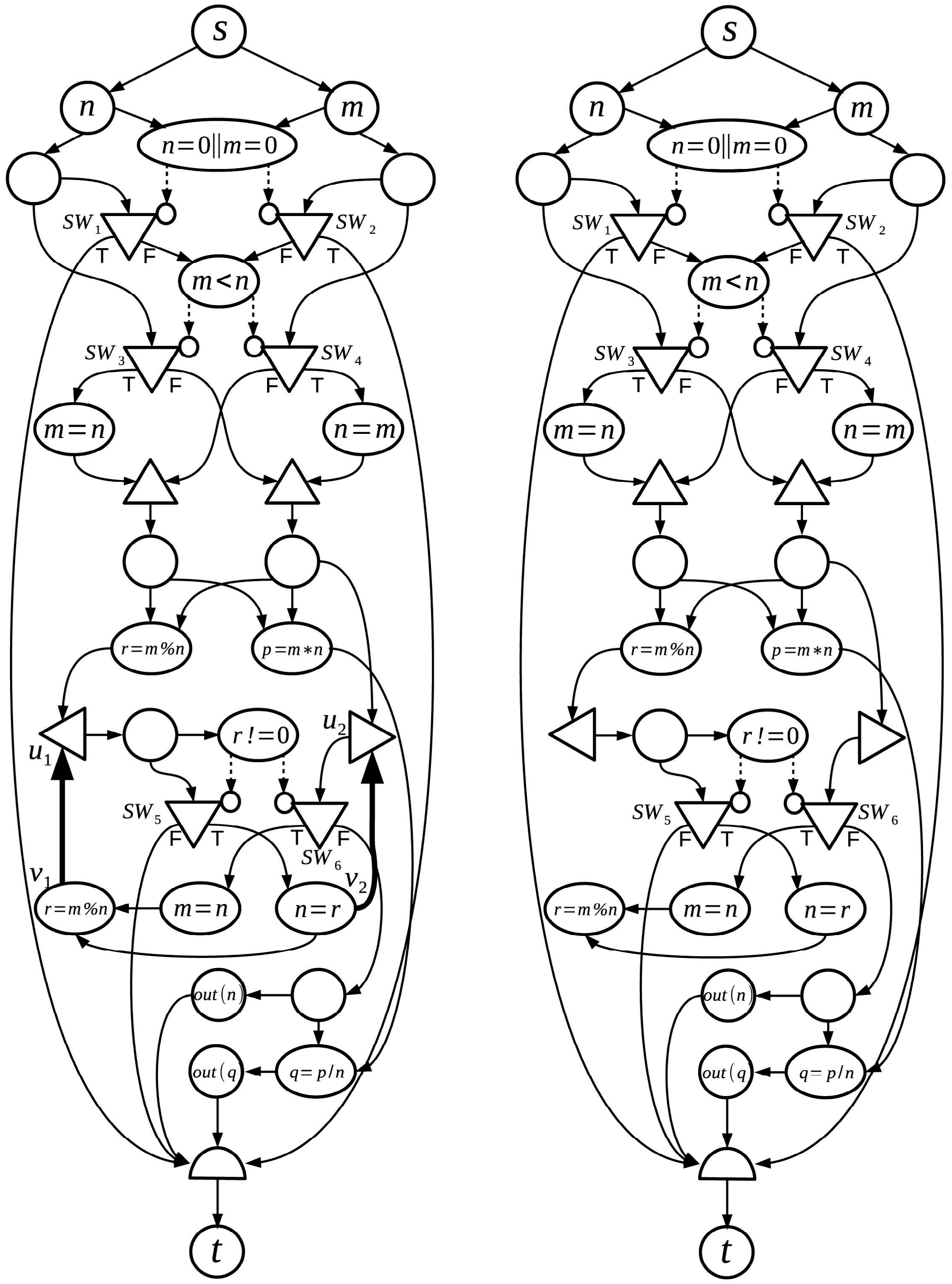


Figure 3.9: The acyclic program net of Figure 3.7.

We construct a program net PN from the program shown in Figure 2.4 (a). Then we apply Algorithms 1 and 2 to the program net PN shown in Figure 3.10 (a). As a result, its acyclic program net without displaying layer information is shown in Figure 3.10 (b). Hereafter, for convenience, all acyclic program nets no longer display layer information, just like Figure 3.10 (b).



(a) $(v_1, u_1)$and$(v_2, u_2)$to be deleted (b) Acyclic program net

Figure 3.10: A program net and its acyclic program net according to the program of Figure 2.4 (a).

3.3 Subnets Generation

In this section, we are to propose a method of generating subnets for the acyclic program net constructed in last section in order to finally find a set of test data used in software testing. A subnet will be such that all the nodes can fire for certain given input test data, which means the test data can cover the subnet. Further, if a set of input test data can make all the subnets firable, then this data set is what we want to find in software testing.

Further for the acyclic program net $\widehat{PN}$, we divide it into a set of subnets that include all the nodes of the original program net PN. Since each SWITCH-node of PN has the behaviors that the input tokens may flow to both its T (True) and F (False) terminals, the set of subnets should also include both T and F terminals of all the SWITCH-nodes.

3.3.1 Analysis of Acyclic Program Net

In a program net, there exists a possible situation that some SWITCH-nodes work under the same predicated condition. Such SWITCH-nodes are called synchronous nodes.

[**Definition 3.2**] Let SSW=$\{sw_1, sw_2, \cdots\}$ be a set of SWITCH-nodes. The nodes in SSW are called synchronous nodes, iff their control-flow edges are connected to a same node. □

In Figure 3.11, the control-flow edges of SWITCH-nodes sw_1 and sw_2 are connected to the same node v_2 and hence they are synchronous nodes. In the behavioral analysis of program net, these two synchronous nodes can be summarized into a group, SSW=$\{sw_1, sw_2\}$, so that the variations of a program net caused by SWITCH-nodes can be decreased.

Suppose a program net PN contains k SWITCH-nodes and has no synchronous SWITCH-nodes, W=$\{sw_1, sw_2, \cdots, sw_k\}$. Since a SWITCH-node transfers an input token to its T (say its state is c=1) or F (c=0) terminal depending on the value of the control token, it produces two nets and all nodes in each net can be firable. So there are 2^k topology variations $\widehat{PN}_1$,$\widehat{PN}_2$,$\cdots$,$\widehat{PN}_{2^k}$ of the original program net PN such that each $\widehat{PN}_j$ ($j = 1, 2, \cdots, 2^k$) is a program net with determinate state of all SWITCH-nodes and binary expression ψ_j=$(c_1, c_2, \cdots, c_k)$ is related to state c_i of SWITCH-node sw_i.

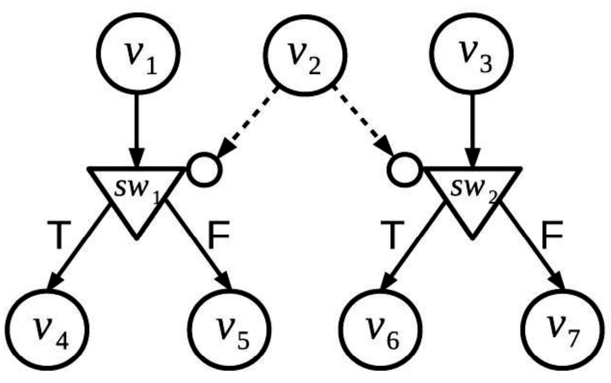


Figure 3.11: Synchronous SWITCH-nodes.

It is obviously that each topology variations can be as a switchless program net because of the determinate state of all SWITCH-nodes in it. Therefore, each SWITCH-node can be replaced by an AND-node and the state of each SWITCH-node is preserved. We can replace the branch node with and-node directly, but in order to facilitate the subsequent subnets generation and test data generation, we did not replace the branch nodes in the specific implementation.

Here we only analyze the nature of acyclic program net. The trajectory $\Theta=(\Gamma_0, \Gamma_1, \cdots)$ of the program execution is a description of fire ordering, such that Γ_i is a set of nodes simultaneously firable with respect to the token distribution resulted from prefix trajectory $(\Gamma_0, \Gamma_1, \cdots, \Gamma_{i-1})$. In the presence of SWITCH-nodes, the trajectories of the program execution are expressed by a concatenation of trajectories $\Theta_0\Theta_1\cdots\Theta_m$, such that Θ_0 is a trajectory on marked program net $MPN_{i_0}=(\widehat{PN}_{i_0}, d^0)$ where d^0 is a starting initial token distribution, Θ_1 is a trajectory on marked program net $MPN_{i_1}=(\widehat{PN}_{i_1}, d^1)$ where d^1 is the final token distribution resulted from Θ_0 on MPN_{i_0} and so on. We must note that each MPN_{i_k} consists only of AND-nodes and OR-nodes but no SWITCH-nodes.

The following result is immediate and used as our basic principle of termination verification for a program net that includes SWITCH-nodes [79].

Proposition 7: Let $\widehat{PN}_1$,$\widehat{PN}_2$,$\cdots$, $\widehat{PN}_{2^k}$ be the switchless nets derived from a program net PN with k SWITCH-nodes. If each of the switchless nets is structurally terminating, then PN is structurally terminating.

Given a program net, we usually find that several SWITCH-nodes are synchronized or controlled by a single predicate so that certain T-F combinations never take place when the program net PN is executed.

Proposition 8: Suppose there are m sets of synchronized SWITCH-nodes, SSW_1, $SSW_2, \cdots, SSW_m$ in a program net PN with k SWITCH-nodes. The number of possible switchless nets is $2^{k-(\sum \gamma_i - 1)}$ where $SSW_i = \{sw_j^{(i)} | j=1, 2, \cdots, \gamma_i\}$.

According to Definition 3.2, a large scale program net always possess synchronous SWITCH-nodes, we divided all SWITCH-nodes into m synchronous groups, consequently, there are at most 2^m topology variations. According to Proposition 8, if there are two synchronized SWITCH-nodes, then half of 2^k switchless nets can be omitted from our discussion. Synchronized SWITCH-nodes can be found by tracing each predicate-evaluation node in such that it provides control tokens to several SWITCH-nodes. Figure 3.12 shows an example that the program net of Figure 1.2 has two switchless nets instead of four (there are two SWITCH-nodes).

The program net of Figure 1.2 is acyclic and contains two synchronized SWITCH-nodes, $SSW = \{v_4, v_5\}$. Hence it possesses two states, $\psi_1 = (0, 0)$ and $\psi_2 = (1, 1)$, and two topology variations instead of four (there are two SWITCH-nodes) respectively, $\widehat{PN}_1$ and $\widehat{PN}_2$, as shown in Figure 3.12.

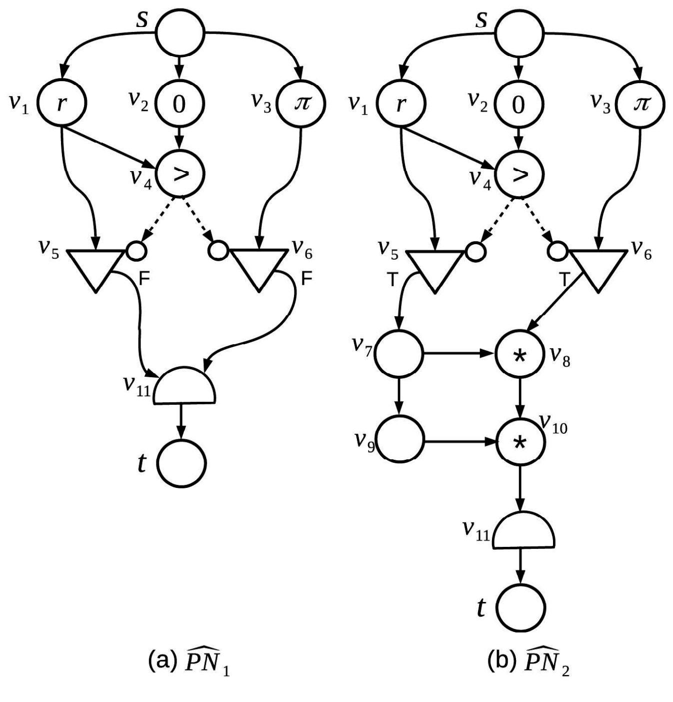


Figure 3.12: Two topology variations of Figure 1.2.

Another approach to structural analysis of termination is that, given a program net PN, we simply transform each SWITCH-node of PN into an AND-node so that the resultant net $\widehat{PN}$ is switchless. The following proposition is immediate from the transformation.

Proposition 9: Let PN be a program net including SWITCH-nodes and $\widehat{PN}$ be the switchless net obtained from PN by transforming each SWITCH-node of PN into an AND-node. PN is structurally terminating if $\widehat{PN}$ is structurally terminating.

As stated before, a subnet $\widehat{PN}_j$ should be firable, and then $\widehat{PN}_j$ is connected and non-firable nodes need to be deleted, which can be done by iterating the following operations (i) and (ii):

(i) For a SWITCH-node sw with c, if c=1 (or 0), deleting all its F (or T) terminal's successor nodes until reaching at OR-nodes;

(ii) If an OR-node becomes a source (with no input edges), deleting this OR-node and further its successor nodes until reaching at other OR-nodes.

Let $\Psi=\{\psi_1, \psi_2, \cdots, \psi_{2^m}\}$ be the set of SWITCH-nodes' states and $WPN=\{\widehat{PN}_i|\psi_i \in \Psi\}$ be the set of all subnets. In each subnet $\widehat{PN}_i$, a SWITCH-node contains only T terminal or F terminal. If sw_j does not exist in $\widehat{PN}_i$, its state is marked by c_j=$*$.

According to the above discussions, it seems that we need only to find WPN and pick up some of them that can cover all nodes of $\widehat{PN}$. Nevertheless, m is generally a large number and the computational time complexity is proportional to $O(2^m)$. Therefore, we are to design a polynomial algorithm to find subnets covering $\widehat{PN}$. There are many possible combinations in selecting subnets and the better selection is to have the number of subnets as less as possible. Hence, we need to avoid duplicate selection of T and F terminals of SWITCH-nodes in selecting subnets. Concretely, we first find an initial subnet and then iteratively choose the others in which the T or F terminals of SWITCH-nodes have not been included in obtained subnets as possible.

There are techniques to discover subgraphs (expressing elements or units) in a graph (expressing a program or a system), such as finding a group of interacting program elements in control flow graphs of a program [80] and selecting custom function units to meet the demands of an application in a system [81]. However these techniques cannot be applied, since we have to choose the necessary T or F terminals for each SWITCH-node in order to make a whole subnet executable for some specific input test data.

3.3.2 Initial Subnet Generation

We have pointed out previously that it is not feasible to (a) first generate all the subnets of a given program net, and then (b) select a part of them to cover all nodes of the net, since the processing of (a) will take a huge amount of time. Therefore, we try to directly generate a set of subnets that can cover all nodes as well as T and F terminals of all SWITCH-nodes through (i) generating an initial subnet; (ii) generating other subnets based on the initial subnet.

In the following discussions, we firstly generate an initial subnet from $\widehat{PN}$ by the following algorithm that sets states for all the SWITCH-nodes and deletes a part of their successors that may not be fired.

Algorithm 3 Generation of an Initial Subnet $\widehat{PN}_0$

```
Require: $\widehat{PN}$, $L_{PN}$'s layers: $L_1, L_2, \cdots, L_{max}$
Ensure: $\widehat{PN}_0$, $\psi_0$
 1: $\widehat{PN}' \leftarrow \widehat{PN}$
 2: $V_{ToBeDel} \leftarrow \phi$
 3: for each synchronous SWITCH-node set $SSW$ of $\widehat{PN}'$ do
 4:     set $c = 0$ or $c = 1$ randomly
 5:     for each $sw_i \in SSW$ do
 6:         set state $c_i = c$
 7:         $V_{ToBeDel} \leftarrow V_{ToBeDel} \cup \{sw_i\}$
 8:     end for
 9: end for
10: procedure DelNode($\widehat{PN}'$, $V_{ToBeDel}$)
11:     while $V_{ToBeDel} \neq \phi$ do
12:         take out a node $x$ with $l(x) = \min\{l(z) \mid z \in V_{ToBeDel}\}$
13:         if $x$ is SWITCH-node then
14:             if $sw_j$ has $c_j = 0$ (or 1) then
15:                 find its $T$(or $F$) terminal output node set $CH$
16:                 delete $T$ (or $F$) terminal edge
17:             else if $sw_j$ has $c_j = *$ then
18:                 find its output node set $CH$
19:                 delete $x$
20:             end if
21:         else if $x$ is AND-node or OR-node then
22:             find its output node set $CH$
23:             delete $x$
24:         end if
25:         for each node $y$ in CH do
26:             if $y$ is a SWITCH-node $sw_j$ then
```

```
27:                 set $c_j = *$
28:                 $V_{ToBeDel} \leftarrow V_{ToBeDel} \cup \{y\}$
29:             **end if**
30:             set state $c_i = c$
31:             $V_{ToBeDel} \leftarrow V_{ToBeDel} \cup \{sw_i\}$
32:         **end for**
33:     **end while**
34: **end procedure**
35: $\widehat{PN}_0 \leftarrow \widehat{PN}'$
36: **Output** $\widehat{PN}_0$ and its $\psi_0 = (c_1, c_2, \cdots, c_k)$
```

The computational time complexity of the above algorithm is $O(|V|^2+|E|)$. We apply Algorithm 3 to Figure 3.3 to explain the full process of initial subnet generation.

Firstly, the inputs of Algorithm 3 are the program net PN shown in Figure 3.3 and its corresponding layer net L_{PN} shown in Figure 3.6. Because PN is acyclic and possesses two synchronous nodes group SSW_1=$\{sw_1, sw_2, sw_3\}$ and SSW_1=$\{sw_4, sw_5\}$, we can set c_1=c_2=c_3=1, c_4=c_5=0 for each SWITCH-node in synchronous nodes group. As the result, an initial subnet $\widehat{PN}_0$ of PN is generated as shown in Figure 3.13 and its state ψ_0=$(0, 0, 1, 1, 0, 0)$ is obtained.

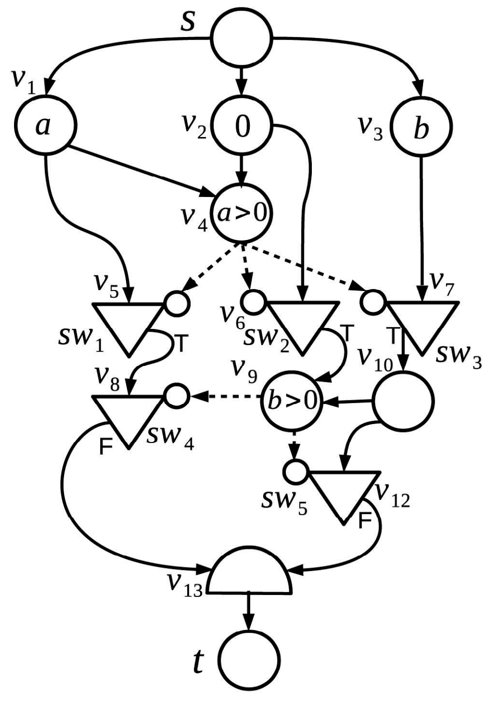


Figure 3.13: The initial subnet of Figure 3.3.

3.3.3 Subnets Set Generation

So far, we have generated an initial subnet $\widehat{PN}_0$ with SWITCH-nodes state ψ_0. The initial subnet can only cover a part of nodes and SWITCH-nodes state in the original program net PN. Therefore, we need to generate other subnets so that the obtained subnets set including the initial subnet can cover all the nodes of the original program net and also include both T and F terminals of all SWITCH-nodes.

Based on the obtained $\widehat{PN}_0$, we propose an algorithm to generate other subnets that (including $\widehat{PN}_0$) can cover all possible states of SWITCH-nodes. Before explaining the algorithm, in order to avoid duplicate selection of T and F terminals of SWITCH-nodes, we introduce a new expression $\tilde{\psi}$=$(\tilde{c_1}, \tilde{c_2}, \cdots, \tilde{c_k})$, in which each element $\tilde{c_i}$ may have the values "0", "1", "2" and "$*$". $\tilde{c_i}$=0 (1, 2, $*$) means SWITCH-node sw_i's "F Terminal" ("T Terminal", "both F and T Terminals", "neither F nor T Terminals") has appeared in the subnets obtained so far. The following algorithm is to find such a set of subnets that all the SWITCH-nodes finally have $\tilde{c}$=2. An operator $\odot$ for $\tilde{\psi}$ and ψ is defined as follows:

$$\begin{cases} \tilde{\psi} \leftarrow \tilde{\psi} \odot \psi = (\tilde{c_1} \odot c_1, \tilde{c_2} \odot c_2, \cdots, \tilde{c_k} \odot c_k) \\ \tilde{c_i} \odot c_i = \begin{cases} 0 & (\tilde{c_i} \in \{0, *\}, c_i \in \{0, *\}, \text{except } \tilde{c_i} = c_i = *) \\ 1 & (\tilde{c_i} \in \{1, *\}, c_i \in \{1, *\}, \text{except } \tilde{c_i} = c_i = *) \\ 2 & ((\tilde{c_i}, c_i) \in \{(0,1), (1,0), (2,0), (2,1), (2,*)\}) \\ * & (\tilde{c_i} = c_i = *) \end{cases} \end{cases}$$

Initially, $\tilde{\psi}$=$(*, *, \cdots, *)$ is set. After obtaining $\widehat{PN}_0$ with ψ_0, $\tilde{\psi}$ is updated according to the above equations. In fact, $\tilde{\psi}$=ψ_0 at the time when $\widehat{PN}_0$ is obtained.

Algorithm 4 Generation of Coverable Subnets CSN

```
Require: $\widehat{PN}$, $\widehat{PN}_0$, $\tilde{\psi} = \psi_0$, $L_{PN}$'s layers: $L_1, L_2, \cdots, L_{max}$
Ensure: $CSN$
 1: $CSN \leftarrow \{\widehat{PN}_0\}$
 2: $V_{ToBeDel} \leftarrow \phi$
 3: $\psi = (c_1, c_2, \cdots, c_k) = (*, *, \cdots, *)$
 4: let $W$ be the set of all SWITCH-nodes in the $\widehat{PN}$
 5: while true do
 6:     $\widehat{PN}' \leftarrow \widehat{PN}$
 7:     Let $SSW$ be a set of synchronous SWITCH-nodes.
 8:     if $sw_i \in SSW$ with $\tilde{c_i} \neq 2$ exists then
 9:         $W' \leftarrow W - SSW$
10:     **else**
11:         **break**
12:     **end if**
13:     **for each** $sw_i$ of $SSW$ **do**
14:         **if** $\tilde{c_i} = 0$ (or 1) **then**
15:             set $c_i = 1$ (or 0)
16:         **else if** $\tilde{c_i} = *$ **then**
17:             set $c_i = 0$ (or 1) randomly
18:         **end if**
19:         $V_{ToBeDel} \leftarrow V_{ToBeDel} \cup \{sw_i\}$
20:     **end for**
21:     do DELNODE($\widehat{PN}'$, $V_{ToBeDel}$)
22:     $\tilde{\psi} \leftarrow \tilde{\psi} \odot \psi$
23:     $PRE_{SSW} = \{v|v \in Pre(sw), sw \in SSW\}$
24:     $SUC_{SSW} = \{v|v \in Suc(sw), sw \in SSW\}$
25:     **for each** $sw_i \in W' \cap PRE_{SSW}$ **do**
26:         **if** $sw_i$ is a predecessor of any SWITCH-node in $SSW$ through its $T$ (or
    $F$) terminal **then**
27:             set $c_i = 1$ (or 0)
28:         **else if** $sw_i$ is a predecessor of one or more SWITCH-nodes in $SSW$ through
    both of its $T$ and $F$ terminals **then**
29:             set $c_i = 1$ (or 0) randomly
30:         **end if**
31:         $V_{ToBeDel} \leftarrow V_{ToBeDel} \cup \{sw_i\}$
32:     **end for**
33:     **for each** $sw_i \in W' \cap SUC_{SSW}$ **do**
34:         **if** $\tilde{c_i} = 0$ (or 1) **then**
35:             set $c_i = 1$ (or 0)
36:         **else**
37:             set $c_i = 0$ (or 1) randomly
38:         **end if**
39:         $V_{ToBeDel} \leftarrow V_{ToBeDel} \cup \{sw_i\}$
40:     **end for**
41:     do DELNODE($\widehat{PN}'$, $V_{ToBeDel}$)
42:     $\tilde{\psi} \leftarrow \tilde{\psi} \odot \psi$
```

```
43:     $CSN \leftarrow CSN \cup \{\widehat{PN}'\}$
44: **end while**
45: **Output** $CSN$
```

Theorem 2: Let $\widehat{PN}$=(V, E) be an acyclic program net, CSN be a set of subnets generated by ≪Algorithm 4 Generation of Coverable Subnets CSN≫.
(1) CSN can cover all nodes in $\widehat{PN}$, i.e., each node of $\widehat{PN}$ is included at least in one subnet of CSN.
(2) The computational time complexity is $O(k(|V|^2+|E|))$, where k is the number of SWITCH-nodes. □

Proof: (1) When the algorithm stops, every SWITCH-node sw_i satisfies $\tilde{c_i}$=2, which means its T and F terminals are individually included at least one subnet of CSN. Let v be an AND-node. If there is only one or no SWITCH-node included in $Pre(v)$, then obviously v appears at least in one subnet, and hence we need only to consider the case that v has more than one SWITCH-node included in $Pre(v)$. Suppose v has two input nodes, of which one is a SWITCH-node sw_x connecting to v through its T (or F) terminal and the other is any node z. From the deletion operations in ≪Algorithm 3 Generation of an Initial Subnet $\widehat{PN}_0$≫, the possible case for v not to appear in any subnets is, (i) z has at least one SWITCH-nodes sw_y (included in $Pre(z)$) connecting to z through its T (or F) terminal; and (ii) sw_x and sw_y never simultaneously switch to their T (or F) states, which means sw_x and sw_y are synchronous. Such existence of synchronous SWITCH-nodes obviously contradicts our precondition of self-cleanness, since there always remain tokens on edge (sw_x, v) or (z, v).

(2) The number of iterations from the lines 5∼44 is at most k. In the steps from the lines 5∼44, the most time consuming steps are 4° and 8°, which takes $O(|V|^2+|E|)$ from Theorem 1. Therefore total time complexity is $O(k(|V|^2+|E|))$. □

We apply Algorithms 3 and 4 to the net of Figure 3.10 (b), which contains three synchronous groups of SWITCH-nodes SSW_1=$\{sw_1, sw_2\}$, SSW_2=$\{sw_3, sw_4\}$ and SSW_3=$\{sw_5, sw_6\}$.

Firstly, setting c_1=c_2=0, c_3=c_4=1 and c_5=c_6=0 respectively for the SWITCH-nodes in SSW_1, SSW_2 and SSW_3, and further applying ≪Algorithm 3 Generation of an Initial Subnet $\widehat{PN}_0$≫, we get an initial subnet $\widehat{PN}_0$ shown in Figure 3.14 (a). The state of SWITCH-node in $\widehat{PN}_0$ is ψ_0=$(0, 0, 1, 1, 0, 0)$.

Secondly, based on $\widehat{PN}_0$, we use ≪Algorithm 4 Generation of Coverable Subnets CSN≫ to generate new subnet $\widehat{PN}_1$. Referring $\tilde{\psi}$=ψ_0=$(0, 0, 1, 1, 0, 0)$, we choose

SSW_1 and set c_i for each sw_i with $\tilde{c}_i \neq 2$ to possibly let $c_i \odot \tilde{c}_i = 2$. As the result, $c_1 = c_2 = 1$ is set and a subnet $\widehat{PN}_1$ is obtained as shown in Figure 3.14 (b), which has $\psi_1 = (1, 1, *, *, *, *)$. Thus current $\tilde{\psi}$ and CSN are $\tilde{\psi} \leftarrow \tilde{\psi} \odot \psi_1 = (2, 2, 1, 1, 0, 0)$ and $CSN = \{\widehat{PN}_0, \widehat{PN}_1\}$.

Finally, we choose SSW_2 and set c_i for each its sw_i with $\tilde{c}_i \neq 2$. $c_3 = c_4 = 0$ is set and $\widehat{PN}_2$ is obtained as shown in Figure 3.14 (c). Resultantly, the SWITCH-node states of $\widehat{PN}_2$ is $\psi_2 = (0, 0, 0, 0, 1, 1)$, $\tilde{\psi} \leftarrow \tilde{\psi} \odot \psi_2 = (2, 2, 2, 2, 2, 2)$ and $CSN = \{\widehat{PN}_0, \widehat{PN}_1, \widehat{PN}_2\}$ hold. Since no SWITCH-node has $\tilde{c}_i \neq 2$, CSN is the required subnets set. Obviously all the nodes of $\widehat{PN}$ are included in at least one of the subnets.

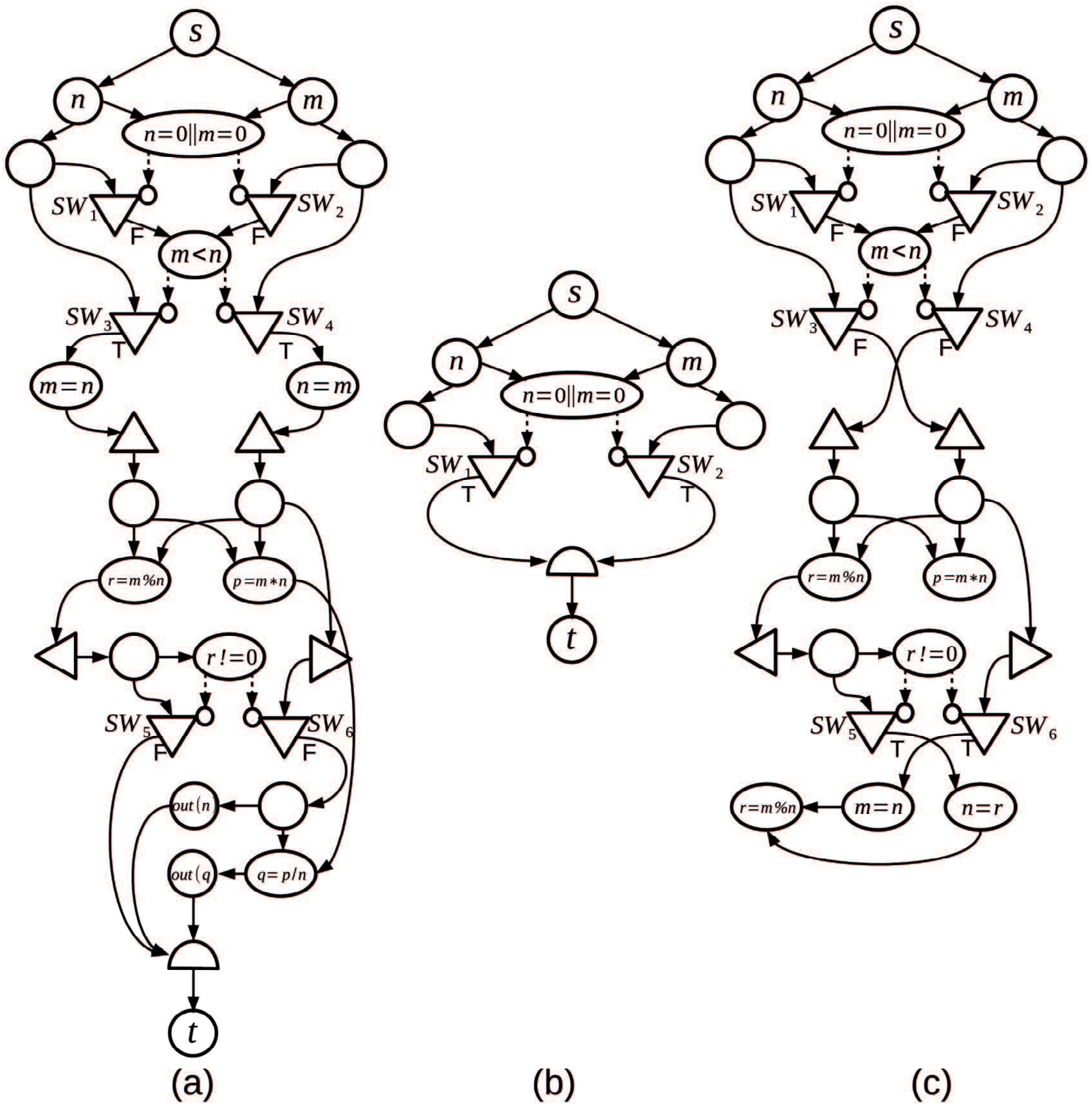


Figure 3.14: Generated subnets set CSN of Figure 3.10 (b).

Chapter 4

Test Data Generation for Subnets

In Chapter 3, we have structurally constructed a set of subnets that can cover all nodes for a given program net. This chapter will pay much attention to the detailed behaviour of nodes in subnets. Therefore, as the final goal, we are to propose a method of generating test input data for a given subnet through analyzing its dynamic behaviour and using SMT solver.

4.1 Basic Consideration

Software testing consists of a series of activities including quality assurance, verification and validation, reliability estimation and so on. Test data generation is an important testing activity aimed at finding errors of a software program. Currently, many test data generation techniques have been developed. As a code-based testing method, data flow testing is a white box technique, which uses data-flow relations in a program as test requirement, and the test goal is to select proper test data to cover the requirement.

In data flow testing, almost all researchers select All-DU-PATHS as the test adequacy criteria to guide the generation of test data generation, because All-DU-PATHS is one of the top data flow criteria in the test adequacy criteria system architecture proposed by Rapps and Weyuker. Therefore, general data flow testing technique contains two aspects: one is DU-paths generation introduced in Section 2.2; the other is the generation of test data that cover as much DU-paths as possible. The former has been introduced in Section 2.1. The latter is generating test data based on DU-paths and the generation process is shown in Figure 4.1.

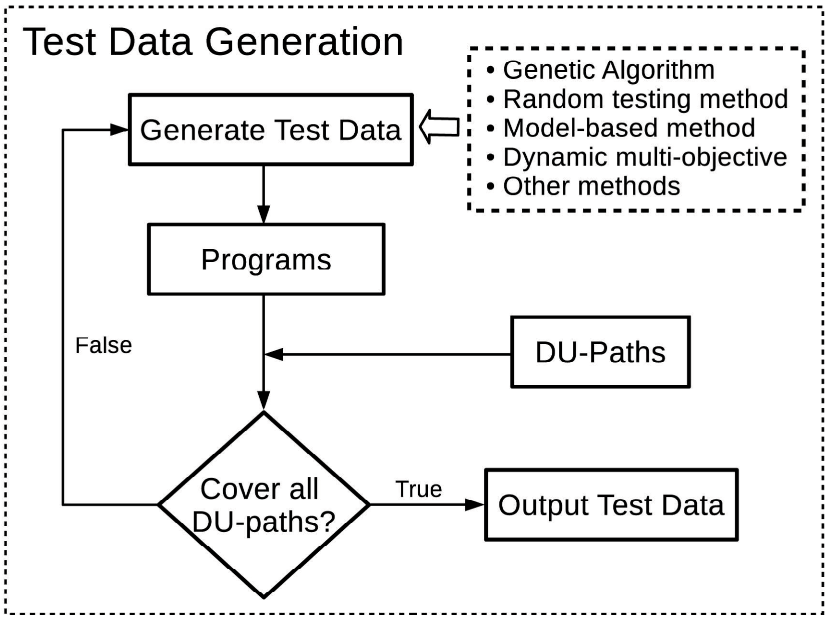


Figure 4.1: The processes of generating test data based on All-DU-Paths.

Corresponding to using program nets to approach software testing problem, we have already generated subnets that can cover all nodes of a given program net in Chapter 3. In this chapter, we mainly introduce how to generate test data for each subnet. The processes of test data generation are shown in Figure 4.3 and concretely include the following steps:

(1) According to the definition of exhaustive program nets in Section 2.1, we can add some dynamic behaviour information to a subnet and express it as an exhaustive subnet.

(2) Design a kind of adjacency matrix to express the static structure and dynamic behaviour information of an exhaustive subnet.

(3) Based on the adjacency matrix, design an algorithm to obtain all constraint conditions of the exhaustive subnet.

(4) Use an SMT solver to solve the constraint conditions and obtain a feasible solution. The solution is such test input data that we want to find for the exhaustive subnet.

In order to generate test data for a subnet, we need to mainly solve two important problems:

(i) How to represent dynamic behavior information of a subnet and how to design an adjacency matrix to express an exhaustive subnet.

(ii) How to design a polynomial algorithm to obtain all constraint conditions of an exhaustive subnet based on its adjacency matrix.

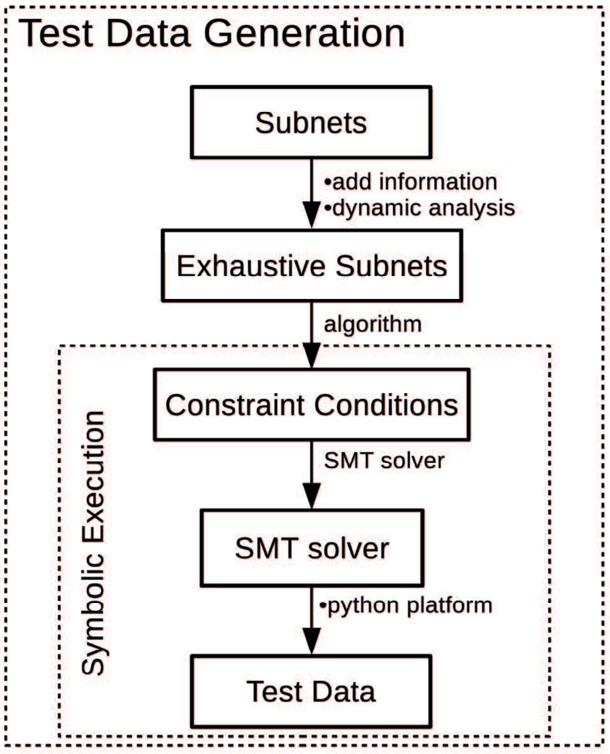


Figure 4.2: The processes of generating test data for subnets.

4.2 Constraint Conditions Generation

In this section, we propose algorithms to generate constraint conditions for the subnets obtained in Chapter 3 through analyzing the nets.

4.2.1 Analysis of Exhaustive Subnets

We have structurally constructed a set of subnets that can cover all nodes for a given program net in Chapter 3. 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 and how it flows in the subnet. Since 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.

To obtain constraint conditions, we need to express operation results for all the AND-nodes. Hence it is essential to (i) indicate from which input edge the input data comes; and (ii) 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 such one that possesses at most two input edges in this paper in order to conveniently express constraint conditions. We can do so due to that any AND-node with three or more input edges can be simply transformed to ones with two input edges as shown in Figure 4.3.

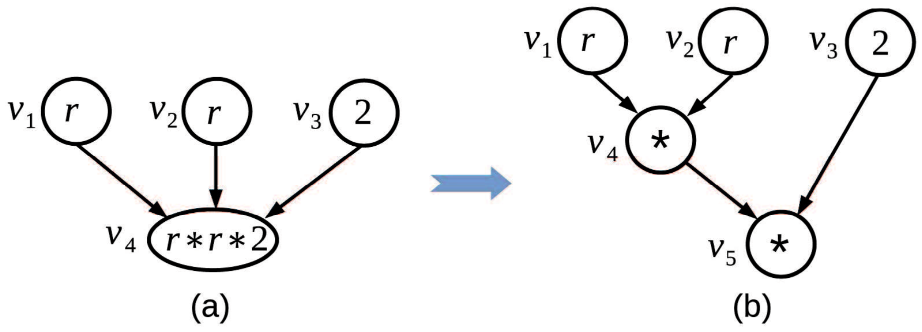


Figure 4.3: Transform operation from (a) to (b).

Based on the above discussions, we extend a general program net to an exhaustive program net introduced in the section 2.2. For example, Figure 4.4 shows an exhaustive program net transformed from the general program net in Figure 3.3.

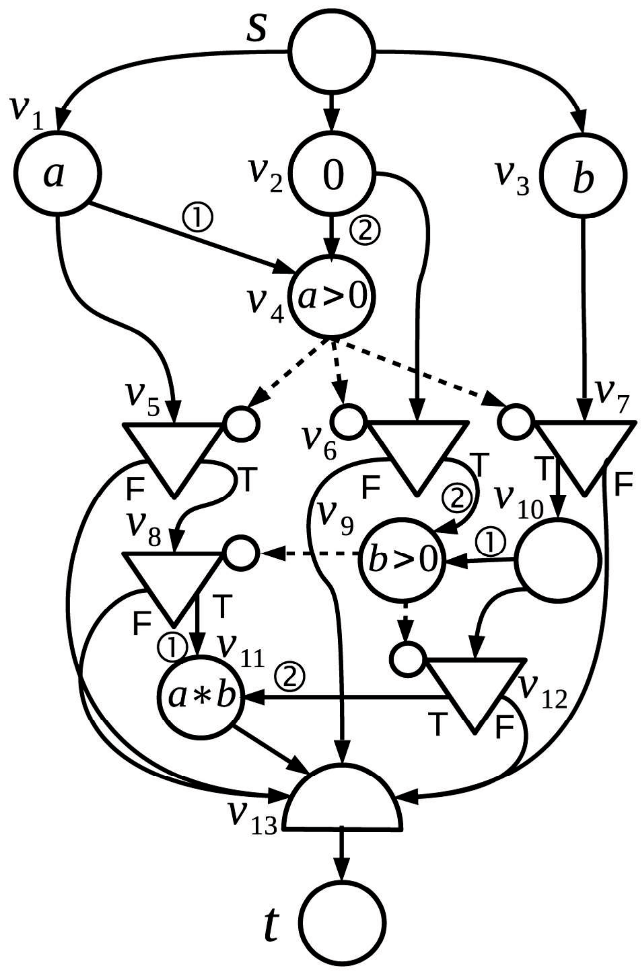


Figure 4.4: The exhaustive program net EN of Figure 3.3.

Hereafter, a set of subnets $\{\widehat{PN}_i\}$ obtained in Chapter 3 is expressed by a set of exhaustive subnets and denoted by $\{\widehat{EN}_i\}$. Figure 4.5 shows the exhaustive subnets of Figure 4.4, $\widehat{EN}_1$, $\widehat{EN}_2$ and $\widehat{EN}_3$. In the following discussions, we say choiceless exhaustive subnet or choiceless net denoted by $\widehat{EN}_i$ ($i{\in}N$), we mean an exhaustive subnet.

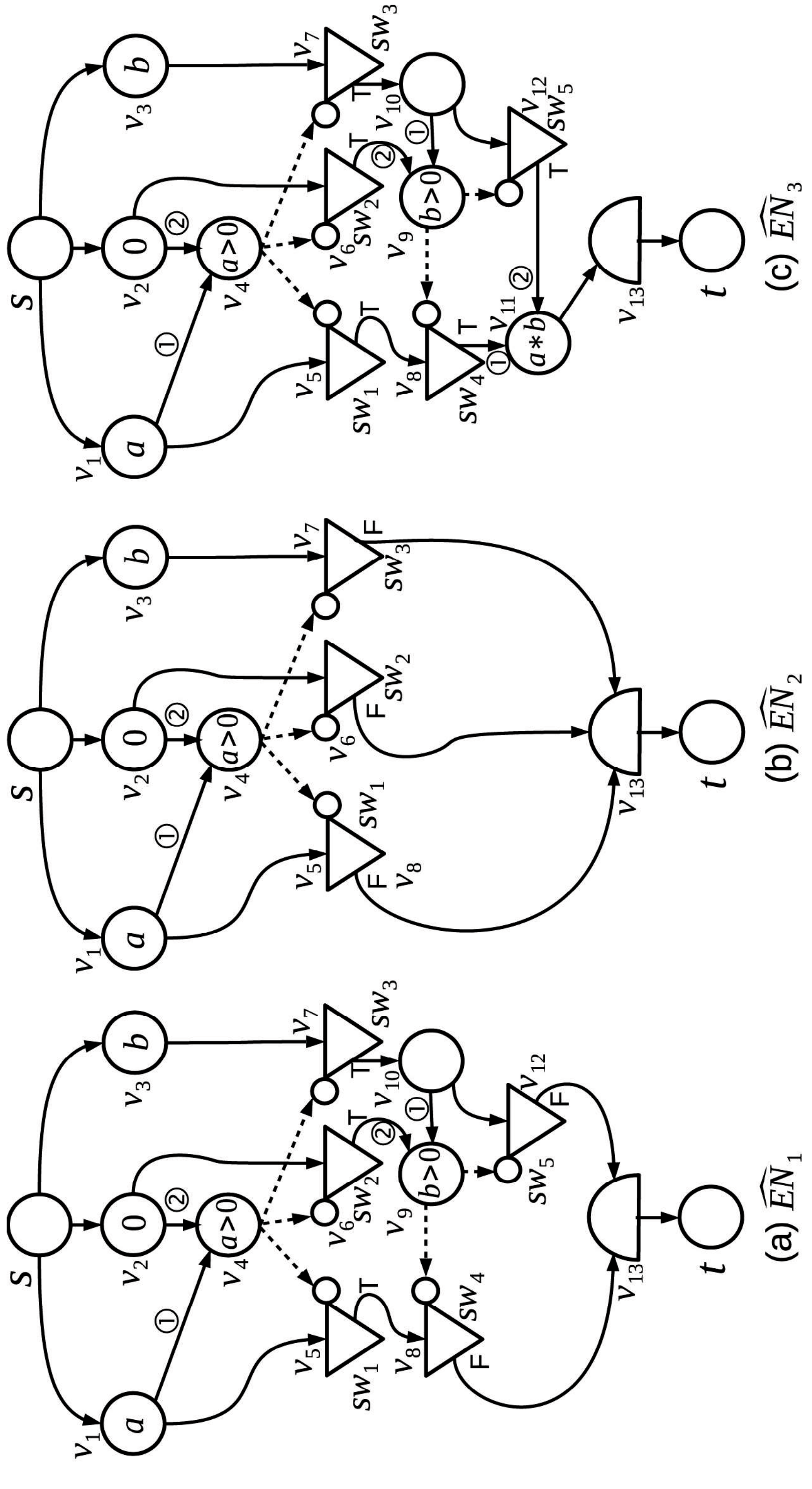


Figure 4.5: Exhaustive subnets $\widehat{EN}_1$, $\widehat{EN}_2$ and $\widehat{EN}_3$ of Figure 4.4.

Let's see how to use $o(v)$ and $r(e)$ to express operation results of the nodes in Figure 4.6. 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 2.9 (v). For Figure 4.6, $g(v_5)$= $g(v_4)o(v_5)$=$g(v_4)$++ 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 are the expressions of operation results of control-flow input nodes, which must meet T or F state of each related SWITCH-nodes in a choiceless net. For $\widehat{EN}_2$ shown in Figure 4.5 (b), the operation results of control-flow input nodes, v_4, are $g(v_4)$=$g(v_1)o(v_4)g(v_2)$=$(a>0)$. Since all the SWITCH-nodes, v_5, v_6 and v_7, have only F terminals, $(\neg a>0)$ is the constraint condition.

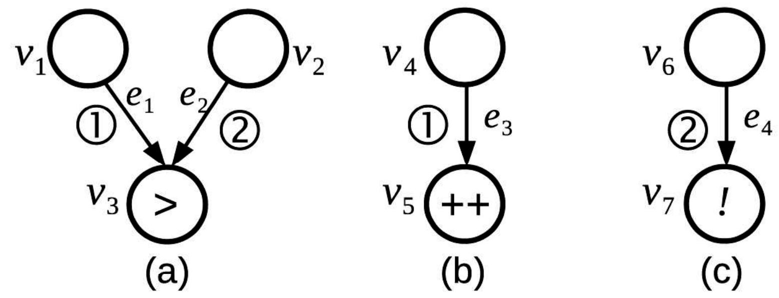


Figure 4.6: Example AND-nodes for expression of operation result.

Choiceless nets have the properties shown in the following propositions.

Proposition 10: Let $\widehat{EN}_0$ be a choiceless net. If all the SWITCH-nodes fire and further output data to their terminals that exist in $\widehat{EN}_0$ under a set of input data, then all the nodes of $\widehat{EN}_0$ can fire. □

Proof: Since each SWITCH-node fires and outputs data to its T or F terminal that is included in $\widehat{EN}_0$, then we can fire each node in such a way: (i) firing the start node and deleting it; (ii) recursively firing the source nodes and deleting them. Finally all the nodes will be deleted, i.e., all the nodes can fire, since there is no directed circuit in $\widehat{EN}_0$. □

Proposition 11: Let v_{SW} be a SWITCH-node in a choiceless net $\widehat{EN}_0$ and $Suc(v_{SW})$ be a set of its successor nodes. If $Suc(v_{SW})$ contains no SWITCH-nodes, then there exit no constraint conditions in the subnet induced by $Suc(v_{SW})$. □

This proposition obviously holds and hence we can simply delete such subnet induced by $Suc(v_{sw})$ in generating constraint conditions.

4.2.2 Expression of Exhaustive Subnets

From the above discussions, 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 $\widehat{EN}_0$. In order to facilitate the generation of constraint conditions, we construct an adjacency matrix A to express all the information of a choiceless net $\widehat{EN}_0$. The following adjacency matrix A is constructed to represent the net $\widehat{EN}_1$ of Figure 4.5 (a).

$$A = \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 10 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 10 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

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

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

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

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

Further, 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} means that there is an edge (v_i, v_j).

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

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.

4.2.3 Constraint Conditions Generation

According to the above description, what we need to do next is obtaining constraint conditions and finding test input data. Since each SWITCH-node corresponds one constraint condition in a choiceless net, we can find a way to obtain such all constraint conditions by doing two steps: (1) determining 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 11; (2) designing algorithms to obtain constraint conditions from the net obtained in step (1). The following Algorithm is for step (1).

Algorithm 5 Construction of Simplified Program Net

```
Require: A choiceless net $\widehat{EN}_0=(V, E, o, g, r, \alpha, \beta)$, adjacency matrix $A$ of $\widehat{EN}_0$
Ensure: Simplified Program Net $\widetilde{EN}_0$, adjacency matrix $B$
 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: $\widetilde{EN}_0\leftarrow\widehat{EN}_0$
```

```
13: while a sink node $v_x$ that is not a SWITCH-node ($b_{xx}{\neq}3$) exists in $\widetilde{EN}_0$ do
14:     $\widetilde{EN}_0 \leftarrow \widetilde{EN}_0 - \{v_x\}$
15:     update $B$ (delete $v_x$-related row and column)
16: end while
17: Output $\widetilde{EN}_0$
```

Theorem 3: Let $\widehat{EN}_0$ be a choiceless net and $\widetilde{EN}_0$ be the simplified net obtained by Algorithm 5.
(1) All sink nodes in $\widetilde{EN}_0$ 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, i.e., the execution of lines 13∼16 also takes $O(|V|^2)$. Therefore, the total computational time complexity is $O(|V|^2)$. □

The simplified program nets $\widetilde{EN}_1$, $\widetilde{EN}_2$ and $\widetilde{EN}_3$ of Figure 4.7 shows the result of applying Algorithm 5 to the choiceless nets of Figure 4.5 respectively through deleting sink nodes iteratively until all the sink nodes become SWITCH-nodes. The corresponding adjacency matrix B of $\widetilde{EN}_1$ showed in Figure 4.7 (a) is also obtained by updating the adjacency matrix A of $\widehat{EN}_1$ showed in Figure 4.5 (a).

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

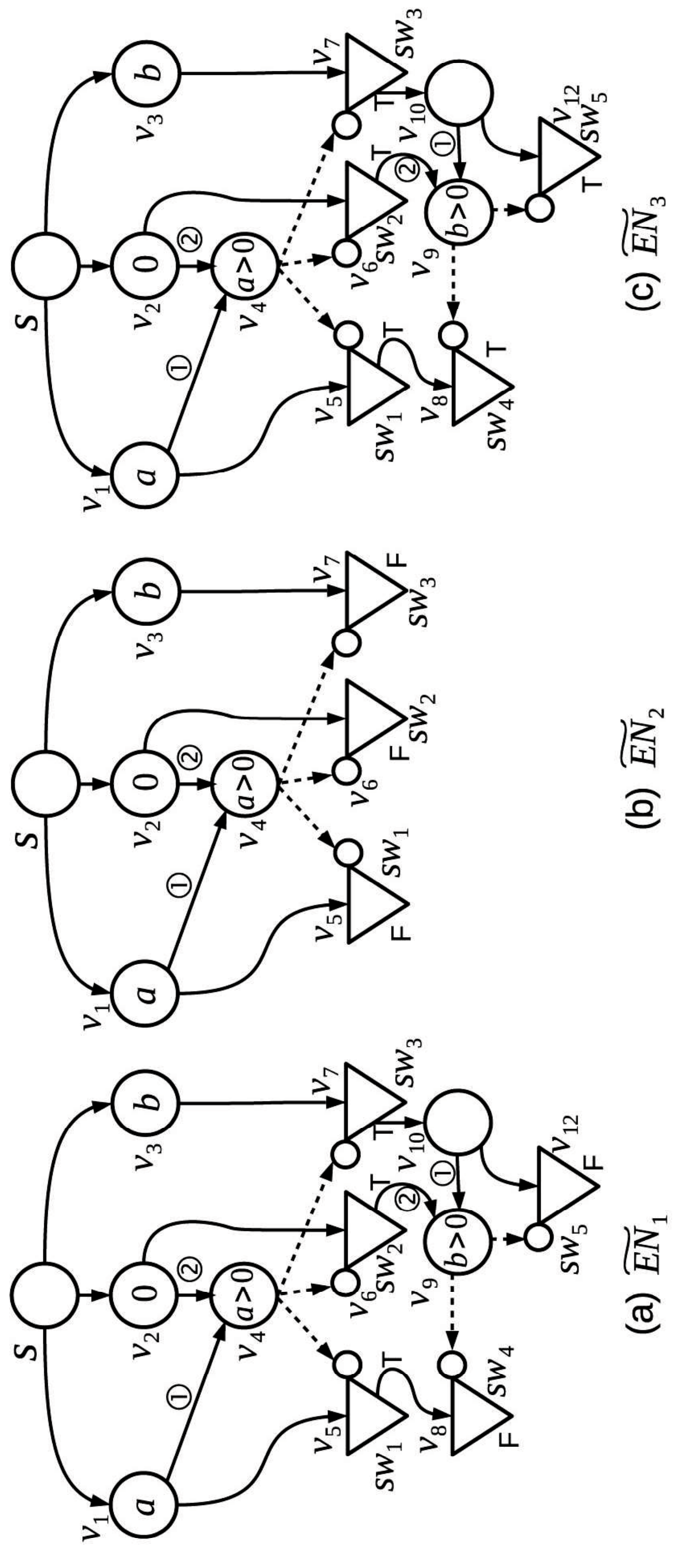


Figure 4.7: The simplified program net $\widetilde{EN}_1$, $\widetilde{EN}_2$ and $\widetilde{EN}_3$ of Figure 4.5.

The following Algorithm 6 is for step (2).

Algorithm 6 Generation of Constraint Conditions

```
Require: A simplified program net $\widetilde{EN}_0$, adjacency matrix $B$ of $\widetilde{EN}_0$
Ensure: Constraint conditions set $CCS$
 1: mark all nodes "unused"
 2: queue $Q \leftarrow \phi$
 3: enqueue $s$ (start node of $\widetilde{EN}_0$) 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 $v_j$ is AND-node without an operation description ($b_{jj}$=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 $v_j$ is AND-node with an operation description ($b_{jj}$<0) then
28:             if $v_j$ has no "unused" input node then
29:                 if $v_j$ has only one input edge $(v_i, v_j)$ ($b_{ij}$=1) then
30:                     $g(v_j) \leftarrow g(v_i)O[-b_{jj}]$
31:                 else if $v_j$ has only one input edge $(v_i, v_j)$ ($b_{ij}$=2) then
```

```
32:                     $g(v_j) \leftarrow O[-b_{jj}]g(v_i)$
33:                 **else if** $v_j$ has two input edges ($b_{ij}$=1 and $b_{xj}$=2) **then**
34:                     $g(v_j) \leftarrow g(v_i)O[-b_{jj}]g(v_x)$
35:                 **else if** $v_j$ has two input edges ($b_{ij}$=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 4: Let $\widehat{EN}_0$ be a choiceless program net, $\widetilde{EN}_0$ be the simplified net constructed from $\widehat{EN}_0$ by Algorithm 5 and CCS be the set of constraint conditions obtained by Algorithm 6.
(1) If there exists a set of test input data satisfying CCS, then all the nodes of $\widehat{EN}_0$ 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 outputs data to its T or F terminal included in $\widehat{EN}_0$. 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. And further it takes at most $2|V|$ times to generate constraint condition at lines 27∼37 in searching in B. Therefore, the computational time complexity is $O(|V|^2)$. □

By using Algorithm 6, we can obtain all constraint conditions of the exhaustive subnet $\widehat{EN}_1$ in Figure 4.5 (a). As the result, the constraint conditions can be synthesized as $F : (a > 0) \wedge (\neg(b > 0))$. The constraint conditions of other two exhaustive subnets in Figure 4.5 are shown in Table 4.1 respectively. In the following discussions, we are to find a solution satisfying such F.

Table 4.1: Constraint conditions of the program net in Figure 4.5.

Exhaustive Subnets	Simplified Subnets	Obtained Constraint Conditions
$\widehat{EN}_1$	$\widetilde{EN}_1$	$(a > 0) \wedge (\neg(b > 0))$
$\widehat{EN}_2$	$\widetilde{EN}_2$	$\neg(a > 0)$
$\widehat{EN}_3$	$\widetilde{EN}_3$	$(a > 0) \wedge (b > 0)$

4.3 Test Data Generation

In this section, we aim to concretely generate test input data based on the constraint conditions obtained in Section 4.2.

4.3.1 The Platform of SMT solver $Z3$

SMT solvers can be used in extended static checking, predicate abstraction, test case generation, bounded model checking over infinite domains and so on. Till now, many SMT solvers have been developed to find a solution satisfying a constraint condition. The SMT solvers include $Z3$ prover, Yices, SMT-RAT and so on, which are introduced in Chapter 2. Among these SMT solvers $Z3$ 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 [74, 82].

$Z3$ prover is a so useful and convenient tool that, (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; (iv) it has been developed as a package used in Python programing language and can be used conveniently. Therefore, we choose to use $Z3$ prover in generating test input data for choiceless nets.

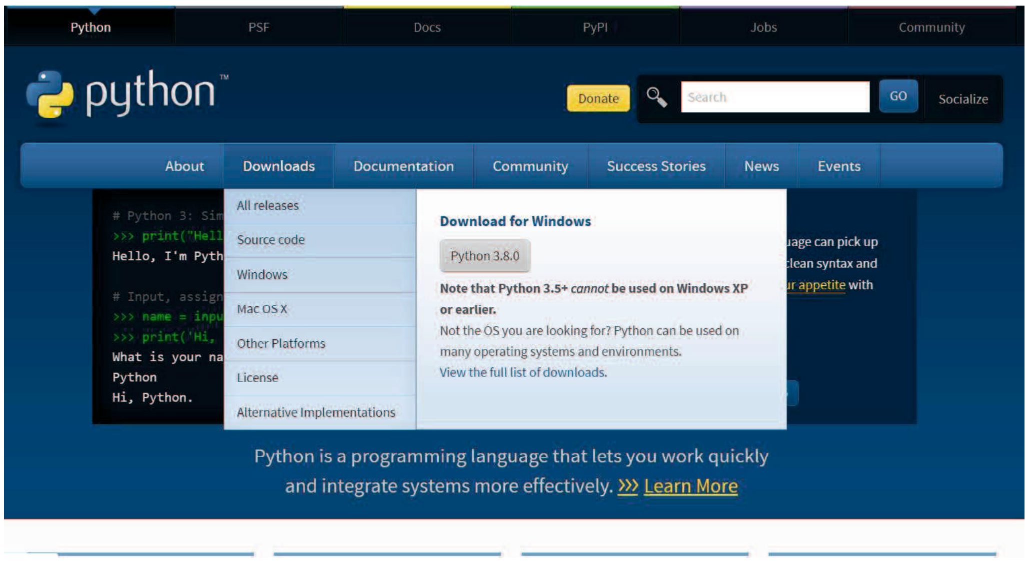
We will use $Z3$ prover to generate test data in such a way that, (a) to construct a Python compilation environment; (b) to install $Z3$ Application Programming Interface in Python. Because $Z3$ is unstable when used with Python 2, we apply Python 3. As of now, the latest version of Python 3 is Python 3.8.0. The following steps are for step (a).

(1) Go to “https://www.python.org/”, which is Python’s official website as shown in Figure 4.8. Move cursor onto “Downloads”, and click on “Python 3.8.0”, then Python should start download automatically.

(2) Go to “Downloads” directory in your computer. One can see a file named $python-3.8.0-amd64.exe$. Note that the filename extension may vary depending on the OS of your computer. Then open the file.

(3) An installation window like the one in Figure 4.9 should be found. It is recommended to use default settings in the installation. When the installation is complete, exit the installer.

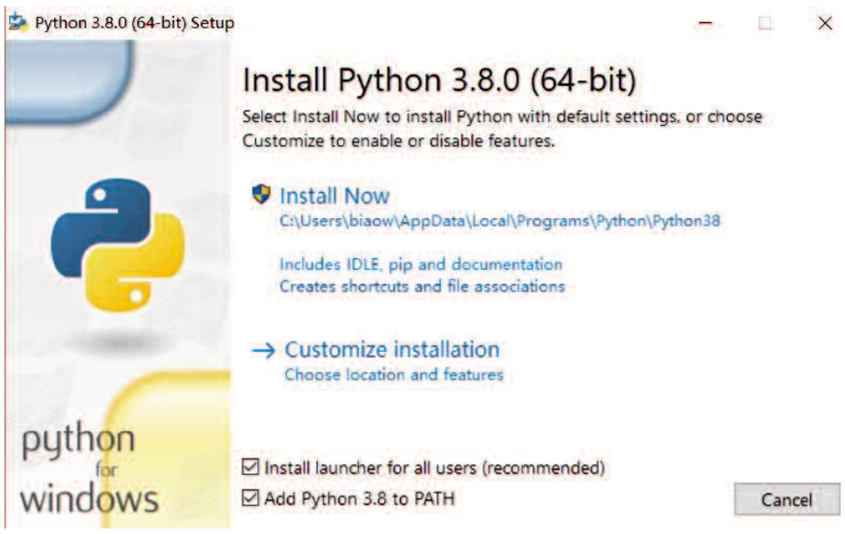
(4) Open the Integrated Development and Learning Environment (IDLE) for Python in the computer. After successfully installing Python 3.8.0, one will see the Python welcome screen as shown in Figure 4.10.



Note that Python 3.5+ *cannot* be used on Windows XP or earlier.
Not the OS you are looking for? Python can be used on many operating systems and environments.
View the full list of downloads.

Python is a programming language that lets you work quickly and integrate systems more effectively. >>> Learn More

Figure 4.8: Download the latest version of Python.



Install Python 3.8.0 (64-bit)

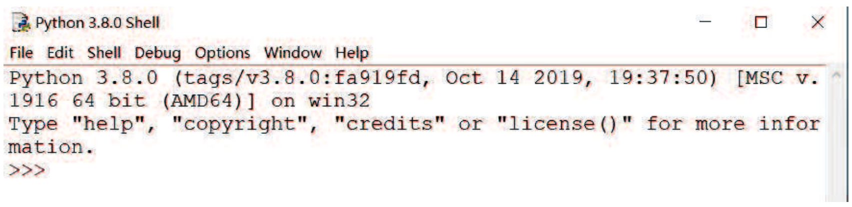
Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\biaow\AppData\Local\Programs\Python\Python38
Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.8 to PATH

Figure 4.9: Install Python 3.8.0.

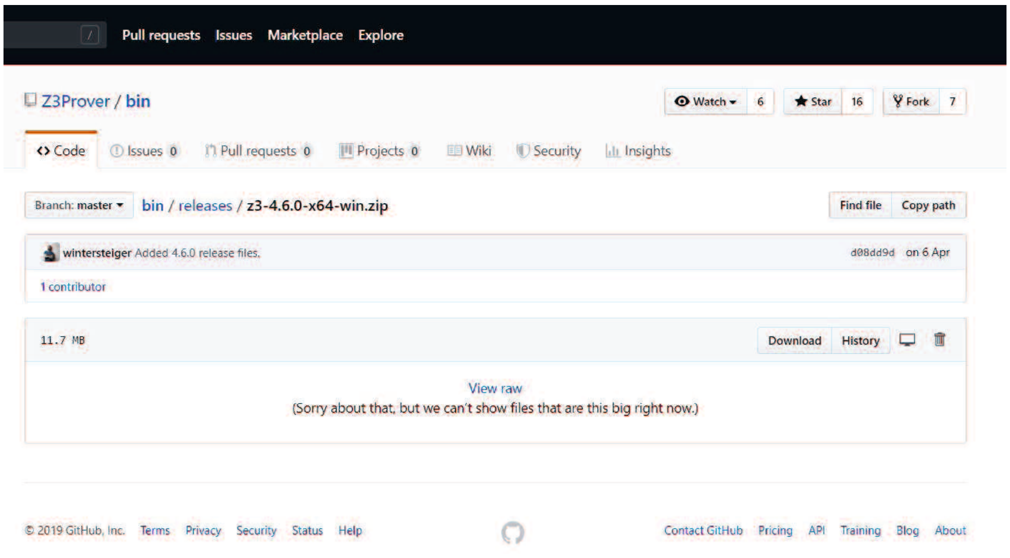


```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.
1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more infor
mation.
>>>
```

Figure 4.10: Open IDLE (Python 3.8.0 64-bit).

In order to generate test data based on constraint conditions, we will use *z3py*, which is an Application Programming Interface of *Z*3 prover in Python. The following steps will demonstrate how to install *z3py* on Windows platform.

(1) Go to "https://github.com/Z3Prover/bin" and click on "releases". Then select a *zip* file that matches corresponding Python's version and OS's version. When an interface as shown in Figure 4.11 appears, then click on "View Raw" to download $Z3$.



bin / releases / z3-4.6.0-x64-win.zip

wintersteiger Added 4.6.0 release files. d08dd9d on 6 Apr

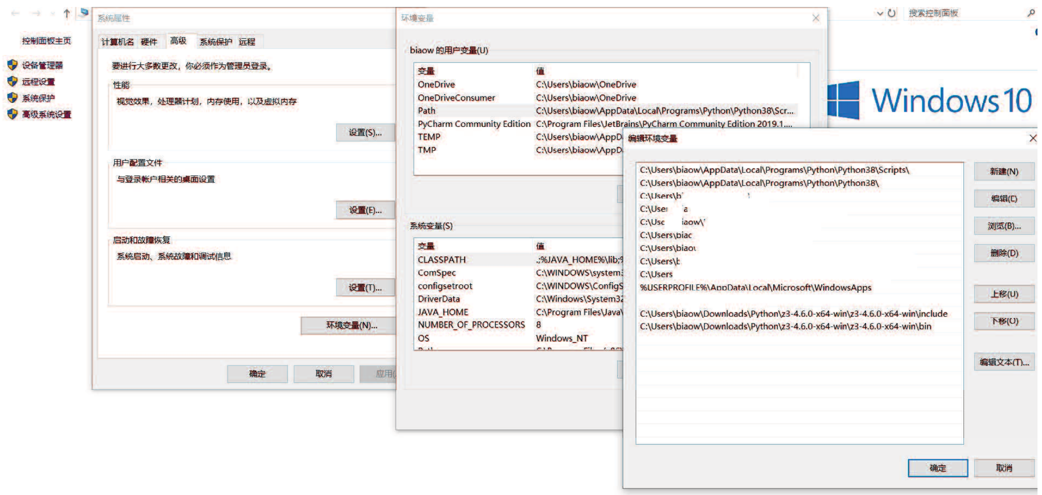
1 contributor

11.7 MB

(Sorry about that, but we can't show files that are this big right now.)

Figure 4.11: Download $Z3$ prover.

(2) Go to the directory you stored $Z3$ in the computer, and unzip the file you just downloaded. Then add two directory paths "bin" and "build" to the environmental variables as shown in Figure 4.12.



要进行大多数更改，你必须作为管理员登录。

性能

视觉效果，处理器计划，内存使用，以及虚拟内存

用户配置文件

与登录帐户相关的桌面设置

启动和故障恢复

系统启动、系统故障和调试信息

biaow 的用户变量(U)

变量	值
OneDrive	C:\Users\biaow\OneDrive
OneDriveConsumer	C:\Users\biaow\OneDrive
Path	C:\Users\biaow\AppData\Local\Programs\Python\Python38\Scr...
PyCharm Community Edition	C:\Program Files\JetBrains\PyCharm Community Edition 2019.1....
TEMP	C:\Users\biaow\AppD
TMP	C:\Users\biaow\AppD

系统变量(S)

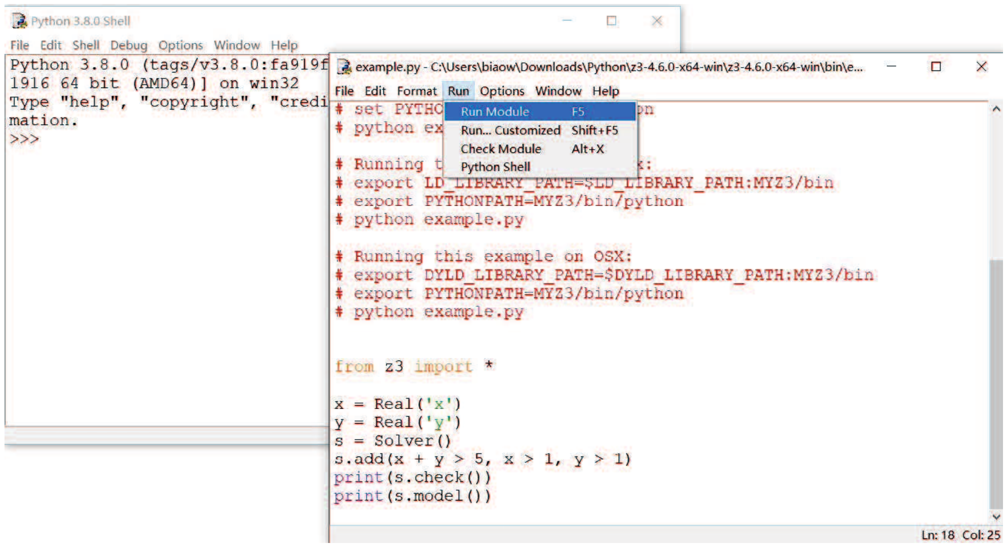
变量	值
CLASSPATH	.;%JAVA_HOME%\lib;%
ComSpec	C:\WINDOWS\system3
configsetroot	C:\WINDOWS\ConfigS
DriverData	C:\Windows\System32
JAVA_HOME	C:\Program Files\Java\
NUMBER_OF_PROCESSORS	8
OS	Windows_NT

```
C:\Users\biaow\AppData\Local\Programs\Python\Python38\Scripts\
C:\Users\biaow\AppData\Local\Programs\Python\Python38\
C:\Users\b[redacted]
C:\User[redacted]
C:\Us[redacted]iaow\
C:\Users\biac[redacted]
C:\Users\biao[redacted]
C:\Users\b[redacted]
C:\Users[redacted]
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps

C:\Users\biaow\Downloads\Python\z3-4.6.0-x64-win\z3-4.6.0-x64-win\include
C:\Users\biaow\Downloads\Python\z3-4.6.0-x64-win\z3-4.6.0-x64-win\bin
```

Figure 4.12: Add $Z3$ to the environmental variable.

(3) Opening the unzipped file and go to the "bin/python" directory, a file *example.py* can be found. Then right-click on *example.py* and select "open with IDLE", an in-

terface as shown in Figure 4.13 appears. We will create Python files in “bin/python” directory to generate test data.


Python 3.8.0 Shell
File Edit Shell Debug Options Window Help
File Edit Format Run Options Window Help
Run Module F5
Run... Customized Shift+F5
Check Module Alt+X
Python Shell
Ln: 18 Col: 25

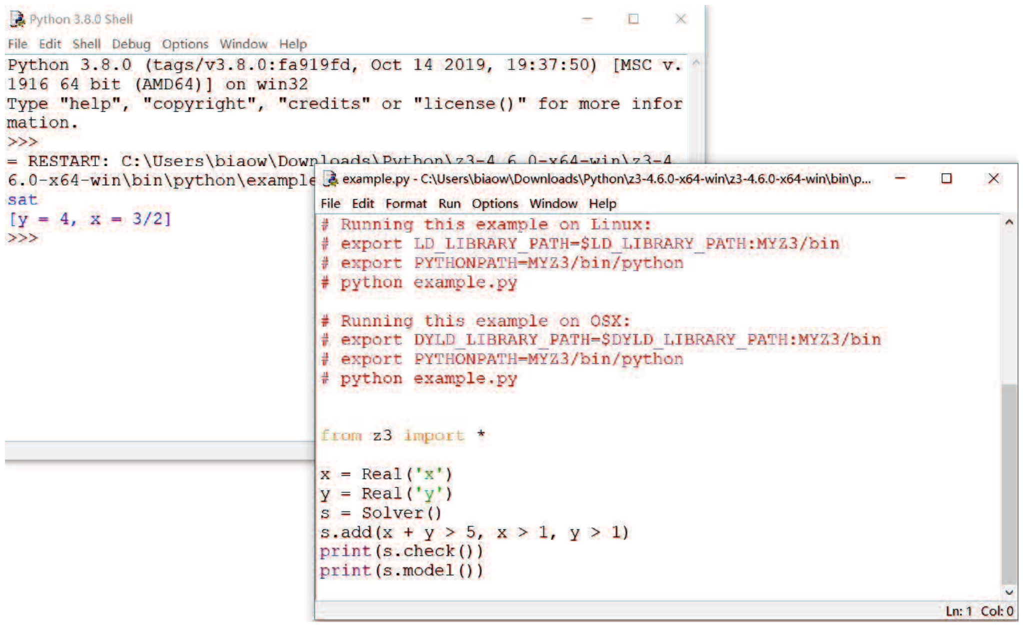
```
Python 3.8.0 (tags/v3.8.0:fa919f
1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credi
mation.
>>>
```

example.py - C:\Users\biaow\Downloads\Python\z3-4.6.0-x64-win\z3-4.6.0-x64-win\bin\e...

```
# set PYTHO                     on
# python ex

# Running t                     :
# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:MYZ3/bin
# export PYTHONPATH=MYZ3/bin/python
# python example.py

# Running this example on OSX:
# export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:MYZ3/bin
# export PYTHONPATH=MYZ3/bin/python
# python example.py


from z3 import *

x = Real('x')
y = Real('y')
s = Solver()
s.add(x + y > 5, x > 1, y > 1)
print(s.check())
print(s.model())
```

Figure 4.13: Open an example $Z3$ file in Python.

(4) Open *example.py* with IDLE and click “Run” → “Run Module” as shown in Figure 4.13. If a solution pops up onto the Python console as shown in Figure 4.14, it means $Z3$ has been successfully installed.


Python 3.8.0 Shell
File Edit Shell Debug Options Window Help
File Edit Format Run Options Window Help
Ln: 1 Col: 0

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.
1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more infor
mation.
>>>
= RESTART: C:\Users\biaow\Downloads\Python\z3-4.6.0-x64-win\z3-4.
6.0-x64-win\bin\python\example
sat
[y = 4, x = 3/2]
>>>
```

example.py - C:\Users\biaow\Downloads\Python\z3-4.6.0-x64-win\z3-4.6.0-x64-win\bin\p...

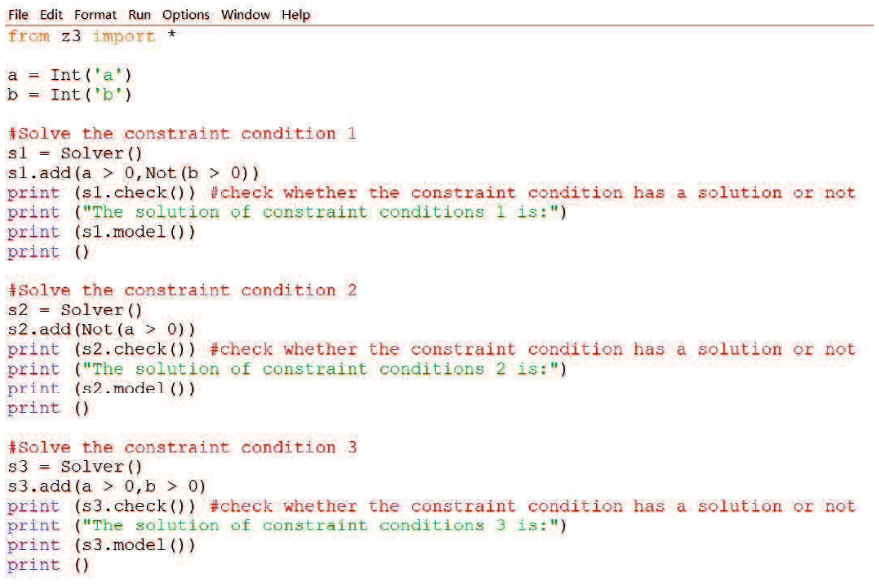
```
# Running this example on Linux:
# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:MYZ3/bin
# export PYTHONPATH=MYZ3/bin/python
# python example.py

# Running this example on OSX:
# export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:MYZ3/bin
# export PYTHONPATH=MYZ3/bin/python
# python example.py


from z3 import *

x = Real('x')
y = Real('y')
s = Solver()
s.add(x + y > 5, x > 1, y > 1)
print(s.check())
print(s.model())
```

Figure 4.14: Run a constraint condition of $Z3$ file in Python.

4.3.2 The Implement of Test Data Generation

Now we can use $Z3$ prover and Python to generate test input data based on constraint conditions.

Firstly, we write a Python program for all the obtained constraint conditions as shown in Table 4.1 and import $Z3$ package in the program header. The whole program is shown in Figure 4.15.



```
from z3 import *

a = Int('a')
b = Int('b')

#Solve the constraint condition 1
s1 = Solver()
s1.add(a > 0,Not(b > 0))
print (s1.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 1 is:")
print (s1.model())
print ()

#Solve the constraint condition 2
s2 = Solver()
s2.add(Not(a > 0))
print (s2.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 2 is:")
print (s2.model())
print ()

#Solve the constraint condition 3
s3 = Solver()
s3.add(a > 0,b > 0)
print (s3.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 3 is:")
print (s3.model())
print ()
```

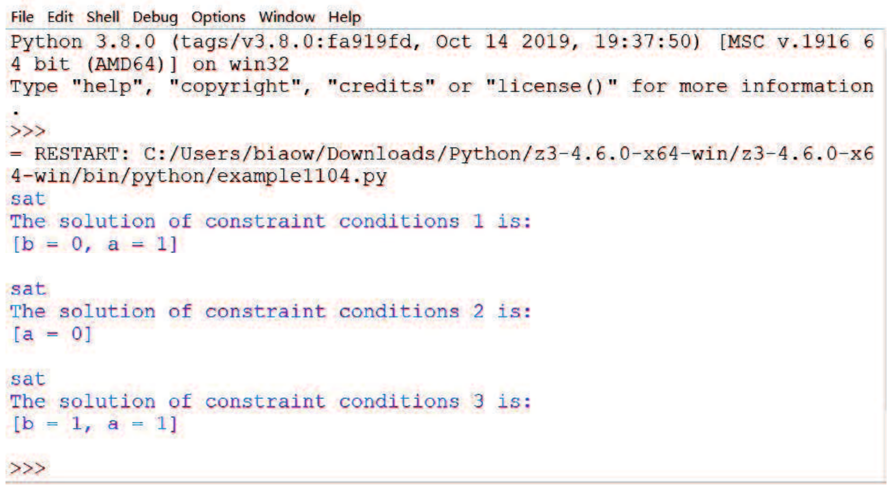
Figure 4.15: A program of solving the constraint conditions shown in the three column of Table 4.1 using $Z3$ prover in Python.

Then, click "Run" and "Run Module" in turn, the solutions of solving this three constraint conditions are shown in Figure 4.16. As an implement of test data generation, the test data of all three subnets in Figure 4.5 are shown in Table 4.2.

Table 4.2: Test input data of the program net in Figure 4.5.

Subnets	Simplified Subnets	Obtained Constraint Conditions	Test Input Data
$\widehat{EN}_1$	$\widetilde{EN}_1$	$(a > 0)\wedge(\neg(b > 0))$	$(a = 1, b = 0)$
$\widehat{EN}_2$	$\widetilde{EN}_2$	$\neg(a > 0)$	$(a = 0, \forall b \in Z)$
$\widehat{EN}_3$	$\widetilde{EN}_3$	$(a > 0)\wedge(b > 0)$	$(a = 1, b = 1)$

As the result, the test input data set "$(a, b) \in TD$=$\{(1, 0), (0, \forall Z), (1, 1)\}$" can cover all nodes and can be used to check all possible paths of the original net shown in Figure 3.3.



```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 6
4 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information
.
>>>
= RESTART: C:/Users/biaow/Downloads/Python/z3-4.6.0-x64-win/z3-4.6.0-x6
4-win/bin/python/example1104.py
sat
The solution of constraint conditions 1 is:
[b = 0, a = 1]

sat
The solution of constraint conditions 2 is:
[a = 0]

sat
The solution of constraint conditions 3 is:
[b = 1, a = 1]

>>>
```

Figure 4.16: The solutions of solving the constraint conditions shown in the three column of Table 4.1 using $Z3$ prover in Python.

Chapter 5

Case Studies

In Chapters 3 and 4, we have introduced the method of applying program nets to software testing detailedly including subnets generation and test data generation. In this chapter, we will use three actual examples to illustrate the entire processes of our method and its effectiveness.

5.1 Basic Consideration

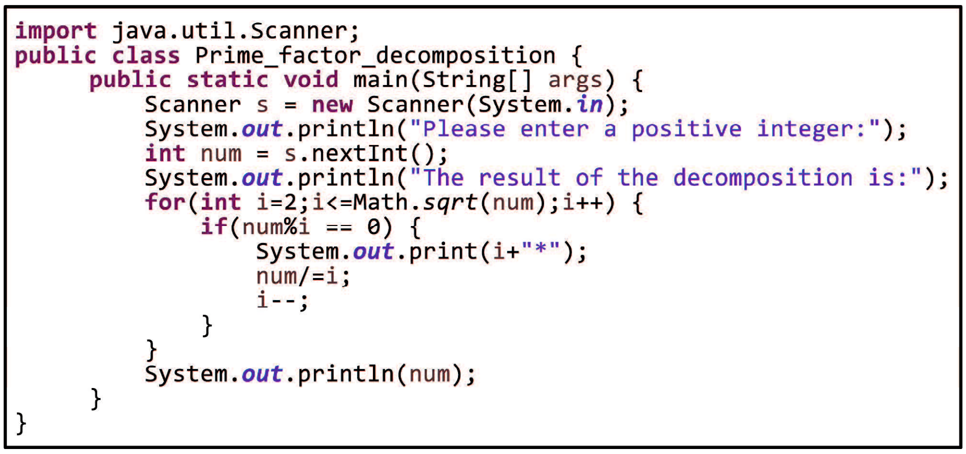
Till now, we proposed a method of applying program nets to approach a complex NP-complete problem of software testing. This method is mainly divided into two major steps: (1) subnets generation for a given program net; and (2) test data generation for these subnets. Here, three actual Java programs are used to verify the effectiveness of our method.

Because actual software testing is inseparable from a computer's support, we also need to make certain choices for computer configuration requirements. Since Python and $Z3$ prover do not have much requirements for memory and processing ability, a computer with more than 2G of memory can be used to conduct this experiment. Our computer configuration is shown in Table 5.1.

Table 5.1: The configuration of the computer.

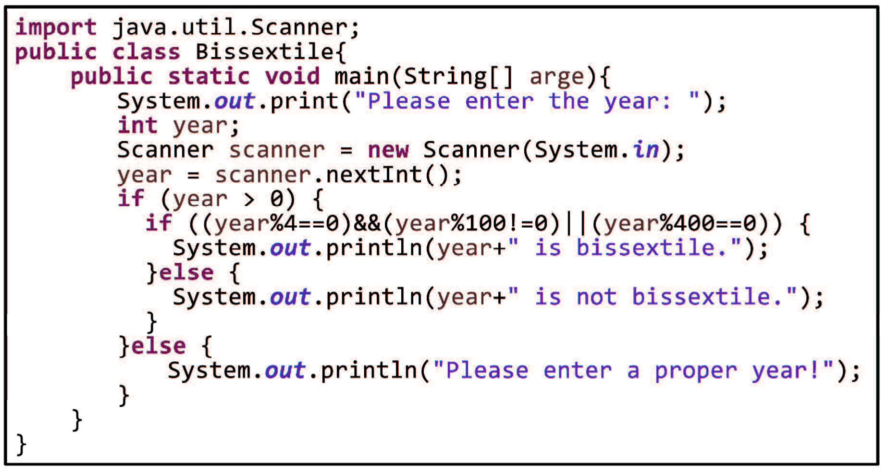
Items	Configuration
1. Sensor	resolution 1920 $\times$ 1080 pixels with gray 256 levels
2. Memory	8.00 GB RAM
3. CPU	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80GHz
4. Software	Python 3.8 with $Z3$
5. System Type	64-bit operating system, x64-based processor

Three actual Java programs are shown in Figures 5.1$\sim$5.3, respectively. Among these three programs, program 1 and program 3 contain loop structure (*for* and *while*) and all programs possess branch structure (*if*). Further, programs 2 and 3 include nested branch structure.



```
import java.util.Scanner;
public class Prime_factor_decomposition {
     public static void main(String[] args) {
         Scanner s = new Scanner(System.in);
         System.out.println("Please enter a positive integer:");
         int num = s.nextInt();
         System.out.println("The result of the decomposition is:");
         for(int i=2;i<=Math.sqrt(num);i++) {
             if(num%i == 0) {
                 System.out.print(i+"*");
                 num/=i;
                 i--;
             }
         }
         System.out.println(num);
     }
}
```

Figure 5.1: Program 1: decomposes a positive integer as prime factors.



```
import java.util.Scanner;
public class Bissextile{
     public static void main(String[] arge){
         System.out.print("Please enter the year: ");
         int year;
         Scanner scanner = new Scanner(System.in);
         year = scanner.nextInt();
         if (year > 0) {
           if ((year%4==0)&&(year%100!=0)||(year%400==0)) {
             System.out.println(year+" is bissextile.");
           }else {
             System.out.println(year+" is not bissextile.");
           }
         }else {
            System.out.println("Please enter a proper year!");
         }
     }
}
```

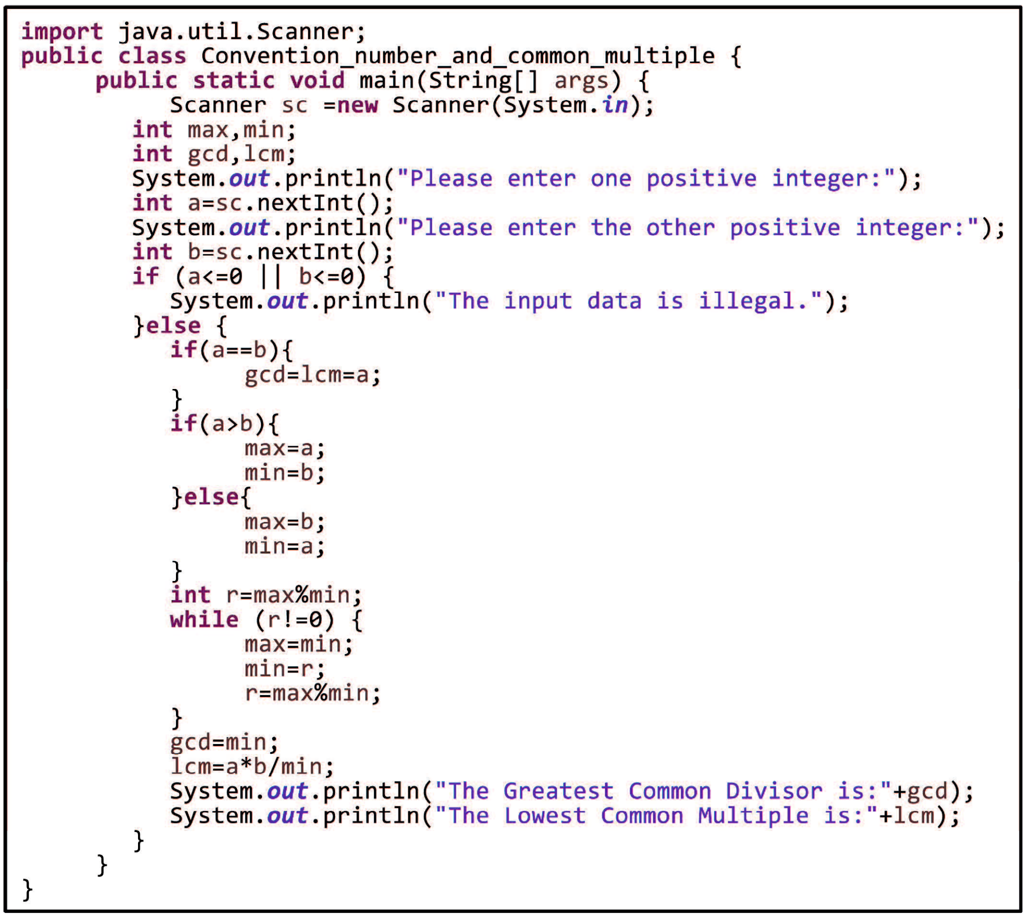
Figure 5.2: Program 2: determine if a year is a leap year.

5.2 Empirical Studies

5.2.1 Subnets Generation

We apply the algorithms proposed in Chapter 3 to Programs 1∼3 to show the possible usage of finding test input data in software testing. The program nets PN^1∼PN^3 constructed from Programs 1∼3 are shown in Figures 5.4∼5.6, respectively.

The detailed information of three program nets PN^1∼PN^3 is shown in Table 5.2. In PN^1, synchronous nodes are grouped by $SSW_1^1 = \{sw_1, sw_2, sw_3\}$ and $SSW_2^1 = \{sw_4\}$. Similarly, PN^2 contains synchronous nodes groups $SSW_1^2 = \{sw_1, sw_2, sw_3,$ $sw_4, sw_5\}$, $SSW_2^2 = \{sw_6\}$, $SSW_3^2 = \{sw_7\}$, and $SSW_4^2 = \{sw_8\}$. Among three



```
import java.util.Scanner;
public class Convention_number_and_common_multiple {
    public static void main(String[] args) {
        Scanner sc =new Scanner(System.in);
      int max,min;
      int gcd,lcm;
      System.out.println("Please enter one positive integer:");
      int a=sc.nextInt();
      System.out.println("Please enter the other positive integer:");
      int b=sc.nextInt();
      if (a<=0 || b<=0) {
        System.out.println("The input data is illegal.");
      }else {
        if(a==b){
            gcd=lcm=a;
        }
        if(a>b){
            max=a;
            min=b;
        }else{
            max=b;
            min=a;
        }
        int r=max%min;
        while (r!=0) {
            max=min;
            min=r;
            r=max%min;
        }
        gcd=min;
        lcm=a*b/min;
        System.out.println("The Greatest Common Divisor is:"+gcd);
        System.out.println("The Lowest Common Multiple is:"+lcm);
      }
    }
}
```

Figure 5.3: Program 3: Calculate the greatest common divisor and the least common multiple of two positive integers.

program nets, PN^3 is the most complex and it contains 5 synchronous nodes groups, $SSW_1^3 = \{sw_1, sw_2, sw_3\}$, $SSW_2^3 = \{sw_4, sw_5, sw_6\}$, $SSW_3^3 = \{sw_7, sw_8, sw_9\}$, SSW_4^3 $=\{sw_{10}, sw_{11}\}$ and SSW_5^3=$\{sw_{12}, sw_{13}\}$. The following steps are to generate subnets for PN^1~PN^3.

Table 5.2: The basic information of three program nets PN^1~PN^3.

Program Nets	Figures	NO. of SWITCH-nodes	NO. of Synchronous Nodes Groups
PN^1	Figure 5.4	4	2
PN^2	Figure 5.5	8	4
PN^3	Figure 5.6	13	5

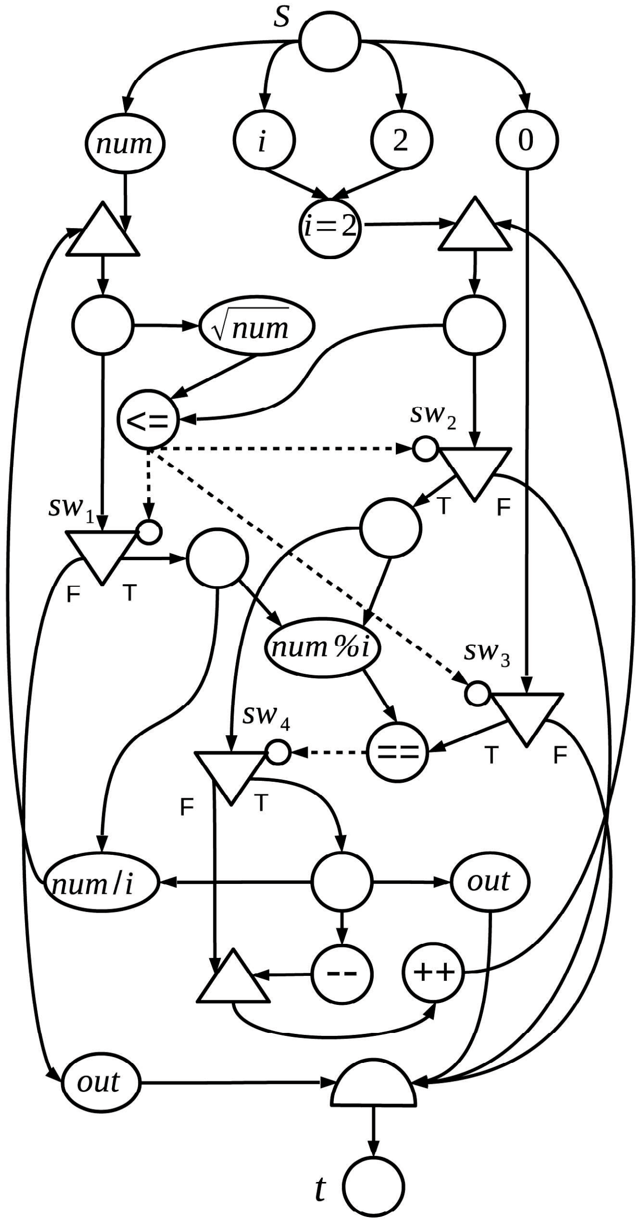


Figure 5.4: The program net PN^1 constructed from the program in Figure 5.1.

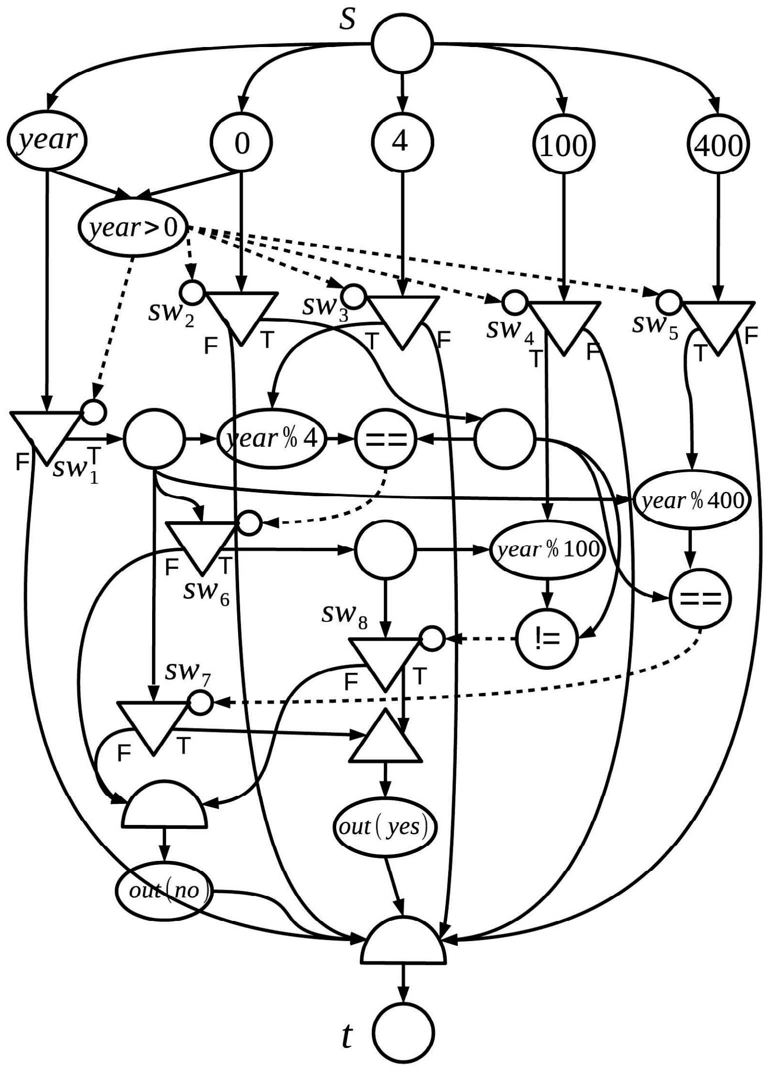


Figure 5.5: The program net PN^2 constructed from the program in Figure 5.2.

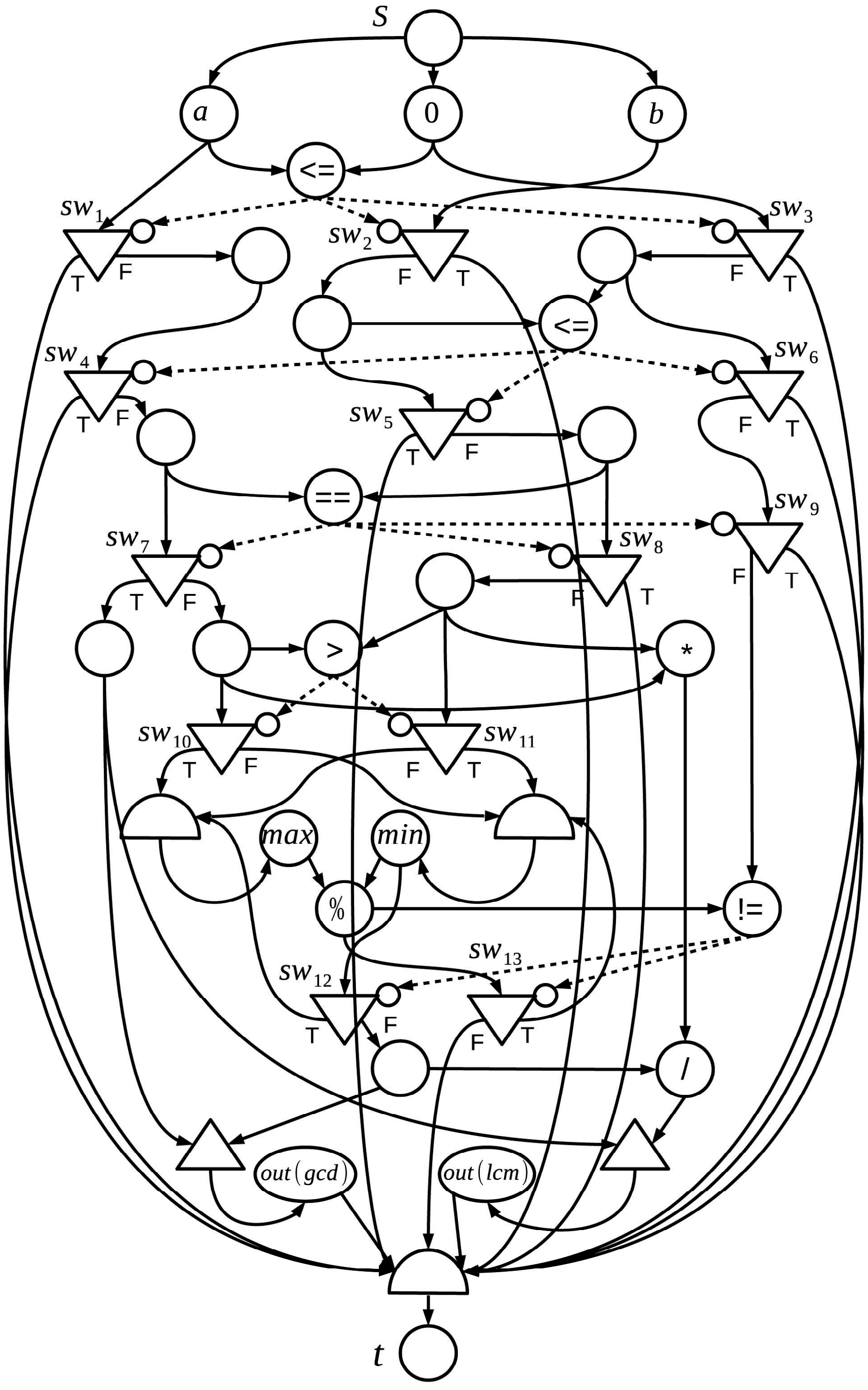


Figure 5.6: The program net PN^3 constructed from the program in Figure 5.3.

(1) Constructing layer nets of $PN^1 \sim PN^3$ by applying Algorithm 1, we can get three layer nets, $L_{PN^1} \sim L_{PN^3}$, as shown in Figures 5.7∼5.9, respectively. We can easily find all back edges (u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), (u_5, v_5), which are represented by bold black arrows in the two layer nets, L_{PN^1} and L_{PN^3}, respectively. Because L_{PN^2} does not contain back edge, PN^2 contains no directed circuit and is acyclic.

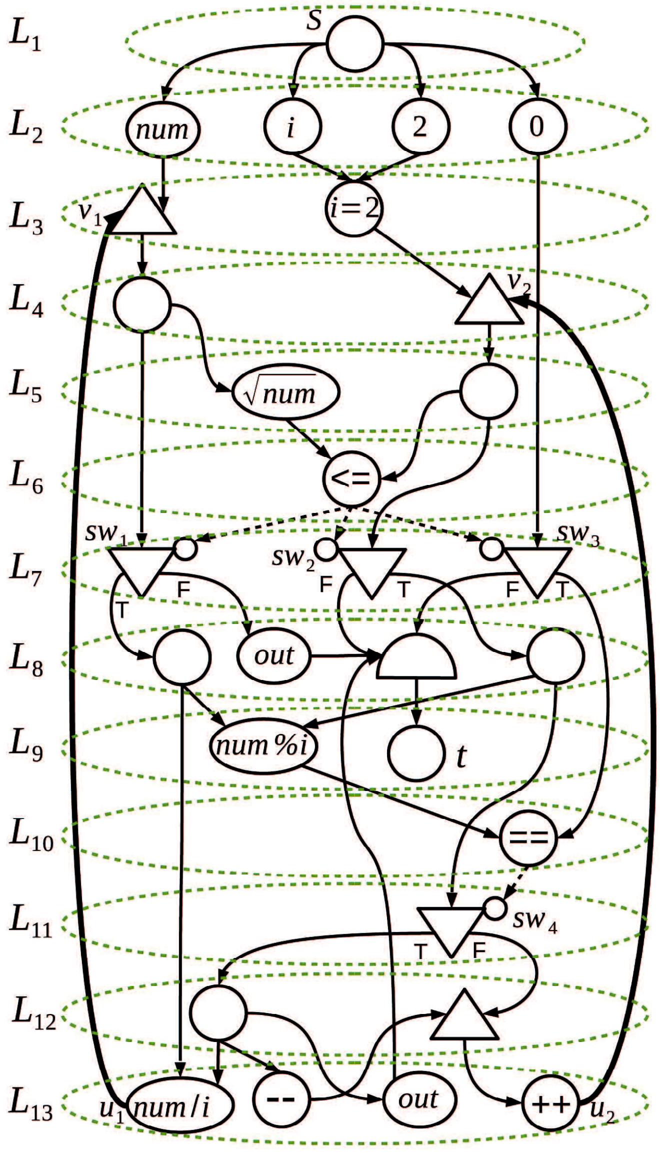


Figure 5.7: The constructed layer net L_{PN^1} of PN^1.

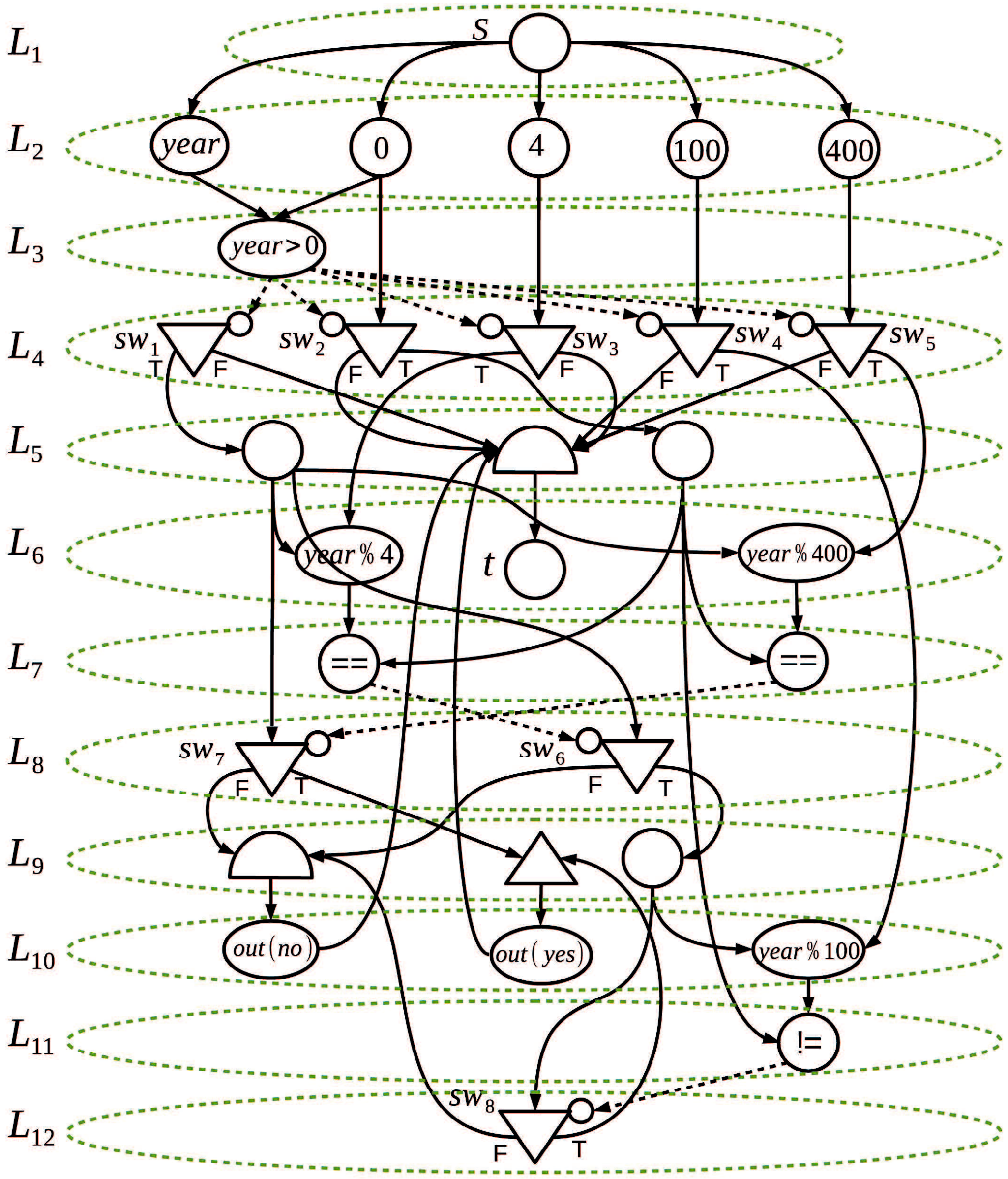


Figure 5.8: The constructed layer net L_{PN^2} of PN^2.

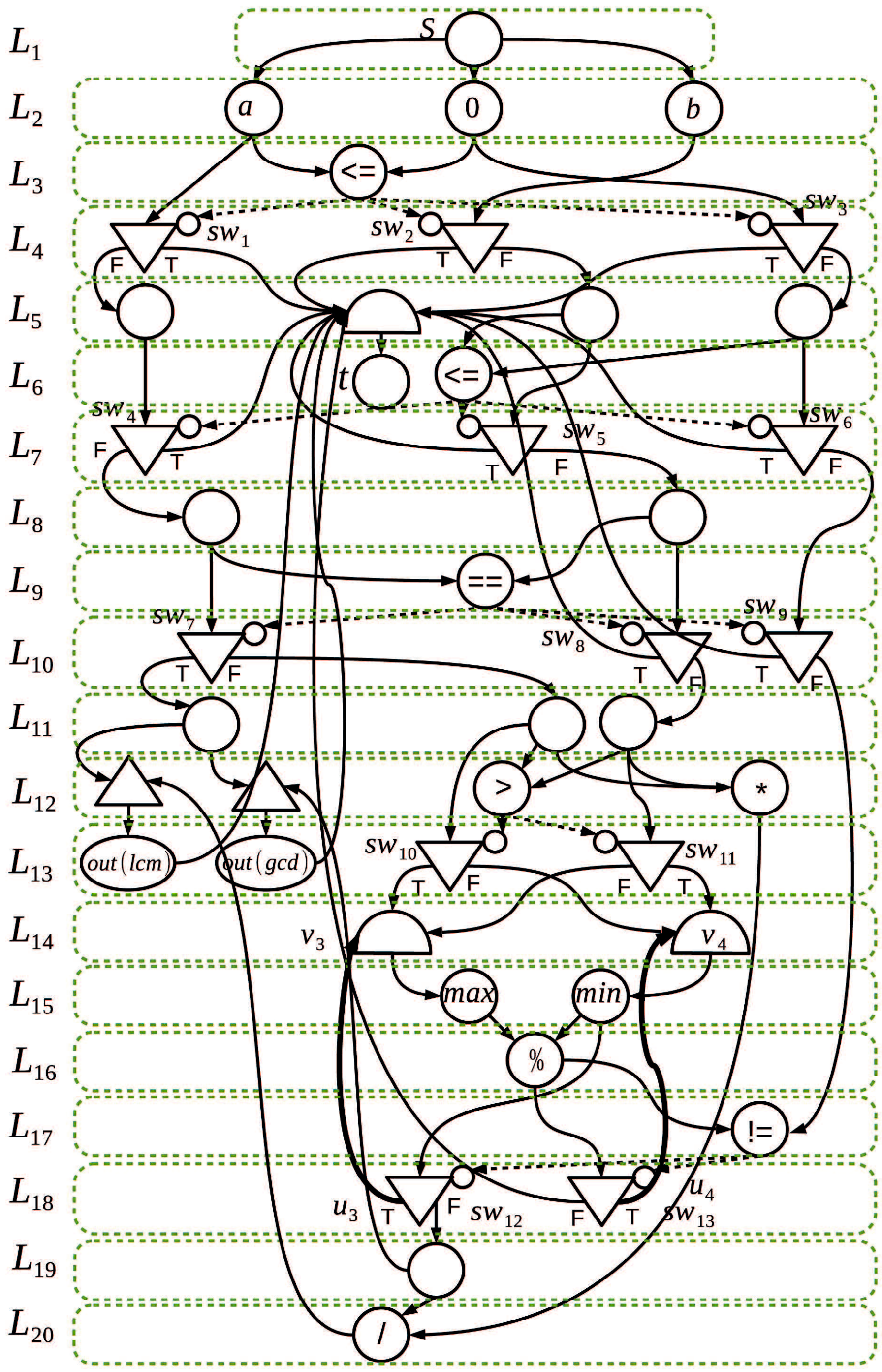


Figure 5.9: The constructed layer net L_{PN^3} of PN^3.

(2) To construct acyclic program nets of $PN^1 \sim PN^3$ by applying Algorithm 2. Because program nets PN^1 and PN^3 contain directed circuits, their acyclic program nets can be constructed by deleting all back edges in L_{PN^1} and L_{PN^3} respectively. As the result, acyclic program nets $\widehat{PN}^1$ and $\widehat{PN}^3$ of PN^1 and PN^3 are obtained as shown in Figure 5.10 and Figure 5.11. Because PN^2 is acyclic, its acyclic program net $\widehat{PN}^2$ is itself as shown in Figure 5.12.

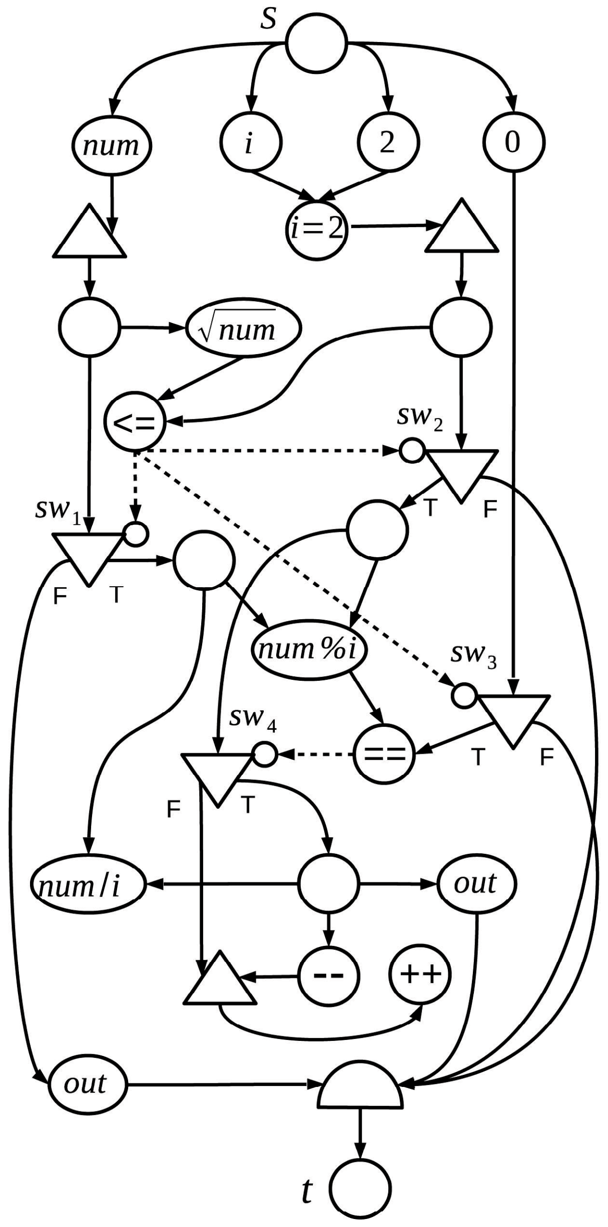


Figure 5.10: The acyclic program net $\widehat{PN}^1$ of PN^1.

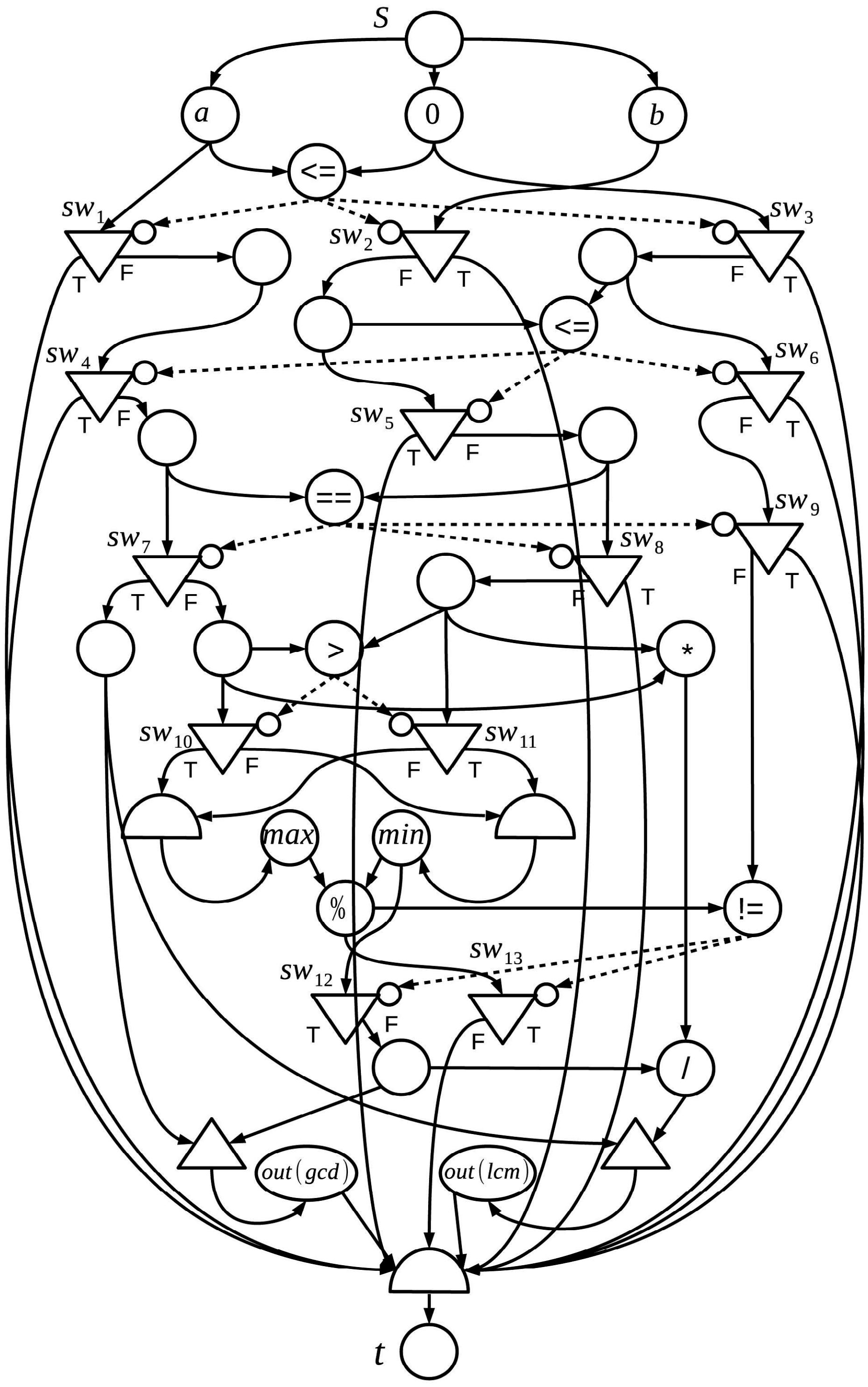


Figure 5.11: The acyclic program net $\widehat{PN}^3$ of PN^3.

]

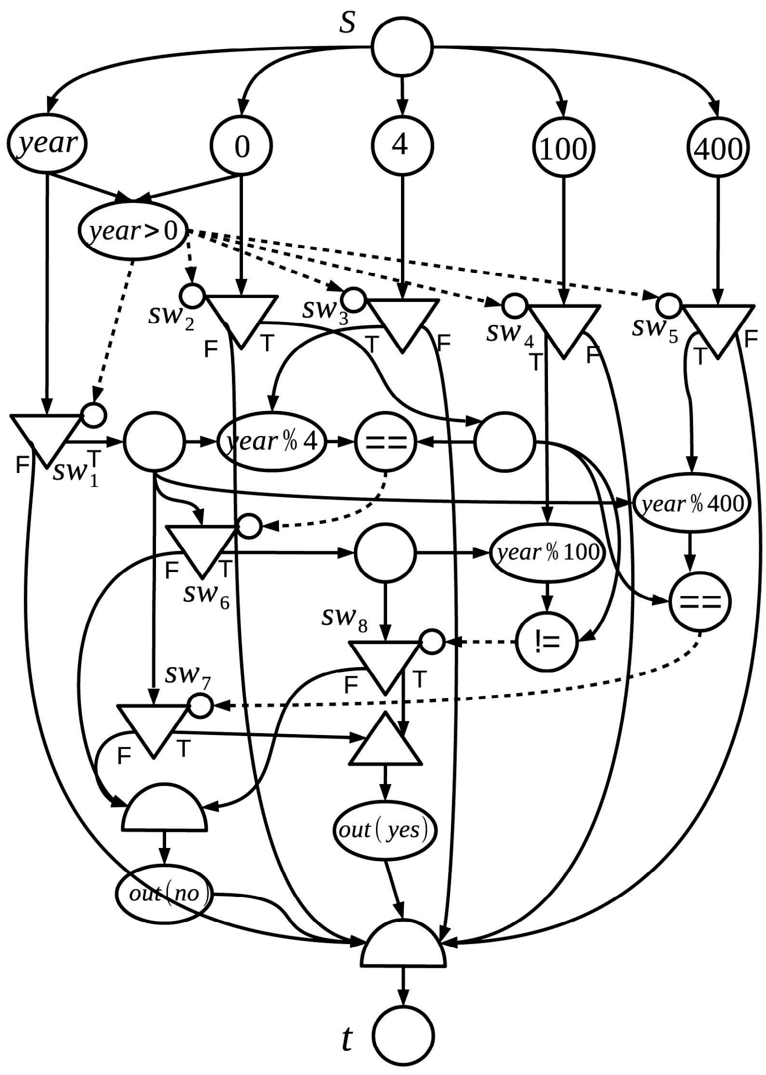


Figure 5.12: The acyclic program net $\widehat{PN}^2$ of PN^2.

(3) This step is to apply Algorithm 3 to generate initial subnets for $\widehat{PN}^1 \sim \widehat{PN}^3$, respectively. Firstly, we set $c_1=c_2=c_3=1$, $c_4=c_5=c_6=0$, $c_7=c_8=c_9=1$, $c_{10}=c_{11}=0$ and $c_{12}=c_{13}=1$ respectively for the SWITCH-nodes in SSW_1^3, SSW_2^3, SSW_3^3, SSW_4^3 and SSW_5^3. Then all unfirable nodes of $\widehat{PN}^3$ are deleted according to the states initially set for all SWITCH-nodes. Finally, the remaining parts of $\widehat{PN}^3$ is the initial subnet $\widehat{PN}_1^3$ as shown in Figure 5.13 (a) and its state is $\psi_1^3=(1,1,1,*,*,*,*,*,*,*,*,*,*)$. Similarly, the initial subnets $\widehat{PN}_1^1$ and $\widehat{PN}_1^2$ can be generated as shown in Figure 5.14 (a) and Figure 5.15 (a), respectively.

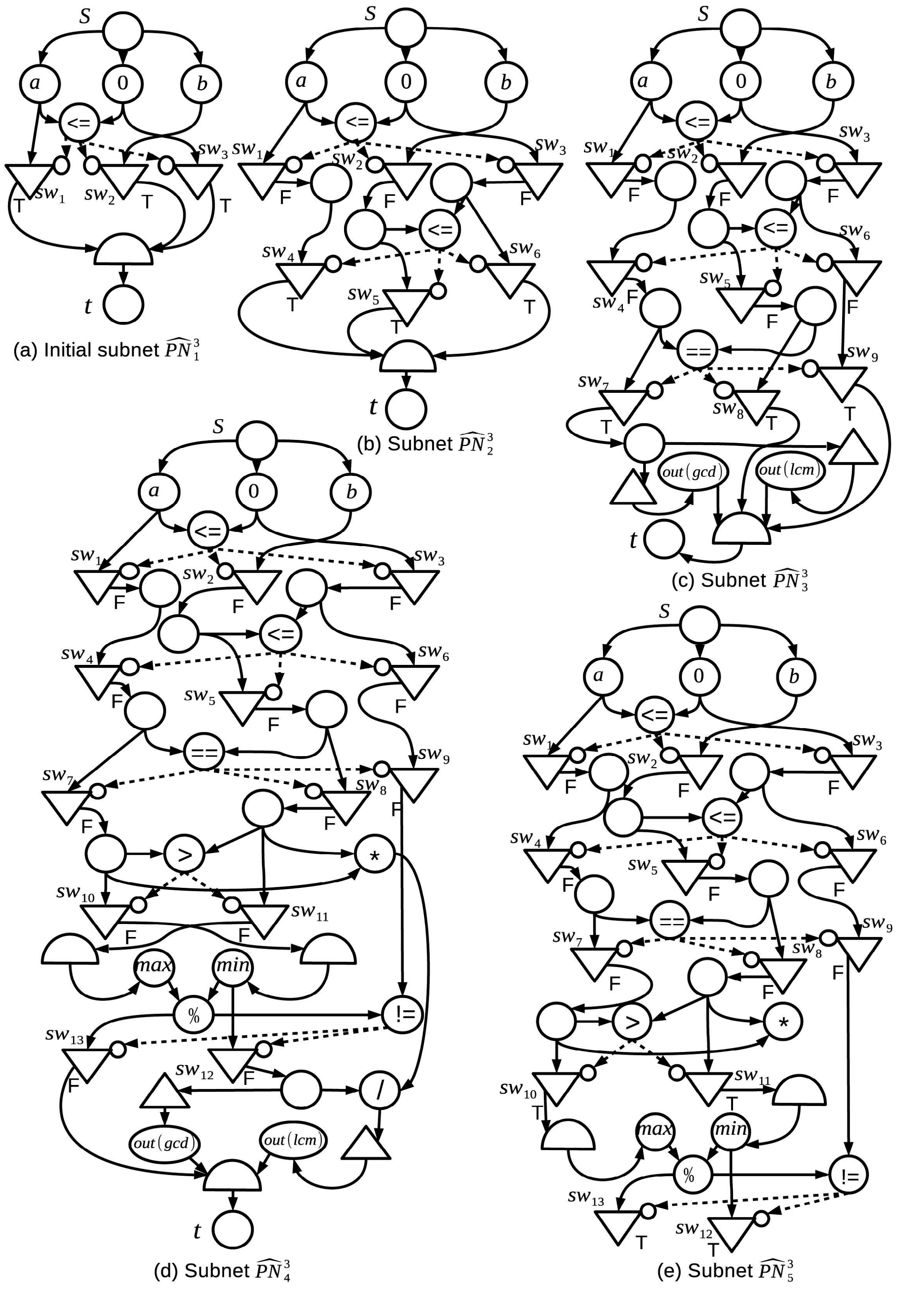
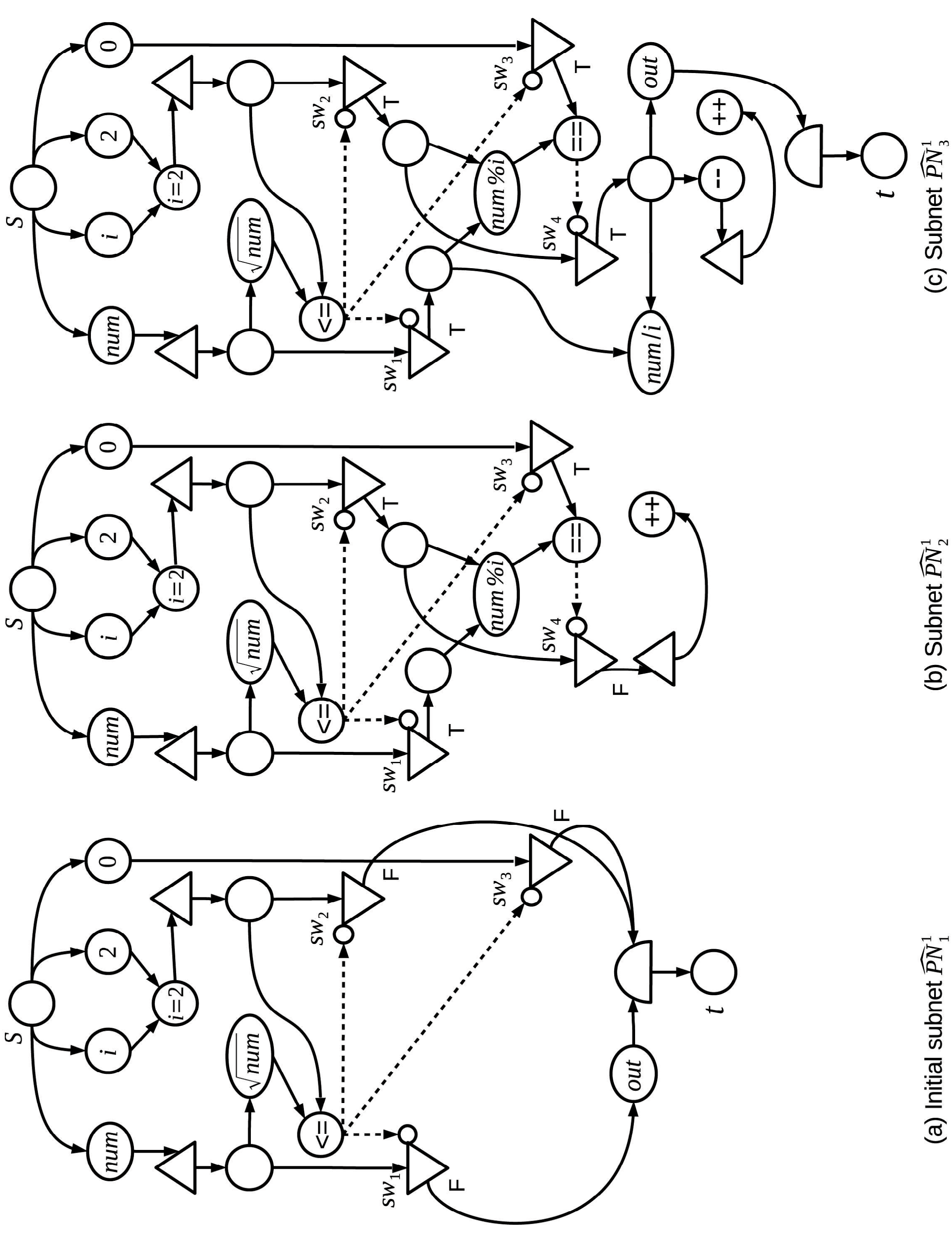
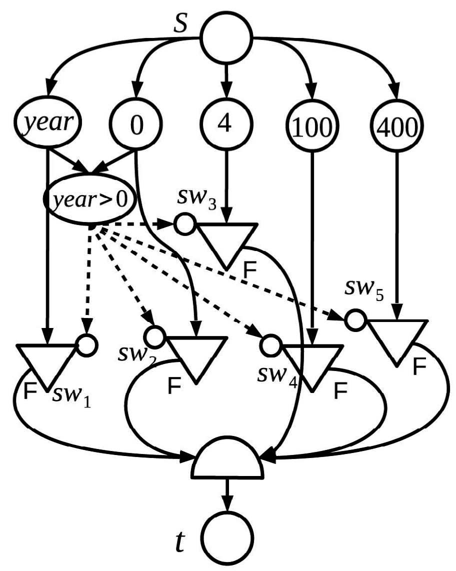

(a) Initial subnet $\widehat{PN}_1^3$
(b) Subnet $\widehat{PN}_2^3$
(c) Subnet $\widehat{PN}_3^3$
(d) Subnet $\widehat{PN}_4^3$
(e) Subnet $\widehat{PN}_5^3$
s
t
a
0
b
<=
==
sw1
sw2
sw3
sw4
sw5
sw6
sw7
sw8
sw9
sw10
sw11
sw12
sw13
T
F
out(gcd)
out(lcm)
max
min
%
!=
>
*
/

Figure 5.13: The subnets set CSN^3 of $\widehat{PN}^3$.

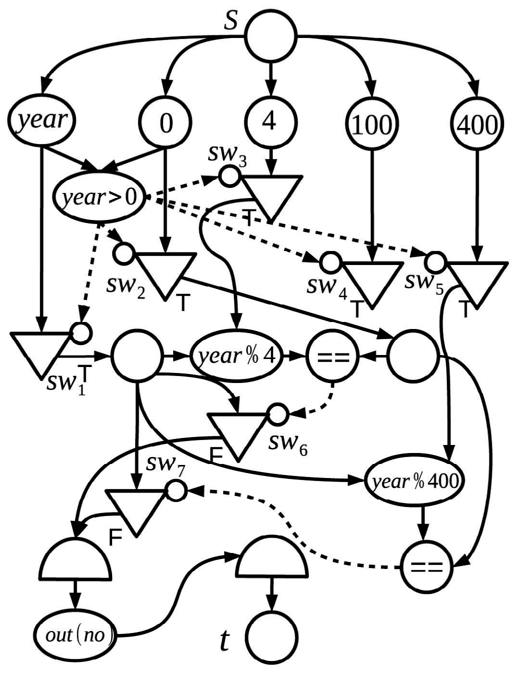


(a) Initial subnet $\widehat{PN}_1^1$
(b) Subnet $\widehat{PN}_2^1$
(c) Subnet $\widehat{PN}_3^1$
s
num
i
2
0
i=2
$\sqrt{num}$
<=
sw_1
sw_2
sw_3
sw_4
T
F
num%i
==
out
num/i
++
--
t

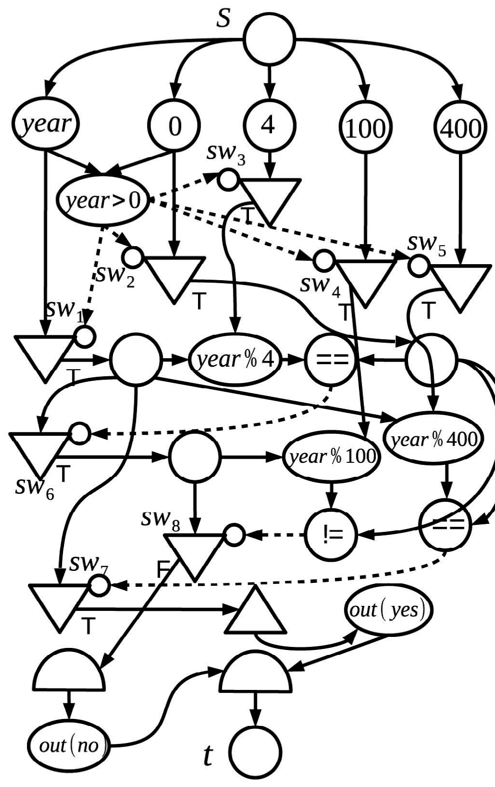
Figure 5.14: The subnets set CSN^1 of $\widehat{PN}^1$.



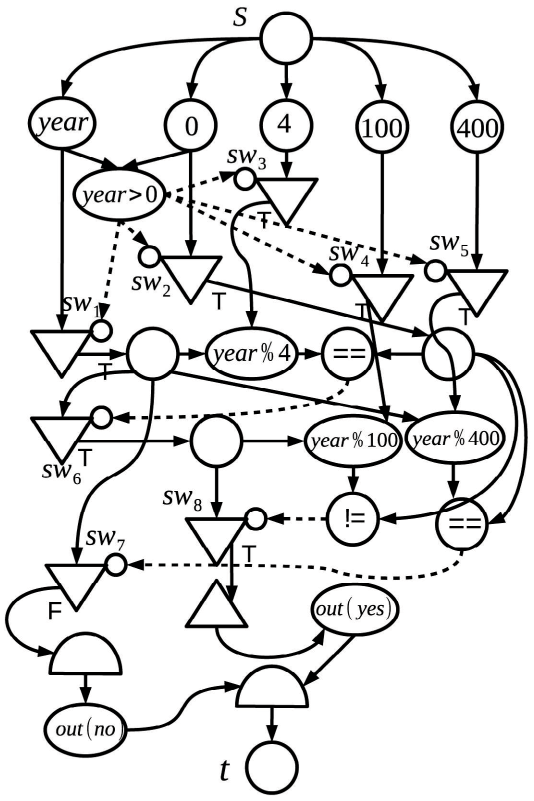
(a) Initial subnet $\widehat{PN}_1^2$



(b) Subnet $\widehat{PN}_2^2$



(c) Subnet $\widehat{PN}_3^2$



(d) Subnet $\widehat{PN}_4^2$

Figure 5.15: The subnets set CSN^2 of $\widehat{PN}^2$.

(4) This is the final step to generate other subnets through applying Algorithm 4 based on the obtained initial subnets in (3). For example, generate remaining subnets for $\widehat{PN}^3$ based on $\widehat{PN}^3_1$ and ψ^3_1=(1, 1, 1, ∗, ∗, ∗, ∗, ∗, ∗, ∗, ∗, ∗, ∗). Firstly, by referring $\tilde{\psi}^3$=ψ^3_1=(1, 1, 1, ∗, ∗, ∗, ∗, ∗, ∗, ∗, ∗, ∗, ∗) and the SWITCH-nodes in SSW^3_1 that are located in the lowest layer among all SWITCH-nodes, we set c_i for each sw_i in SSW^3_1 with $\tilde{c}_i \neq 2$ to possibly let $c_i \odot \tilde{c}_i = 2$. As the result, $c_1=c_2=c_3=0$ is set and a subnet $\widehat{PN}^3_2$ shown in Figure 5.13 (b) is obtained by using Algorithm 4, which has state ψ^3_2=(0, 0, 0, 1, 1, 1, ∗, ∗, ∗, ∗, ∗, ∗, ∗). Thus current $\tilde{\psi}^3$ and CSN^3 are such that $\tilde{\psi}^3 \leftarrow \tilde{\psi}^3 \odot \psi^3_2$=(2, 2, 2, 1, 1, 1, ∗, ∗, ∗, ∗, ∗, ∗, ∗) and $CSN^3=\{\widehat{PN}^3_1, \widehat{PN}^3_2\}$.

Next, we choose SSW^3_2 and set c_i for each its sw_i with $\tilde{c}_i \neq 2$ according to the layer number of SWITCH-nodes in SSW^3_2. $c_4=c_5=c_6=0$ is set and $\widehat{PN}^3_3$ is obtained as shown in Figure 5.13 (c). Resultantly, the SWITCH-node states of $\widehat{PN}^3_3$ is ψ^3_3 = (0, 0, 0, 0, 0, 0, 1, 1, 1, ∗, ∗, ∗, ∗), $\tilde{\psi} \leftarrow \tilde{\psi} \odot \psi^3_3$ = (2, 2, 2, 2, 2, 2, 1, 1, 1, ∗, ∗, ∗, ∗) and $CSN^3=\{\widehat{PN}^3_1, \widehat{PN}^3_2, \widehat{PN}^3_3\}$ hold.

Continuously, we choose SSW^3_3 and set c_i for each its sw_i with $\tilde{c}_i \neq 2$. $c_7=c_8=c_9=0$ is set and $\widehat{PN}^3_4$ is obtained as shown in Figure 5.13 (d). The SWITCH-node states of $\widehat{PN}^3_4$ is ψ^3_4=(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), and consequently $\tilde{\psi} \leftarrow \tilde{\psi} \odot \psi^3_4$=(2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0) and $CSN^3=\{\widehat{PN}^3_1, \widehat{PN}^3_2, \widehat{PN}^3_3, \widehat{PN}^3_4\}$.

Finally, we choose SSW^3_4 and set c_i for each its sw_i with $\tilde{c}_i \neq 2$. $c_10=c_{11}=1$ is set and $\widehat{PN}^3_5$ is obtained as shown in Figure 5.13 (e). The SWITCH-node states of $\widehat{PN}^3_5$ is ψ^3_5=(0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1), and consequently $\tilde{\psi} \leftarrow \tilde{\psi} \odot \psi^3_5$=(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2) and $CSN^3=\{\widehat{PN}^3_1, \widehat{PN}^3_2, \widehat{PN}^3_3, \widehat{PN}^3_4, \widehat{PN}^3_5\}$. Since no SWITCH-node has $\tilde{c}_i \neq 2$, CSN^3 is the required subnets set of $\widehat{PN}^3$. Obviously all the nodes of $\widehat{PN}^3$ are included in at least one of the subnets included in CSN^3.

Similarly, applying Algorithm 4 to $\widehat{PN}^1$ and $\widehat{PN}^2$, we can get subnets set CSN^1 and CSN^2 shown in Figure 5.14 and Figure 5.15, respectively.

5.2.2 Test Data Generation

We apply the algorithms proposed in Chapter 4 and $Z3$ prover to find a test input data for each subnet in CSN^i $(i = 1, 2, 3)$ shown in Figures 5.14∼5.13.

(1) To make an exhaustive subnet for each subnet by adding dynamic behavior information to it as stated in 4.2.1. As the result, the corresponding exhaustive subnets set $CESN^i$ of CSN^i $(i = 1, 2, 3)$ can be obtained as shown in Figures 5.16∼5.18.

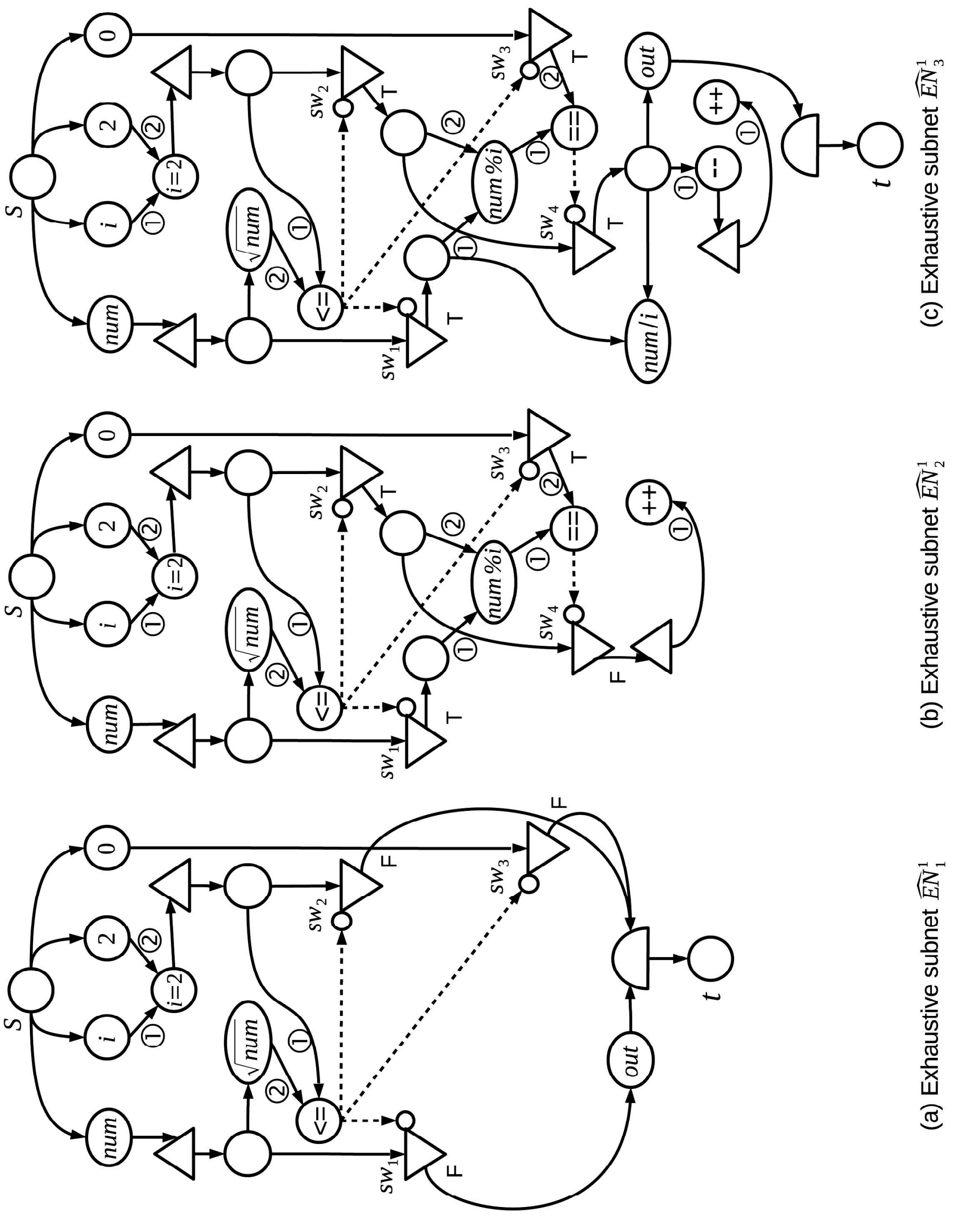
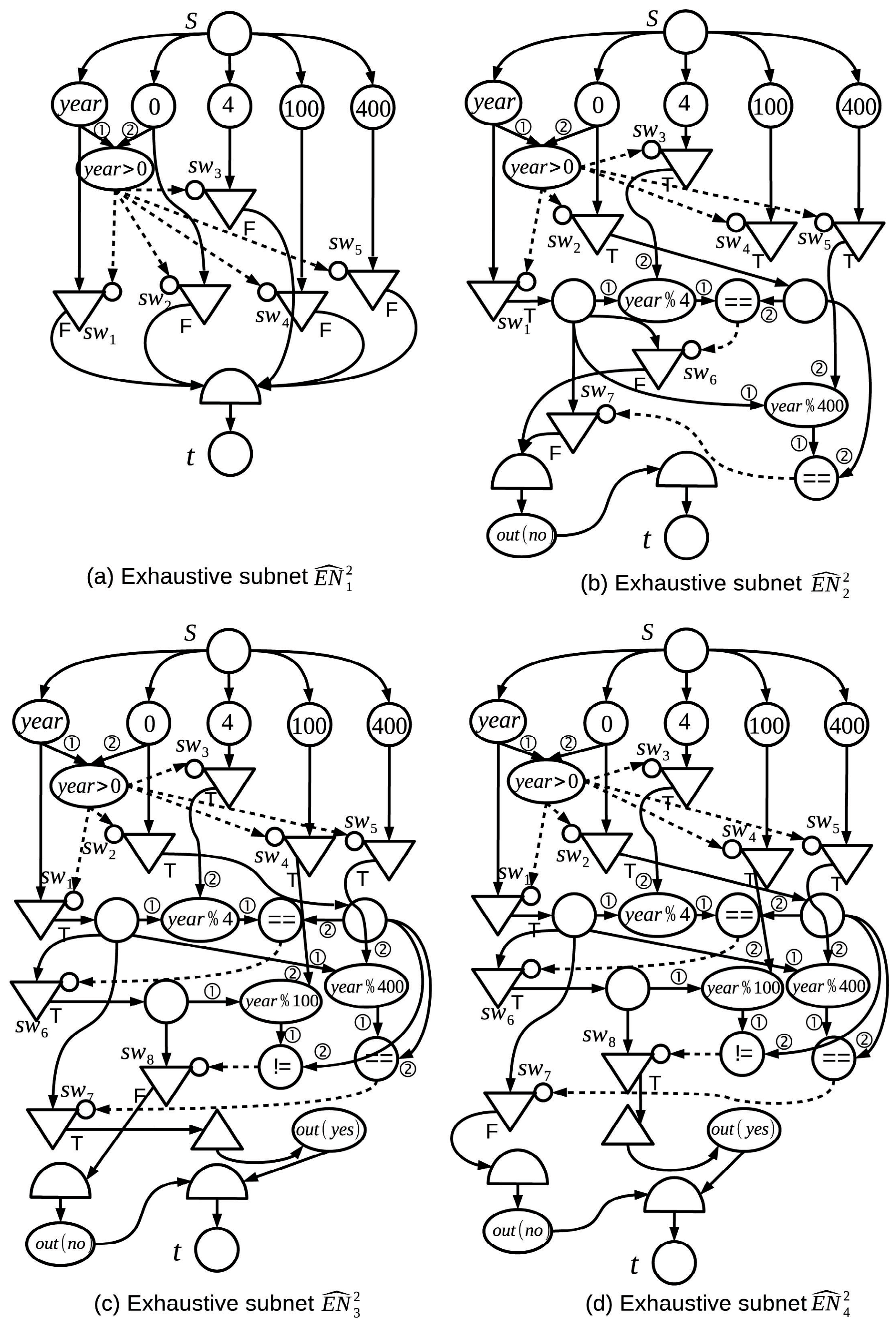

(a) Exhaustive subnet $\widehat{EN}_1^1$
(b) Exhaustive subnet $\widehat{EN}_2^1$
(c) Exhaustive subnet $\widehat{EN}_3^1$
S
t
num
i
2
0
i=2
$\sqrt{num}$
<=
num%i
==
out
num/i
++
--
sw_1
sw_2
sw_3
sw_4
T
F

Figure 5.16: The exhaustive subnets set $CESN^1$ of $\widehat{PN}^1$.



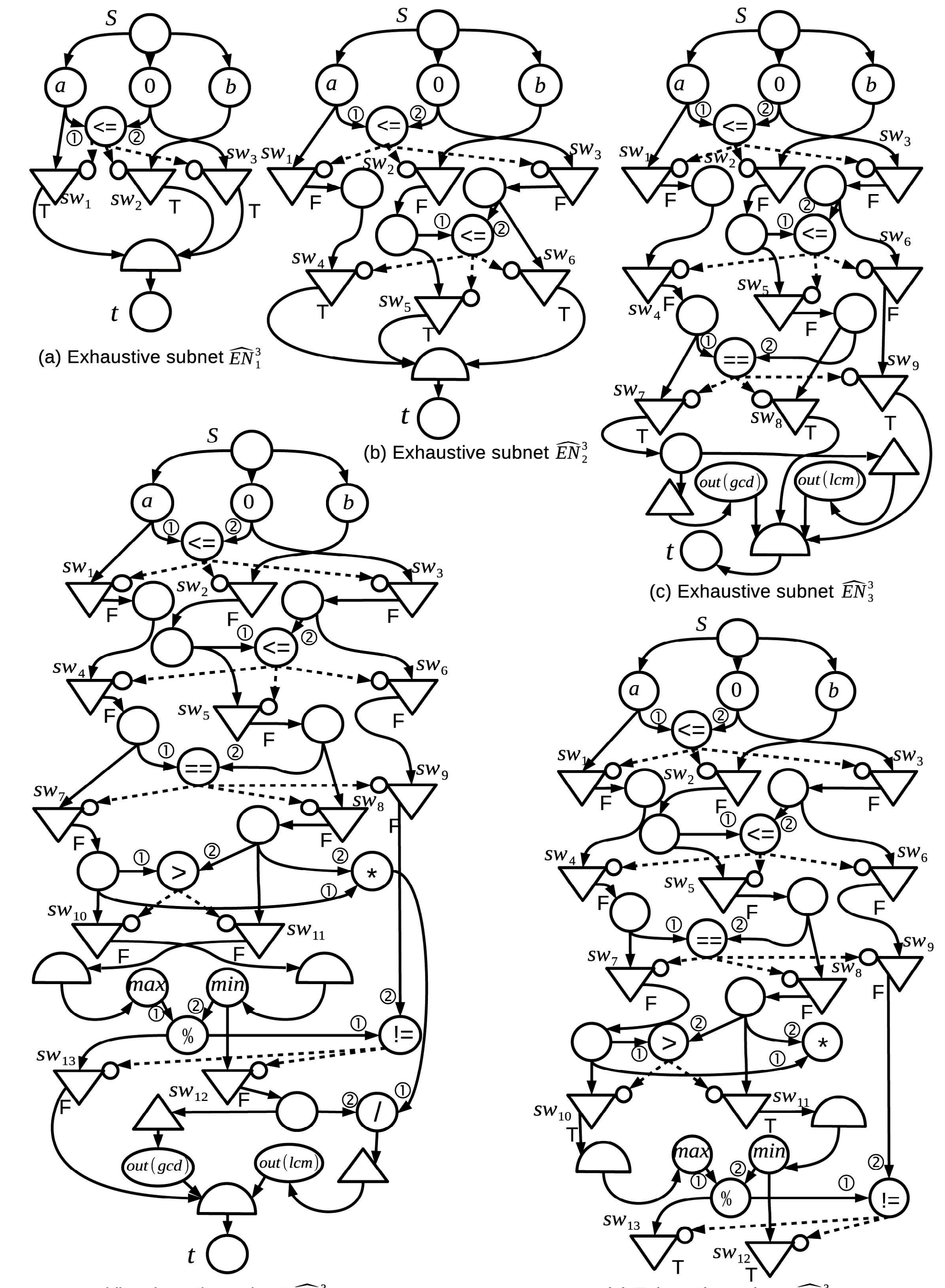
(a) Exhaustive subnet $\widehat{EN}_1^2$

(b) Exhaustive subnet $\widehat{EN}_2^2$

(c) Exhaustive subnet $\widehat{EN}_3^2$

(d) Exhaustive subnet $\widehat{EN}_4^2$

Figure 5.17: The exhaustive subnets set $CESN^2$ of $\widehat{PN}^2$.



(a) Exhaustive subnet $\widehat{EN}_1^3$

(b) Exhaustive subnet $\widehat{EN}_2^3$

(c) Exhaustive subnet $\widehat{EN}_3^3$

(d) Exhaustive subnet $\widehat{EN}_4^3$

(e) Exhaustive subnet $\widehat{EN}_5^3$

Figure 5.18: The exhaustive subnets set $CESN^3$ of $\widehat{PN}^3$.

(2) To make an adjacency matrix for each exhaustive subnet in order to facilitate the generation of constraint conditions. For example, the following adjacency matrix A_4^3 is made for $\widehat{EN}_4^3$ and contains all information of $\widehat{EN}_4^3$ as shown in Figure 5.18 (e).

$$A_4^3 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -14 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -14 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

(3) Next, to apply Algorithm 5 to determine boolean value of the corresponding constraint condition for each SWITCH-node and delete sink nodes iteratively until all the sink nodes become SWITCH-nodes to get a simplified exhaustive subnet. For example, the simplified subnet $\widetilde{EN}_4^3$ as shown in Figure 5.19 (d) is constructed from $\widehat{EN}_4^3$. As the result, all the simplified nets in $CSN^1 \sim CSN^3$ are shown in Figures 5.19$\sim$5.21, respectively.

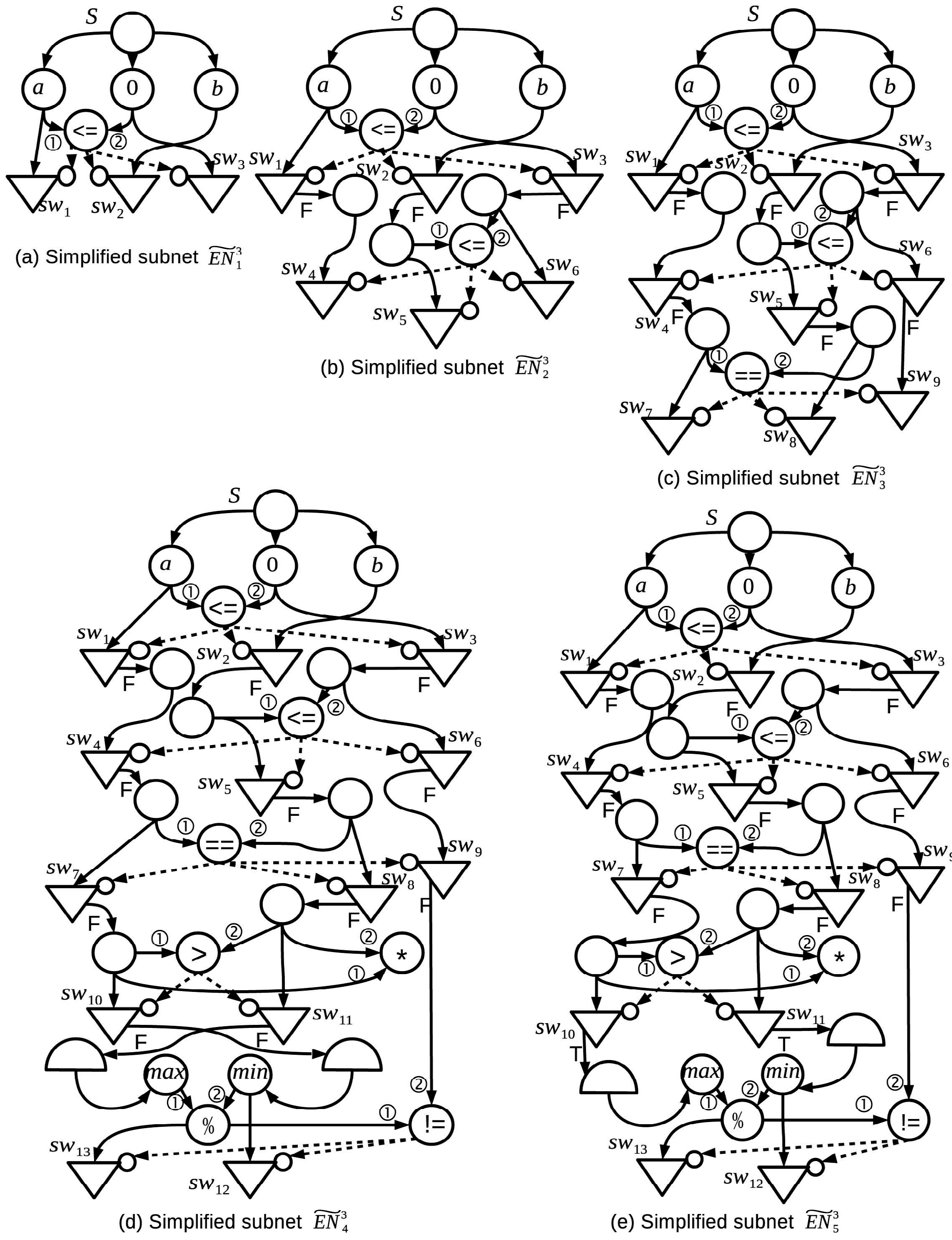
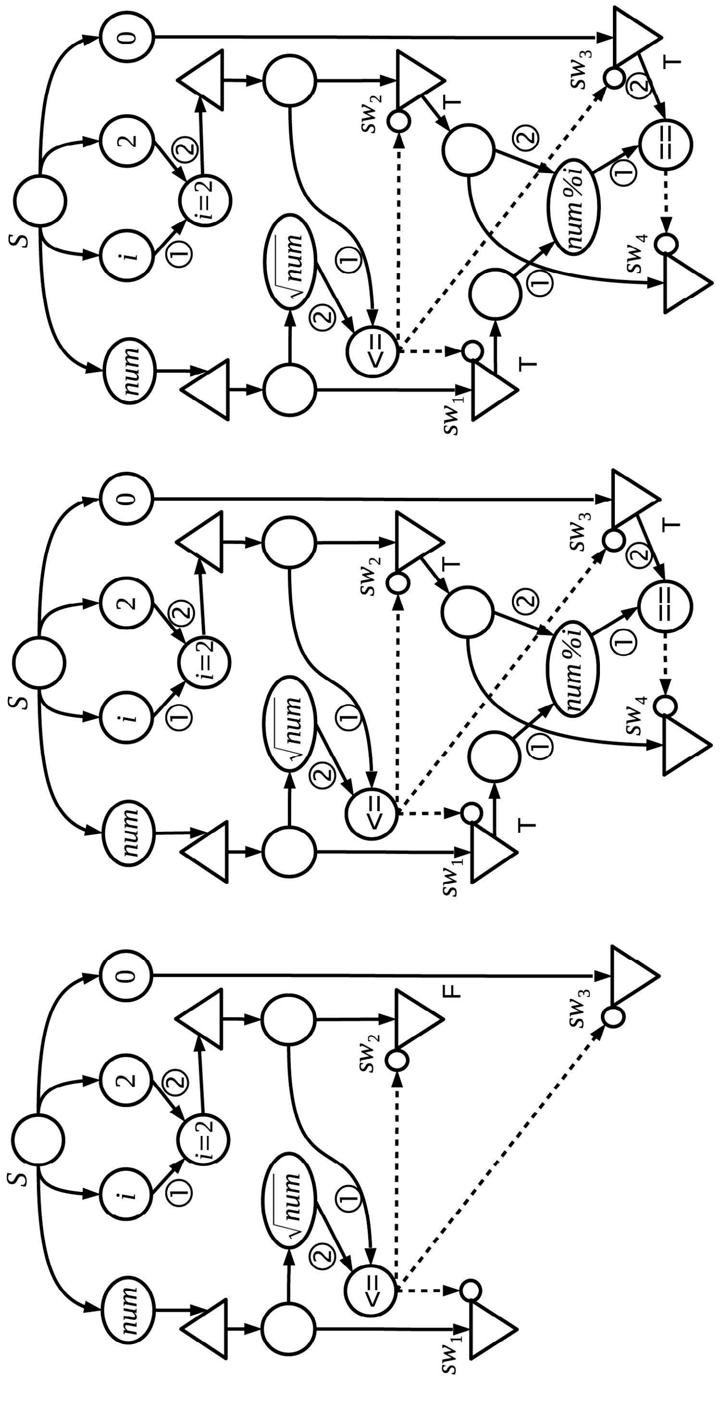

(a) Simplified subnet $\widetilde{EN}_1^3$
(b) Simplified subnet $\widetilde{EN}_2^3$
(c) Simplified subnet $\widetilde{EN}_3^3$
(d) Simplified subnet $\widetilde{EN}_4^3$
(e) Simplified subnet $\widetilde{EN}_5^3$

Figure 5.19: The simplified exhaustive subnets set $\widetilde{CESN}^3$ of CSN^3.

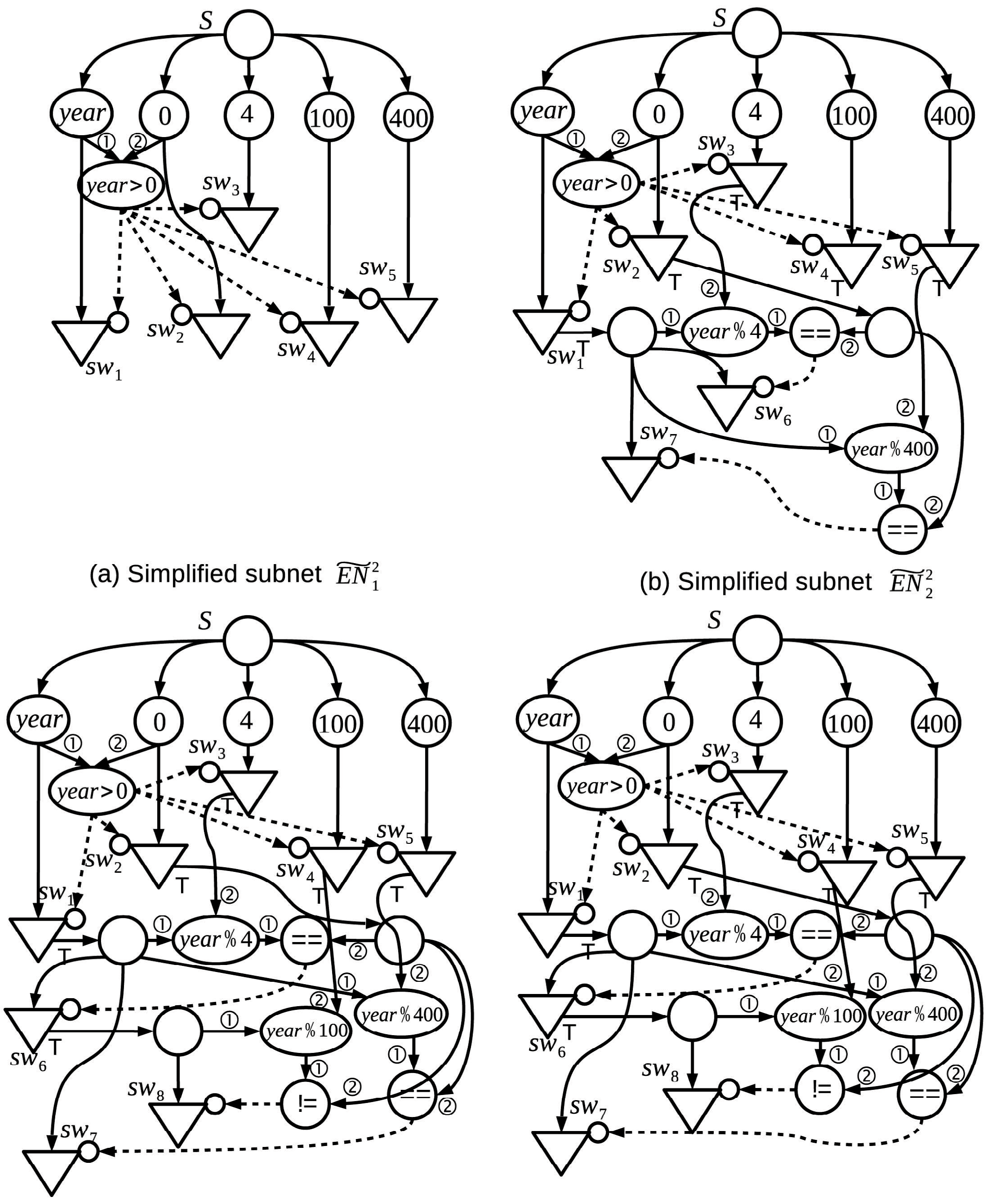


(a) Simplified subnet $\widetilde{EN}_1^1$

(b) Simplified subnet $\widetilde{EN}_2^1$

(c) Simplified subnet $\widetilde{EN}_3^1$

Figure 5.20: The simplified exhaustive subnets set $\widetilde{CESN}^1$ of CSN^1.



(a) Simplified subnet $\widetilde{EN}_1^2$

(b) Simplified subnet $\widetilde{EN}_2^2$

(c) Simplified subnet $\widetilde{EN}_3^2$

(d) Simplified subnet $\widetilde{EN}_4^2$

Figure 5.21: The simplified exhaustive subnets set $\widetilde{CESN}^2$ of CSN^2.

(4) This step is to apply Algorithm 6 to obtain constraint conditions for each simplified exhaustive subnet in $\widetilde{CESN}^1 \sim \widetilde{CESN}^3$. For example, from simplified exhaustive subnet $\widetilde{EN}_4^3$ and its adjacency matrix $\tilde{A}_3^3$ obtained by Algorithm 5, we can get its constraint conditions CCS_5^3=$\{\neg(a\leq 0), \neg(b\leq 0), \neg(a{==}b), \neg(a>b), \neg(b\%a!=0)\}$ and these constraint conditions can be expressed as a conjunctive normal form F_4^3 : $\neg(a\leq 0)\wedge\neg(b\leq 0)\wedge\neg(a{==}b)\wedge \neg(a>b)\wedge\neg(b\%a!=0)$. Similarly, all the constraint conditions of other simplified exhaustive subnets are obtained and shown in Tables 5.3∼5.5, respectively.

$$\tilde{A}_3^3 = \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 2 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -14 & 0 & 2 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -14 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

Table 5.3: Constraint conditions of exhaustive subnets set $CESN^1$.

Exhaustive Subnets	Simplified Subnets	Constraint Conditions
$\widehat{EN}_1^1$	$\widetilde{EN}_1^1$	$\neg(2\leq sqrt(num))$
$\widehat{EN}_2^1$	$\widetilde{EN}_1^1$	$(2\leq sqrt(num))\wedge(\neg(num\%2{==}0))$
$\widehat{EN}_3^1$	$\widetilde{EN}_1^1$	$(2\leq sqrt(num))\wedge(num\%2{==}0)$

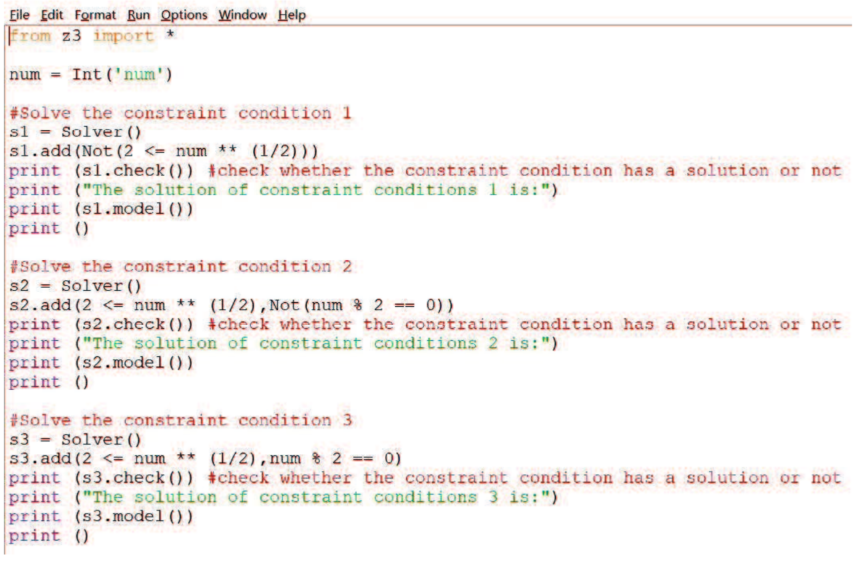
(5) Finally, to generate test input data for all the exhaustive subnets based on the obtained constraint conditions so far by using $Z3$ prover. Importing $Z3$ prover to Python and coding three short programs shown in Figures 5.22∼5.24 for finding test input data, three sets of test input data for three sets of exhaustive subnets are obtained and shown in Tables 5.6∼5.8, respectively.

Table 5.4: Constraint conditions of exhaustive subnets set $CESN^2$.

Exhaustive Nets	Simplified Nets	Constraint Conditions
$\widehat{EN}_1^2$	$\widetilde{EN}_1^2$	$\neg(year>0)$
$\widehat{EN}_2^2$	$\widetilde{EN}_2^2$	$(year>0)\wedge(\neg(year\%4==0))\wedge(\neg(year\%400==0))$
$\widehat{EN}_3^2$	$\widetilde{EN}_3^2$	$(year>0)\wedge((year\%4==0))\wedge(\neg(year\%100!=0))$ $\wedge((year\%400==0))$
$\widehat{EN}_4^2$	$\widetilde{EN}_4^2$	$(year>0)\wedge((year\%4==0))\wedge((year\%100!=0))$ $\wedge(\neg(year\%400==0))$

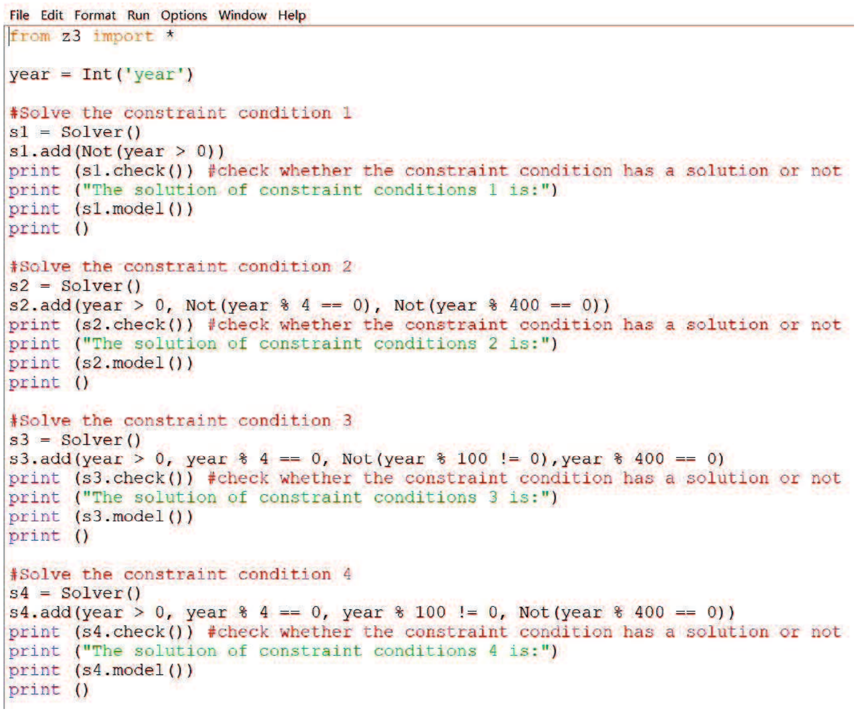
Table 5.5: Constraint conditions of exhaustive subnets set $CESN^3$.

Exhaustive Subnets	Simplified Subnets	Constraint Conditions
$\widehat{EN}_1^3$	$\widetilde{EN}_1^3$	$(a\leq 0)$
$\widehat{EN}_2^3$	$\widetilde{EN}_2^3$	$(\neg(a\leq 0))\wedge(b\leq 0)$
$\widehat{EN}_3^3$	$\widetilde{EN}_3^3$	$(\neg(a\leq 0))\wedge(\neg(b\leq 0))\wedge(a==b)$
$\widehat{EN}_4^3$	$\widetilde{EN}_4^3$	$\neg(a\leq 0)\wedge\neg(b\leq 0)\wedge\neg(a==b)\wedge\neg(a>b)\wedge\neg(b\%a!=0)\}$
$\widehat{EN}_5^3$	$\widetilde{EN}_5^3$	$\neg(a\leq 0)\wedge\neg(b\leq 0)\wedge\neg(a==b)\wedge(a>b)\wedge(a\%b!=0)\}$



```
from z3 import *

num = Int('num')

#Solve the constraint condition 1
s1 = Solver()
s1.add(Not(2 <= num ** (1/2)))
print (s1.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 1 is:")
print (s1.model())
print ()

#Solve the constraint condition 2
s2 = Solver()
s2.add(2 <= num ** (1/2),Not(num % 2 == 0))
print (s2.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 2 is:")
print (s2.model())
print ()

#Solve the constraint condition 3
s3 = Solver()
s3.add(2 <= num ** (1/2),num % 2 == 0)
print (s3.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 3 is:")
print (s3.model())
print ()
```

Figure 5.22: The short program to find a set of test data for $\widetilde{CESN}^1$.



```
from z3 import *

year = Int('year')

#Solve the constraint condition 1
s1 = Solver()
s1.add(Not(year > 0))
print (s1.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 1 is:")
print (s1.model())
print ()

#Solve the constraint condition 2
s2 = Solver()
s2.add(year > 0, Not(year % 4 == 0), Not(year % 400 == 0))
print (s2.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 2 is:")
print (s2.model())
print ()

#Solve the constraint condition 3
s3 = Solver()
s3.add(year > 0, year % 4 == 0, Not(year % 100 != 0),year % 400 == 0)
print (s3.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 3 is:")
print (s3.model())
print ()

#Solve the constraint condition 4
s4 = Solver()
s4.add(year > 0, year % 4 == 0, year % 100 != 0, Not(year % 400 == 0))
print (s4.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 4 is:")
print (s4.model())
print ()
```

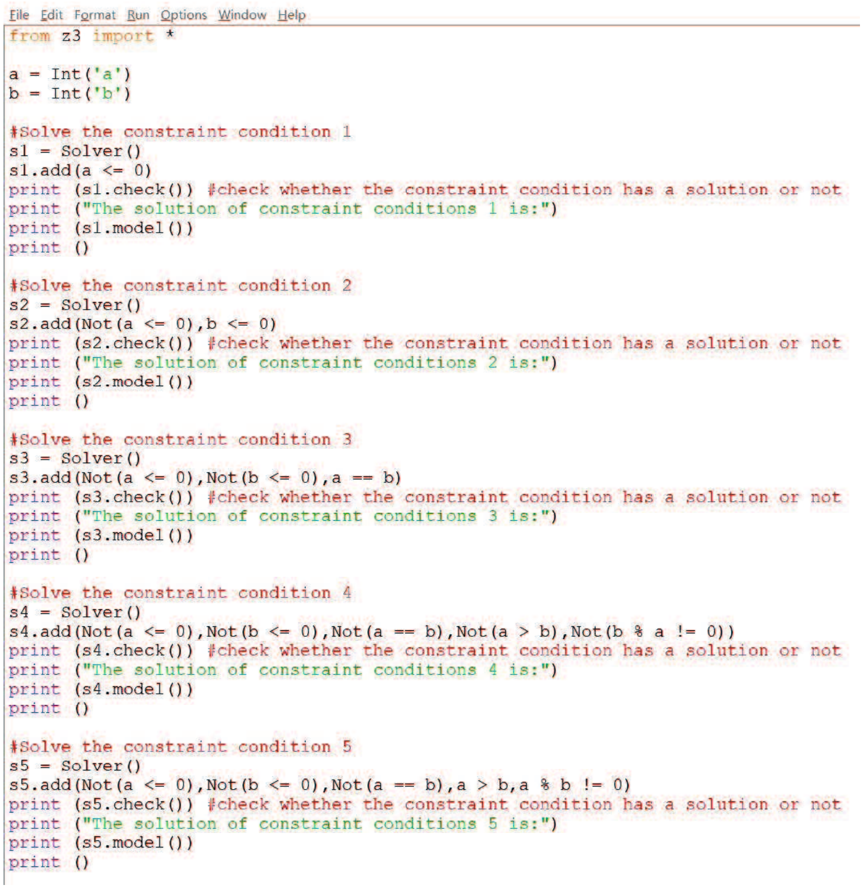
Figure 5.23: The short program to find a set of test data for $\widetilde{CESN}^2$.

Table 5.6: Test Input Data of exhaustive subnets set $CESN^1$.

Exhaustive Subnets	Simplified Subnets	Test Input Data
$\widehat{EN}_1^1$	$\widetilde{EN}_1^1$	num=3
$\widehat{EN}_2^1$	$\widetilde{EN}_1^1$	num=5
$\widehat{EN}_3^1$	$\widetilde{EN}_1^1$	num=4

Table 5.7: Test Input Data of exhaustive subnets set $CESN^2$.

Exhaustive Subnets	Simplified Subnets	Test Input Data
$\widehat{EN}_1^2$	$\widetilde{EN}_1^2$	$year$=0
$\widehat{EN}_2^2$	$\widetilde{EN}_2^2$	$year$=5
$\widehat{EN}_3^2$	$\widetilde{EN}_3^2$	$year$=400
$\widehat{EN}_4^2$	$\widetilde{EN}_4^2$	$year$=4



```
from z3 import *

a = Int('a')
b = Int('b')

#Solve the constraint condition 1
s1 = Solver()
s1.add(a <= 0)
print (s1.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 1 is:")
print (s1.model())
print ()

#Solve the constraint condition 2
s2 = Solver()
s2.add(Not(a <= 0),b <= 0)
print (s2.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 2 is:")
print (s2.model())
print ()

#Solve the constraint condition 3
s3 = Solver()
s3.add(Not(a <= 0),Not(b <= 0),a == b)
print (s3.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 3 is:")
print (s3.model())
print ()

#Solve the constraint condition 4
s4 = Solver()
s4.add(Not(a <= 0),Not(b <= 0),Not(a == b),Not(a > b),Not(b % a != 0))
print (s4.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 4 is:")
print (s4.model())
print ()

#Solve the constraint condition 5
s5 = Solver()
s5.add(Not(a <= 0),Not(b <= 0),Not(a == b),a > b,a % b != 0)
print (s5.check()) #check whether the constraint condition has a solution or not
print ("The solution of constraint conditions 5 is:")
print (s5.model())
print ()
```

Figure 5.24: The short program to find a set of test data for $\widetilde{CESN}^3$.

Table 5.8: Test Input Data of exhaustive subnets set $CESN^3$.

Exhaustive Subnets	Simplified Subnets	Test Input Data
$\widehat{EN}_1^3$	$\widetilde{EN}_1^3$	$(a{=}0, \forall b{\in}Z)$
$\widehat{EN}_2^3$	$\widetilde{EN}_2^3$	$(a{=}1, b{=}0)$
$\widehat{EN}_3^3$	$\widetilde{EN}_3^3$	$(a{=}1, b{=}1)$
$\widehat{EN}_4^3$	$\widetilde{EN}_4^3$	$(a{=}1, b{=}2)$
$\widehat{EN}_5^3$	$\widetilde{EN}_5^3$	$(a{=}3, b{=}2)$

5.3 Discussion

We have shown case studies of the three simple program nets, but our algorithms can be applied to any self-cleaning program nets, no matter how complex the program nets are. Moreover, although the programs used in our experiments are relatively small, our theoretical method is also applicable to large-scale programs. It should be noticed that (i) test data generation problem is generally NP-complete, and (ii) our test data generation uses only polynomial algorithms and $Z3$, which means $Z3$ would take much time when it is applied to large-scale programs. Therefore, it is a key issue to examine the relationship between the scale of a program and the execution time of test data generation, which is also one of our future works.

As introduced in Section 1.1, many techniques have been developed to generate test data, such as random testing technique, intelligent optimization algorithm, model-based testing method and search-based testing method. However, these techniques all have some shortcomings more or less. Random testing technique is not realistic and it will spend more time analysing results [83]. The technique of using intelligent optimization algorithms is a quick method to generate test data, nevertheless, the analysis required for various programs may be quite complex and algorithms may fall into a local optimum, which will result in the inability to obtain the optimal solution [84]. In the model-based testing method, modeling is a challenging task and requires a deep understanding of the application architecture and a wrong model may lead to inaccurate test data and wrong test execution [85]. Search-based testing technique has large search spaces and its fitness functions are an important factor in speed and efficiency, in addition, the technique suffers from prematurity, which may causes the convergence problem that effects on speed and direction [86]. Therefore, we proposed a method of applying program net to test data generation in order to overcome these shortcomings and strive to reduce the complexity. As a result, it is possible to minimize the time required to generate test data. Our method is divided into two steps: one is to generate subnets for an original program net; the other is to generate test data for the subnets.

For the first step, we have done the approach by designing four algorithms. For a given program net, Algorithm 1 is to generate its layer net, Algorithm 2 is to construct an acyclic net, Algorithm 3 is to generate an initial subnet, and Algorithm 4 is to generate a set of subnets that can covering all the nodes of the given net. All these algorithms are of polynomial computation time.

It is ideal to find minimum subnets to cover all the nodes, but that takes much computation time. This is because, the total number of subnets is exponentially proportional to the number of SWITCH-nodes and finding the minimum subnets from the total will be of exponential computation time. Therefore, we have tried to find a set of subnets as few as possible, as can be found at lines 8∼20 of Algorithm 4 by setting c_i=1 (or 0) if $\tilde{c_i}$=0 (or 1) for each SWITCH-node sw_i to avoid duplicate selection of its T or F. As the result, the maximum number of subnets is $2k$ (k is SWITCH-nodes' number). Even if devising this way, the number of obtained subnets is not necessarily the smallest. This is because, generally the 2nd subnet is dependent on the 1st, the 3rd is dependent on the 1st and the 2nd and so on. Incidentally, the numbers of generated subnets of three example program nets in our experiments are all the smallest. We have to mention that although our algorithms generate a set of subnets in polynomial computation time, this does not means software testing problem can also be solved in polynomial time since the second step to find test data for each subnet is also a fairly complex problem.

The second step is to test input data generation for such subnets that have test input data. We also have designed two algorithms to do that. Algorithm 5 is to construct a simplified exhaustive subnet and Algorithm 6 is to obtain all constraint conditions. These two algorithms are also of polynomial computation time.

From the results of 4.3.2, it is obvious that we can certainly find test input data for each subnet, which can make all the parts of each subnet executed. As mentioned in Chapter 2, 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 are generated by the SMT solver that may take computational time, our algorithms are all polynomial. This means that our method is feasible and effective to practically approach the software testing problem.

In summary, our method was successfully applied to the test data generation for three actual programs and finally we got the accurate test data, which implies our method can be considered feasible and effective. Although our method used $Z3$ prover to generate test data at the end, all algorithms, which are designed for a series of pre-operations in generating test data such as acyclic program net construction, subnets generation and constraint conditions generation, are of polynomial computation time. Therefore, our approach can save a lot of pre-operation time in actual test data generation, and it will definitely contribute significantly to software testing.

Chapter 6

Conclusions

Nowadays, software products are becoming more and more widely used in people's lives, and software testing is more important than ever. As introduced in Chapter 1, as an important testing activity, test data generation is an NP-complete problem and various methods have been proposed to approach this problem. As a tool to express programs, program nets can not only describe control flow and data flow but also simulate dynamic behaviours of programs. It is suitable for computer parallel processing and is also specially tuned to flows of data through arithmetic and logical operations. Therefore, it is appropriate to apply program nets to approach software testing problem.

This dissertation has presented a new method of applying program nets to generate test data for software testing.

In Chapter 2, we have given definitions of general program nets, such as three types of nodes, structural representation and firing rules of nodes. The basic properties of general program nets are also given. Then, based on the definition of general program nets, we have extended general program nets to exhaustive program nets, which can describe dynamic behavior information of a program net to obtain constraint conditions. We have also introduced an important concept of token self-cleanness, which is a property that there is no token remaining in the program net when its execution terminates, and this concept is the foundation of this dissertation. Finally, we have introduced software testing problem and symbolic execution technique, which is the guide of our method to test data generation.

In Chapter 3, we have designed a series of algorithms with polynomial computation time to generate a set of subnets which can cover all nodes of a given program net and all the nodes of each subnet can fire for certain given input test data. The subnets set can be generated by (i) treating a program as a original program net PN according to the basic definition of general program nets; (ii) constructing the layer net L_{PN} of the original program net PN in order to find all back edges of PN on the basis of firing rules and firing order of three types of nodes; (iii) deleting all back edges to cut off all directed circuits for constructing the acyclic program net $\widehat{PN}$ of the original program net PN with directed circuits; (iv) designing algorithm to generate an initial subnet $\widehat{PN}_1$ and obtain its SWITCH-nodes' states ψ_1 through setting the state of each synchronous SWITCH-nodes set of $\widehat{PN}$; (v) using a symbol $\odot$ to represent the operation between states $\tilde{\psi}$ of SWITCH-nodes in the current obtained subnets set and states ψ_i of SWITCH-nodes in the newly generated subnet $\widehat{PN}_i$ and then designing an algorithm to generate remaining subnets until the obtained subnets set CSN can cover all nodes of the original program net as well as all the states of SWITCH-nodes.

In Chapter 4, we have proposed a method to solve the remaining task, which is to find specific test input data for the subnets obtained in Chapter 3. This method can be implemented by (i) constructing the corresponding exhaustive subnet $\widehat{EN}_i$ of subnet $\widehat{PN}_i$ generated in Chapter 3 according to definition of exhaustive program net, which can represent dynamic behavior of programs and describe the concrete expression of constraint conditions in $\widehat{EN}_i$; (ii) designing an adjacency matrix A_i used to represent all information contained in exhaustive subnet $\widehat{EN}_i$ such as state of each SWITCH-node and the operator at node; (iii) designing an algorithm to delete all sink nodes that are not SWITCH-nodes to obtain a simplified subnet $\widetilde{EN}_i$ and update the corresponding adjacency matrix, and further record each SWITCH-node's state; (iv) designing an algorithm to obtain all constraint conditions from simplified subnets; (v) using $Z3$ prover with Python to treat all constraint conditions as its input and the output is the required test input data of exhaustive subnets.

In Chapter 5, we have applied our method to three actual Java programs. Firstly, their corresponding original program nets were constructed; then, we applied the designed six algorithms to them to generate subnets set and obtain all constraint conditions; next, $Z3$ prover was used to obtain specific test input data for each exhaustive subnet; finally, we discussed feasibility, effectiveness and applicability for our proposed method.

Program net theory includes mathematical semantics, graphical representation, executable representation, and many analysis techniques. It is a powerful tool for test input data generation in order to approach software testing problem. Except adopting $Z3$ prover, all algorithms designed in this dissertation are of polynomial computation time, which is supposed to greatly improve the efficiency of test data generation technology. Hence these proposed algorithms contribute to the field of software testing. Today is an era of rapid development of information technology.

All walks of life are becoming informationalized and intelligent, and various software technology products are emerging in an endless stream, such as mobile phone quick payment apps, cloud services, Internet of Things, artificial intelligence, etc.. Especially in Asia, with the vigorous development of the software technology industry, many software products have made people's lives more convenient and comfortable, such as Alipay, Nintendo games, driverless car and so on. These products without exception contain a large number of programs and it is necessary to test these large-scale programs to improve their quality and safety. Our approach is expected to further promote the development of the software industry and contribute to the economic development of Asia.

The following problems related to this dissertation are to be resolved in the near future.

(i) To improve our proposed algorithms to find an optimal set of subnets that definitely have some specific input data to make all the SWITCH-nodes firable;

(ii) To implement such a system that automatically construct a program net from a given program and finally uses $Z3$ prover to obtain test data; and

(iii) To develop an effective SMT solver that can efficiently acquire test data of actual large-scale programs.

Bibliography

[1] Halstead and M. Howard, Elements of software science, New York: Elsevier, vol.7, 1977.

[2] C. Ebert, M. Kuhrmann, and R. Prikladnicki, Global software engineering: An industry perspective, IEEE Software, vol.33, no.1, pp.105-108, 2015.

[3] Y.D. Wei, X. Bi, M. Wang, et al, Globalization, economic restructuring, and locational trajectories of software firms in shanghai, The Professional Geographer, vol.68, no.2, pp.211-226, 2016.

[4] A.A. Khan and J. Keung, Systematic review of success factors and barriers for software process improvement in global software development, IET software, vol.10, no.5, pp.125-135, 2016.

[5] L. Wood, Software industry in Asia Pacific 2015-2020–forecast, opportunities & trends, research and markets, September 22, 2016

[6] R. Sinha, M. Shameem, and C. Kumar, SWOT: strength, weaknesses, opportunities, and threats for scaling agile methods in global software development, Proc. of the 13th innovations in software engineering conference on formerly known as India software engineering conference, pp.1-10, 2020.

[7] S. Sikdar, R.C. Das, and R. Bhattacharyya, Role of IT-ITES in economic development of Asia, Springer, 2020.

[8] H. Hirakawa, Global ICT-based services offshoring and Asia, Innovative ICT Industrial Architecture in East Asia, Springer, Tokyo, pp.1-31, 2017.

[9] S. Mahmood, S. Anwer, M. Niazi, et al, Key factors that influence task allocation in global software development, Information and Software Technology, vol.91, pp.102-122, 2017.

[10] N.G. Leveson, The Therac-25: 30 Years Later, Computer, vol.50, no.11 pp.8-11, 2017.

[11] X. Yu, J. Qiu, X. Yang, et al, An graph-based adaptive method for fast detection of transformed data leakage in IOT via WSN, IEEE Access, vol.7, pp.137111-137121, 2019.

[12] Perlroth and Nicole, Yahoo says hackers stole data on 500 million users in 2014, The New York Times, Retrieved September 22, 2016.

[13] V. Goel and N. Perlroth, Yahoo says 1 billion user accounts were hacked, The New York Times, Retrieved December 14, 2016.

[14] D.F. Larios, J. Barbancho, F. Biscarri, et al, A research study for the design of a portable and configurable ground test system for the A400M aircraft, International Journal of Aerospace Engineering, 2019.

[15] R. Gardner, An amazing spectrum: Airbus military in Spain, Vayu Aerospace and Defence Review, no.6, pp.78, 2015.

[16] Y. Singh, Software Testing, Cambridge: Cambridge University Press, 2012.

[17] P. Ammann and J. Offutt, Introduction to software testing, Cambridge University Press, 2016.

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

[19] I. Hooda and R.S. Chhillar, Software test process, testing types and techniques, International Journal of Computer Applications, vol.111, no.13, 2015.

[20] K. Mao, M. Harman, and Y. Jia, Robotic testing of mobile apps for truly black-box automation, IEEE Software, vol.34, no.2, pp.11-16, 2017.

[21] S. Fink, Y.A. Haviv, R. Hay, et al, Simulating black box test results using information from white box testing, U.S. Patent 9,720,798[P], 2017-8-1.

[22] M.J. Andrade, White-box testing automation with sonarQube: continuous integration, code review, security, and vendor branches, code generation, analysis tools, and testing for quality, IGI Global, pp.64-88, 2019.

[23] S.R. Jan, S.T.U. Shah, Z.U. Johar, et al, An innovative approach to investigate various software testing techniques and strategies, International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN, pp.2395-1990, 2016.

[24] C.K. Mohd and F. Shahbodin, Personalized learning environment: Alpha testing, Beta Testing user acceptance test, Procedia-Social and Behavioral Sciences, vol.195, pp.837-843, 2015.

[25] V.P. Katiyar and M.S. Patel, White-box testing technique for finding defects, Global Journal For Research Analysis, vol.8, no.7, 2019.

[26] M.M. Syaikhuddin, C. Anam, A.R. Rinaldi, et al, Conventional software testing using white box method, Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control, vol.3, no.1, pp.65-72, 2018.

[27] E.A. Bock, A. Amadori, C. Brzuska, et al, On the security goals of white-box cryptography, IACR Transactions on Cryptographic Hardware and Embedded Systems, pp.327-357, 2020.

[28] N. Mansour and M. Houri, White box testing of web applications, Journal of Systm and Software, pp.1-9, 2017.

[29] C. Henard, M. Papadakis, M. Harman, et al, Comparing white-box and black-box test prioritization, Proc. of 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp.523-534, 2016.

[30] J. Lin and P. Yeh, Automatic test data generation for path testing using GAs, Information Sciences, vol.131, no.1-4, pp.47-64, 2001.

[31] V.T. Dao, P.N. Hung, and V.H. Nguyen, A method for automated test cases generation from sequence diagrams and object constraint language for concurrent programs, VNU Journal of Science: Computer Science and Communication Engineering, vol.32, no.3, 2016.

[32] R.V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley Professional, 2000.

[33] 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.

[34] N. Mansour and M. Salame, Data generation for path testing, Software Quality Journal, vol.12, no.2, pp.121-136, 2004.

[35] M.E. Khan, Different approaches to white box testing technique for finding errors, International Journal of Software Engineering and Its Applications, vol.5, no.3, pp.1-14, 2011.

[36] A. Maha, K. Ajay, 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.

[37] J. Minj, Feasible test case generation using search based technique, International Journal of Computer Applications(IJCA), vol.70, no.28, pp.51-55, 2013.

[38] P. McMinn, Search-based software test data generation: a survey, Software testing, Verification and reliability, vol.14, no.2, pp.105-156, 2004.

[39] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, Grt: Program-analysis-guided random testing (t), Proc. of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp.212-223, 2015.

[40] M. Khari, P. Kumar, D. Burgos, and R.G. Crespo, Optimized test suites for automated testing using different optimization techniques, Soft Computing, vol.22, no.24, pp.8341-8352, 2018.

[41] A. Fellner, W. Krenn, R. Schlick, T. Tarrach, and G. Weissenbacher, Model-based, mutation-driven test-case generation via heuristic-guided branching search, ACM Transactions on Embedded Computing Systems (TECS), vol.18, no.1, 2019.

[42] N. Jatana and B. Suri, An improved crow search algorithm for test data generation using search-based mutation testing, Neural Processing Letters, vol.52, no.1, pp.767-784, 2020.

[43] M. Ibe, Decomposition of test cases in model-based testing, Proc. of the MODELS 2013 Doctoral Symposium, vol.1071, 2013.

[44] P. Arcaini, A. Gargantini, and E. Riccobene, Decomposition-based approach for model-based test generation, IEEE Transactions on Software Engineering, no.99, 2017.

[45] K. Onaga and W.K. Chen, Data flow analysis of program nets, J. Franklin Inst., vol.313, no.4, pp.219-231, 1982.

[46] J.B. Dennis, First version of data flow procedure language, Proc. of Lecture Notes in Computer Science 19G, Springer, New York, 1974.

[47] J.B. Dennis, The MIT data flow engineering model, Proc. of IFIP Congress 83, pp.553-560, 1983.

[48] J. Rumbaugh, A data flow multiprocessor, IEEE Transactions on Computers, vol.100, no.2, pp.138-146, 1977.

[49] J. Gurd and I. Watson, A data driven system for high speed parallel computing, Computer Design, vol.19, no.6 and no.7, pp.91-100 and pp.97-106, 1980.

[50] H. Terada and H. Nishikawa, A VLSI oriented data driven processor: Q-x, bit (in Japanese), vol.21, no.4, pp.466-475, 1989.

[51] T. Shimada, K. Hiraki, and K. Nishida, An architecture of a data flow machine and its evaluation, Proc. of Compcon. Spring, IEEE, pp.486-490, 1984.

[52] T. Shimada, K. Hiraki, and S. Sekiguchi, A data flow supercomputer SIGMA-l for scientific computations, bit (in Japanese), vol.21, no.4, pp.435-442, 1989.

[53] S. Sekiguchi, T. Shimada, and K. Hiraki, A design of a dataflow language DFCII for new generation supercomputer, Trans. Inf. Process. Soc. Jap., vol.30, No.12, pp.1639-1645, 1989.

[54] J.E. Rodriguez, A graph model for parallel computation, Report MAS-TR-64, 1969.

[55] A.L. Davis and R. M. Keller, Data flow program graphs, IEEE Comput., vol.15, no.2, pp.26-41, 1982.

[56] C.A. Petri, Kommunikation mit Automaten, Schriften des Rheinisch-westfalischen Institutes fur Instrumentelle Mathematik an der Universitat Bonn, Heft 2, Bonn, 1962.

[57] S.E. Elmaphraby, Activity networks: planning and control by network models, Wiley, New York, 1973.

[58] A.A.B. Pritsker and W.W. Happ, GERT: graphical evaluation and review technique, part I, fundamentals, J. Ind. Engng, vol.17, pp.267-274, 1966.

[59] X. Yu, J. Liu, Z. Yang, and L. Xiao, The bayesian network based program dependence graph and its application to fault localization, Journal of Systems and Software, vol.134, pp.44-53, 2017.

[60] 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.

[61] Q.W. Ge, T. Watanabe, and K. Onaga, Topological analysis of firing activities of data-flow program nets, IEICE Transactions (1976-1990), vol.E73, no.7, pp.1215-1224, 1990.

[62] Q.W. Ge, T. Watanabe, and K. Onaga, Topological analysis of firing activities of data-flow program nets, Proc. of ICCAS'89, Nanjing, China, 1989.

[63] 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.

[64] N. Zhang, B. Wu, and X. Bao, Automatic generation of test cases based on multi-population genetic algorithm, Int. J. Multimedia Ubiquitous Eng., vol.10, no.6, pp.113-122, 2015.

[65] S. Rapps and E.J. Weyuker, Data flow analysis techniques for test data selection, Proc. of the 6th international conference on Software engineering, pp.272-278, 1982.

[66] S. Rapps and E.J. Weyuker, Selecting software test data using data flow information, IEEE Trans. Software Eng., vol.11, no.4, pp.367-375, 1985.

[67] J.C. King, Symbolic execution and program testing, Communications of the ACM, vol.19, no.7, pp.385-394, 1976.

[68] C.S. Peanu and W. Visser, A survey of new trends in symbolic execution for software testing and analysis, International journal on software tools for technology transfer, vol.11, no.4, pp.339, 2009.

[69] S. Khurshid, C.S. P*ăsă*reanu, and W. Visser, Generalized symbolic execution for model checking and testing, Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, Berlin, Heidelberg, pp.553-568, 2003.

[70] G. Nelson, Techniques for Program Verification, PhD dissertation, Stanford University, 1981.

[71] 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.

[72] G. Katz, C. Barrett, D.L. Dill, K. Julian, and M. J. Kochenderfer, Reluplex: an efficient smt solver for verifying deep neural networks, In Computer Aided Verification, pp.97-117,2017.

[73] L. De Moura and N. Bjørner, Satisfiability Modulo Theories: introduction and applications, Commun. ACM, vol.54, no.9, pp.69-77, 2011.

[74] L. De Moura and N. Bjørner, Z3: An efficient smt solver, Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, pp.337-340, 2008.

[75] B. Dutertre, Yices 2.2, Proc. of International Conference on Computer Aided Verification, Springer, Cham, pp.737-744, 2014.

[76] C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli, CVC4, Proc. of the 23rd International Conference on Computer Aided Verification (CAV'11), Springer, Berlin, Heidelberg, pp.171-177, 2011.

[77] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, The mathSAT5 SMT solver, Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, Berlin, Heidelberg, pp.93-107, 2013.

[78] A. Niemetz, M. Preiner, and A. Biere, Boolector 2.0 system description, Journal on Satisfiability, Boolean Modeling and Computation, vol.9, no.1, pp.53-58, 2015.

[79] Q.W. Ge, N. Ono, and K. Onaga, Firing activity and well-behavedness of data-flow program nets, Trans. IEICE Jap., vol.E73, no.7, pp.1215-1224, 1990.

[80] T.A. Henderson, Behavioral fault localization by sampling suspicious dynamic control flow subgraphs, Proc. of the IEEE 11th International Conference on Software Testing, Verification and Validation Case, pp.93-104, 2018.

[81] N. Clark, H. Zhong, and S. Mahlke, Processor acceleration through automated instruction set customization, Proc. of the 36th annual IEEE/ACM International Symposium on Microarchitecture, pp.129-140, 2003.

[82] 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.

[83] E. Nikravan and S. Parsa, Path-oriented random testing through iterative partitioning (IP-PRT), Turkish Journal of Electrical Engineering Computer Sciences, vol.27, no.4, pp.2666-2680, 2019.

[84] R.P. Pargas, M.J. Harrold, and R.R. Peck, Test-data generation using genetic algorithms, Software testing, verification and reliability, vol.9, no.4, pp.263-282, 1999.

[85] B.K. Aichernig, W. Mostowski, M.R. Mousavi, et al, Model learning and model-based testing, Machine Learning for Dynamic Software Analysis: Potentials and Limits, vol.11026, pp.74-100, 2018.

[86] S.M. Mohi-Aldeen, S. Deris, and R. Mohamad, Systematic mapping study in automatic test case generation, New Trends in Software Methodologies, Tools and Techniques, vol.265, pp.703-720, 2014.

Appendix A

Abbreviation of Substances

PN :	Program net
MPN :	Marked program net
$FIFO$:	First input first output
L_{PN} :	Layer net
$\widehat{PN}$:	Acyclic program net
L_i :	The i-th layer of L_{PN}
$l(v)$:	The layer number of node v
SSW :	Synchronous nodes set
ψ :	The states of SWITCH-nodes in one subnet
$\widehat{PN}_i$:	Subnet
EN :	Exhaustive program net
$\widehat{EN}_i$:	Exhaustive subnet
$\widetilde{EN}_i$:	Simplified exhaustive subnet
Ψ :	The set of states of SWITCH-nodes in all subnets
WPN :	The set of all subnets
$\tilde{\psi}$:	The states of SWITCH-nodes in subnets set