Sudan University of Science and Technology

College of Graduated Studies

An Entire Security Model for SQL-injection: Formal Specification Using Petri Nets

نموذج أمن (كامل/شامل) لهجمات حقن SQL: وصف رسمى باستخدام شبكات بيتري

A Thesis Submitted in Partial Fulfillment of the Requirements of M.Sc. in Computer Science (Information Security Track)

By

Lobaba Eltayeb Ahmed Mohamed

Supervisor

Dr. Awad Mohamed Awadelkarim

2016

DEDICATION

I dedicate this research to person whose prayer helps me my mother

To person whose encourage me to the way of success my father

My brothers and sisters for their support

My friends and my colleagues the people whom I love and respect

Everyone from him I learned.

ACKNOWLEDGEMENT

I would like to express my special thanks and gratitude to my supervisor Dr. Awad Mohamed Awadelkarim for constructive guidance.

It is great pleasure to thank my best friends and colleagues for their cooperation.

ABSTRACT

Database Security has gained significance and concern as institutions reliance on database systems has increased dramatically in addition to the simultaneous and severe grown of the associated offensives. Furthermore, and with the development, use and widespread of the Internet and web applications, it has been very important to ensure the confidentiality of information and protection from threats such as SQL Injection Attack (SQLIA). Which are considered as one of the top threats and prevalent types of database-driven applications security vulnerability.

Consequently, SQLIA prevention and detection has become one of the most active topics of research in the computer science field. Therefore, this research contributes to such context by proposing an inclusive and formal security model for nearly all existing SQL-injection attacks using Petri Nets language. Additionally, the study has followed a scientific and formal methodology including determination of security requirements based on comprehensive security risk analysis and assessment. Moreover, The proposed model guarantees and supports multi-defense lines with variform-adaptable mechanisms that might gain the superiority of safeguard for the intended model. Finally, the study conducts and develops formal modeling in company with formal system specification for the proposed model using Petri Nets notation in order to assure and prove modularity, conformity, reliability, as well as flexibility.

المستخلص

ظهرت أهمية أمن قواعد البيانات من الإزدياد الواسع لأنظمة قواعد البيانات والهجمات المرتبطة بها، كما أن التطور والإستخدام واسع الإنتشار للإنترنت وتطبيقات الويب جعل من الأهمية ضمان سرية المعلومات وحمايتها من المهددات مثل هجوم حقن SQL (SQLIA). الذي يعتبر واحد من أخطر المهددات للتطبيقات المعتمدة على قواعد البيانات، نتيجة لذلك، أصبحت الحماية من هجوم حقن SQL أحد المواضيع الأكثر نشاطا في البحوث العلمية لمجالات علوم الكمبيوتر.

ونتيجة لذلك، أصبح منع وكشف هجوم حقن SQL أحد أكثر المواضيع النشطة في مجال علم الحاسبات. ولذا، فإن هذا البحث يساهم في هذا المجال من خلال اقتراح نموذج أمني شامل ورسمي لكل هجمات حقن SQL الحالية تقريباً باستخدام لغة شبكات بيتري. بالإضافة إلى ذلك، فقد اتبعت الدراسة منهجية علمية ورسمية تتضمن تحديد المتطلبات الأمنية على أساس تحليل وتقييم شامل للمخاطر الأمنية. وعلاوة على ذلك، فإن النموذج المقترح يضمن ويدعم أكثر من خط دفاع من خلال الدمج بين العديد من الآليات والتي أكسبته تفوق في الحماية من المهددات. وأخيراً، تمت دراسة وتطوير نموذج رسمي وتوصيفه بصورة رسمية بإستخدام باستخدام تدوين لغة شبكات بيتري من أجل ضمان وإثبات النمذجة والموثوقية بالإضافة إلى المرونة.

LIST OF FIGURES

LIST OF TABLES

Table of Contents

Chapter 1 - Introduction

1.1 Introduction

Nowadays a Database security has become an important issue in technical world. The main objective of database security is to forbid unnecessary information exposure and modification data while ensuring the availability of the needed services. A numbers of security methods have been created for protecting the databases. Many security models have been developed based on different security aspects of database. All of these security methods are useful only when the database management system is designed and developing for protecting the database. Recently the growth of web application with database at its backend Secure Database Management System is more essential than only a Secure Database [1] But we must also protect web applications connected to databases because the vulnerabilities in Web applications that can negatively affect on the security of database.

With the rising use of internet, web application vulnerability has been increasing effectively. All web applications are depended on the Internet. Example: e-banking, admission portals, online shopping, and various government activities like online electricity bills payment etc. Since these applications are used by hundreds of people, in many cases the security level is weak, which makes them vulnerable to be attacked by external users. From time to time, the users need to interact with the backend databases through the user interfaces for various tasks such as: modify data, manipulating queries, extracting data, and so forth. For all these operations, design interface plays crucial role, the quality of which has a great bang on the security of the stored data in the database. A less secure Web application design may allow crafted injection and malicious update on the backend database. This trend can cause lots of damages and thefts of trusted users' sensitive data by unauthorized users. In the worst case, the attacker may gainful control over the Web application and totally destroy or

damage the system. This is effectively achieved, in general, through SQL injection attacks on the online Web application database. According to OWASP report released in 2012, SQL Injection attacks are top most risk/danger to Web applications [2].

SQL injection is typically involves malicious modifications of the user SQL input either by adding additional clauses or by changing the structure of an existing clause. SQL injection enables attackers to access, modify, or delete critical information in a database without proper authorization [3].

Formal specification is part of a more general collection of techniques that are known as Formal Methods. These are all based on mathematical representation and analysis of software. This is a technique for unambiguous specification of software [4].

Formal specifications are better than natural and programming language specification. Because the natural language specification is too ambiguous and imprecise and the programming language specification is too many details (not an abstraction of reality), cannot be understood by stakeholders, does not give the overall picture and many design decision on the way. But formal specifications have many advantages as Abstraction (good mechanism to support implementation freedom) and Precision (still maintain ability to precisely describe what is needed of the system) [5].

1.2 Problem Statement

SQL injections Attack still remain as one of the largest web application vulnerabilities. Number of models have been proposed and developed to counter SQL Injection Attack, and according to the best of our knowledge, there is no a particular distinct formal and inclusive model that countermeasure all occurred SQL Injection forms, therefore, this research contributes to such context.

1.3 The Research Objectives

The main objective of this research is to develop an inclusive and formal security model for SQL-injection by using Petri Nets language.

1.4 The Research Methodology

To accomplish the research objectives, the research phases can be summarized as follows:

1. Risk Analysis phase: in order to identify and determine the anticipated security vulnerabilities and attacks.
2. Security requirements definition and determination phase: in order to identify and determine the security requirements of the proposed model.
3. The model development phase.
4. Formal specification and verification of the proposed model using Petri Nets notations.

1.5 Organization of the Research

The thesis consist of six chapters as follows: chapter 1 represents the Introduction, chapter 2 provides the background and related work, chapter 3 deliberates the research methodology, chapter 4 presents the proposed model, chapter 5 demonstrates the formal specification and verification of the proposed model using Petri Nets notations. Finally, conclusion and future works to hand in chapter 6.

Chapter 2 - Literature Review

2.1 Introduction

This chapter provides the literature review done on concepts and methods that are used in this thesis, Section Two presents SQL injection attack (SQLIA) definition, basic concepts, process, and mechanism. Section Three presents classification of SQLIA. Section Four presents consequence of SQLIA. Section Five presents the defense techniques that are used to address the problems of SQLIAs. Section Six presents a formal specification definition, advantages of use it, and languages. Section Seven presents Petri nets language basics, formal definition, and properties. Section Eight presents modeling power of Petri Nets. Section Nine presents analysis of Petri Nets. Finally, in Section Ten presents related work.

2.2 SQLIA Overview

2.2.1 SQL Injection

SQL (Structured Query Language) is a textual language used to interact with relational Database. The typical unit of execution of SQL is the ‘query’, which is a collection of statements that typically return a single ‘resultset’. SQL statements can modify the structure of databases and manipulate the contents of databases by using various DDL, DML commands respectively. SQL Injection occurs when an attacker is able to insert a series of SQL statements into a query by manipulating data input into an application [6].

2.2.2 SQL Injection Vulnerability (SQLIV) versus SQL Injection Attack (SQLIA)

Vulnerability in any system is defined as a bug, loophole, weakness or flaw existing in the system that can be exploited by an unauthorized user in order to gain unlimited

access to the stored data. Attack generally means an illegal access, gained through well crafted mechanisms, to an application or system.

An SQL Injection Attack (SQLIA) is a type of attack whereby an attacker (a crafted user) adds malicious keywords or operators into an SQL query (e.g., SQL malicious code statements), then injects it to a user input box of a Web application. This allows the attacker to have illegal and unrestricted access to the data stored at the backend database. Figure 2.1 shows the normal user input process in a Web application, which is self-explanatory. Figure 2.2 shows an example how a malicious input could be processed in a Web application. In this case, the malicious input is the carefully formulated SQL query which passes through the system's verification method [7].

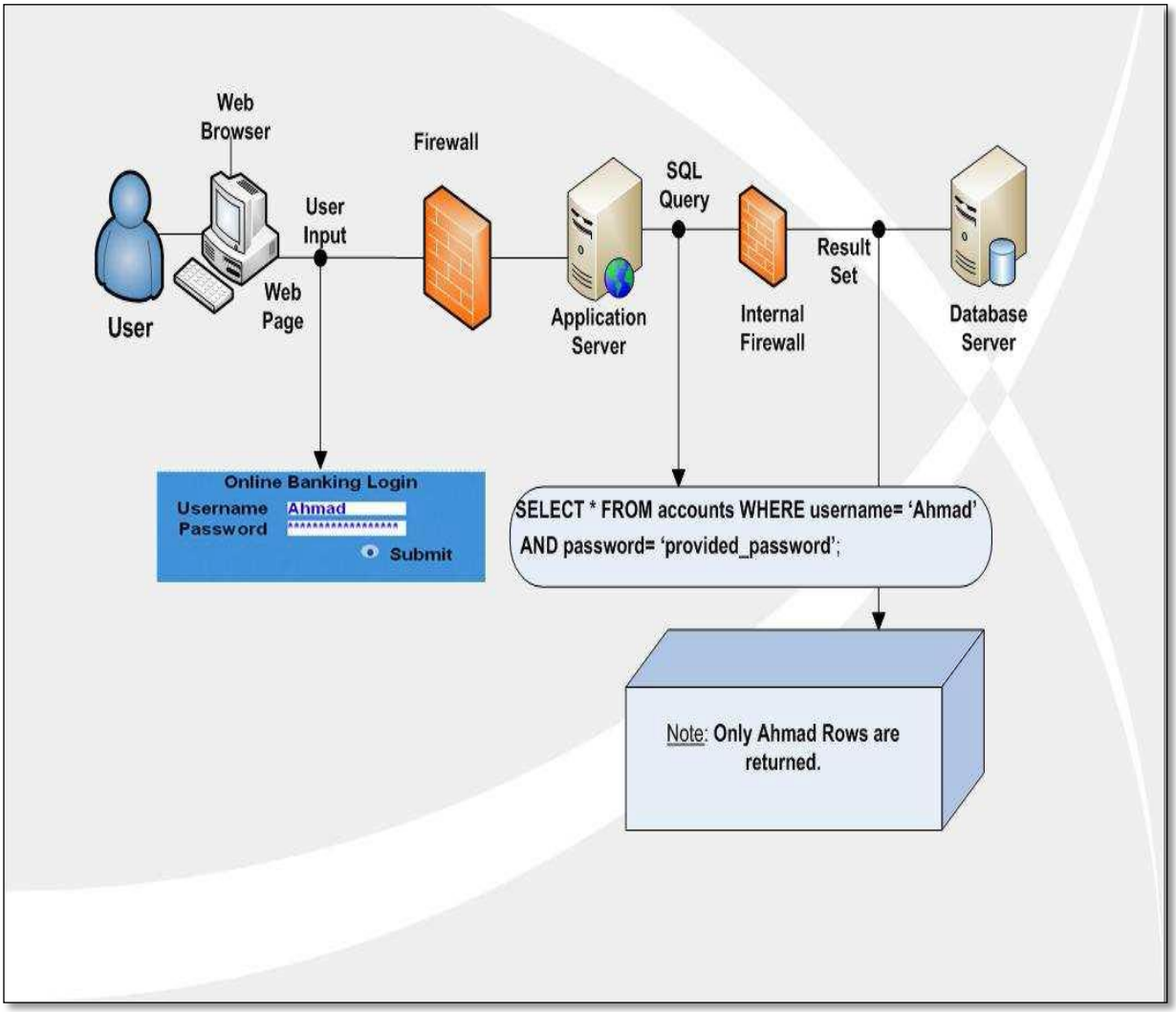

Figure 2.1: Normal User Input Process in a Web Application [7]

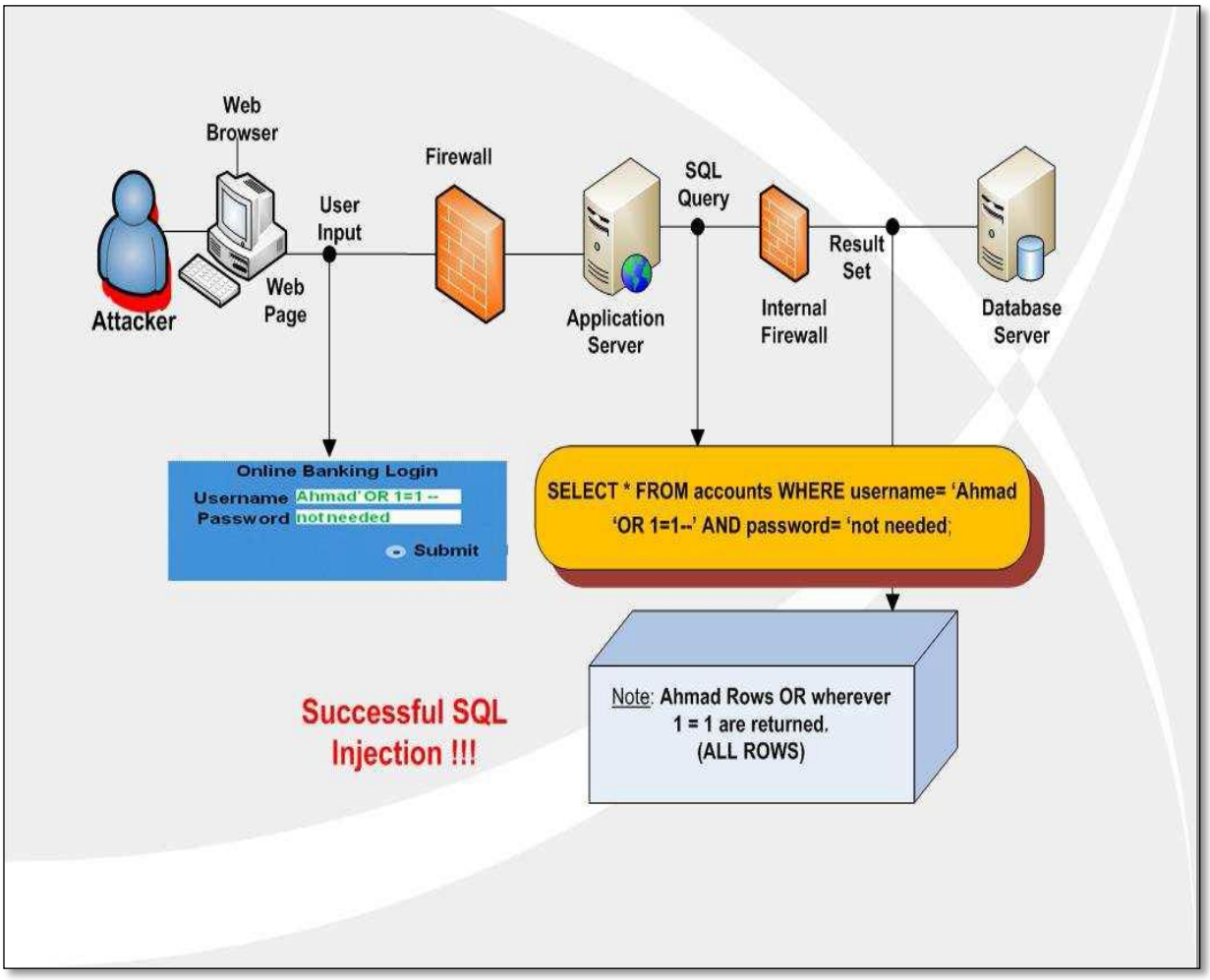

Figure 2.2: Malicious Input Process in a Web Application [7]

2.2.3 SQLIA Process

SQLIA is hacking technique which the attacker adds SQL statements through a web application's input field or hidden parameter to access to resources. Lack of input validation in web applications causes hacker to be successful. Basically SQL process structured in three phases [6]:

i. An attack sends the malicious HTTP request to the web application.

ii. Create the SQL Statements.

iii. Submits the SQL statements to the back end database

Figure 2.3 shows an example for SQLIA data flow.

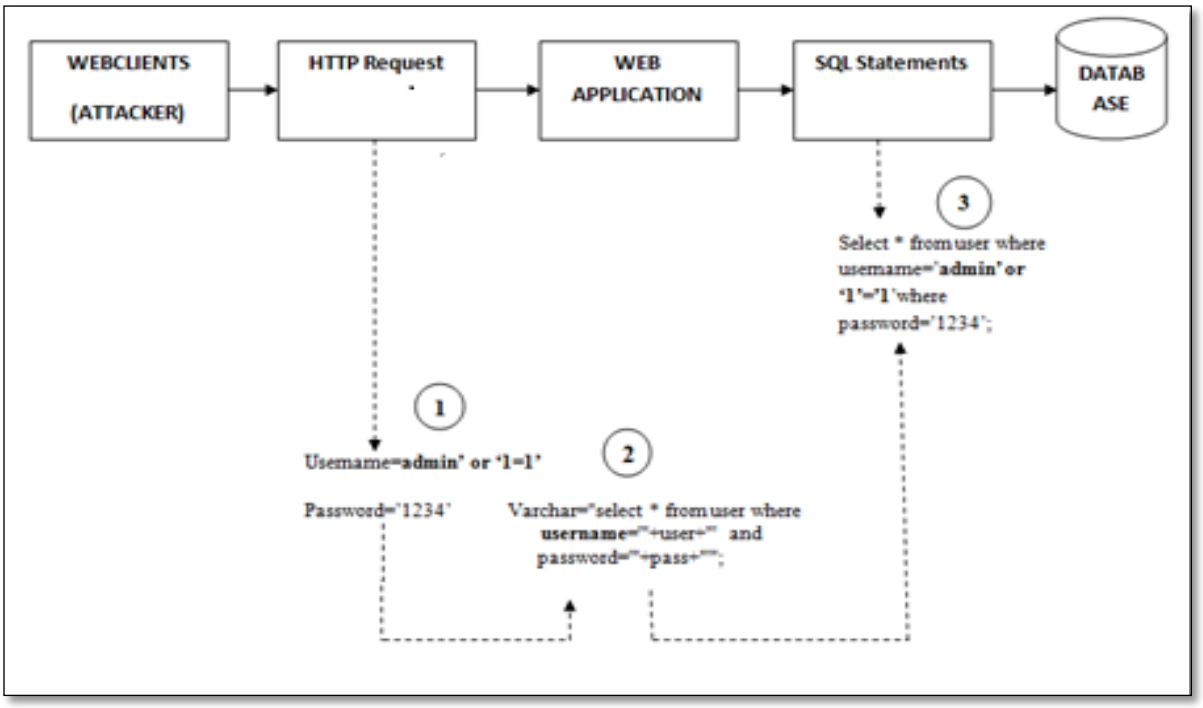

Figure 2.3: Example for SQLIA Data Flow [6]

2.2.4 SQLIA Mechanisms

Malicious SQL statements can be introduced into a vulnerable application using many of different input mechanisms. These are the most common mechanisms [8]

2.2.4.1 Injection through user input

In the type of injection the attacker injects SQL commands by providing suitably crafted user input. A web application can read user's input in several ways based on the environment in which the application is deployed.

2.2.4.2 Injections through cookies

Cookies are the small files that containing state information generated by Web applications and stored on the client machine. When a client returns to the Web application the cookie is used to be restore the client information. Since the client has control over the storage of cookie, a malicious client could tamper with the cookie's content. And then if Web application uses the cookie content to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie.

2.2.4.3 Injections through the server variables

Server variables are collections of variables that contain HTTP, network headers, and environmental variables .Web applications used these server variables in a variety of ways like logging usage. If these servers logged to a database without sanitization, this could create SQLI vulnerability because attacker can forge the values that are placed in HTTP and network headers. They can exploit this vulnerability by placing an SQLIA directly into the headers. And when the query to log the server variable is issued to the database, the attack in the forged header is triggered automatically.

2.2.4.4 Second order injection

In second order injection, attacker seed malicious inputs in to a system or database to indirectly trigger an SQLIA when that input is used at a later time. The attack takes place when the malicious input reaches to the database.

2.3 Classification of SQLIA

An SQLIA can be classified by using some of properties such as attacker intent and attack techniques utilized by threat agents.

2.3.1 By Attacker Intent

An important classification of SQLIA is related to the attacker's intent, or in other words, the goal of the attack.

2.3.1.1 Extracting data

This category of attacks tries to extract data values from the back end database. Based on the type of web application, this information could be sensitive, for example, credit card numbers, social numbers; private data are highly valuable to the attacker. This kind of intent is the most common type of SQLIA [6].

2.3.1.2 Adding or modifying data

The purpose of these attacks is to add or change data values within a database [6].

2.3.1.3 Performing database finger printing

In this category of attack the malicious user wants to discover technical information on the database such as the type and version that a specific web application is using. It is noticeable that certain types of databases respond differently to different queries and attacks, and this information can be used to "fingerprint" the database. Once the intruder knows the type and the version of the database it is possible to organize a particular attack to that database [6].

2.3.1.4 By passing authentication

By this attack, intruders try to bypass database and application authentication mechanisms. Once it has been over passed, such mechanisms could allow the intruder to assume the rights and privileges associated with another application user [6].

2.3.1.5 Identifying injects able parameter

Its goal is to explore a web application to discover which parameters and user-input fields are vulnerable to SQLIA. By using an automated tool called a "vulnerabilities scanner" this intent can be identified [6].

2.3.1.6 Determining database schema

The goal of this attack is to obtain all the database schema information (such as table names, column names, and column data types). This is very useful to an attacker to gather this information to extract data from the database successfully. Usually by exploiting specific tools such as penetration testers and vulnerabilities scanners this goal is achieved [6].

2.3.1.7 Performing denial of service

In these category intruders make interrupt in system services by performing some instruction so the database of a web application shutdown, thus denying service happens. Attacks involving locking or dropping database tables also fall into this category [6].

2.3.1.8 Evading detection

This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms [9].

2.3.1.9 Executing remote commands

These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users [9].

2.3.1.10 Performing privilege escalation

These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attacks focus on exploiting the database user privileges [9].

2.3.2 By attack techniques

2.3.2.1 Tautology:

Attack Intent: Bypassing authentication, identifying inject able parameters, extracting data [6].

Description: In the tautology attack the attacker tries to use a conditional query statement to be evaluated always true. Attacker uses WHERE clause to inject and turn the condition into a tautology which is always true. The simplest form of tautology

Example:

```
SELECT *FROM Accounts WHERE user=’’or1=1— ‘AND pass=’’AND eid=
```

The result would be all the data in accounts table because the condition of the WHERE clause is always true [8].

2.3.2.2 Illegal/Logical Incorrect queries

Attack Intent: Identifying inject able parameters, performing database finger-printing, extracting data [6].

Description: When a query is rejected an error message is returned from the database including useful debugging information. This information helps attackers to make move further and find vulnerable parameters in the application and consequently database of the application.

Example:

```
SELECT * FROM Accounts WHERE user=’ ‘ AND pass=’ ‘AND eid =convert(int,(SELECT TOP 1name FROM sysobjects WHERE xtype=’u’))
```

In the example the attacker attempts to convert the name of the first user defined table in the metadata table of the database to ‘int’. This type conversion is not legal therefore the result is an error which reveals some information that should not be shown [8].

2.3.2.3 Union queries

Attack Intent: Bypassing Authentication, extracting data [6].

Description: In this type of queries unauthorized query is attached with the authorized query by using UNION clause.

Example:

```
SELECT * FROM Accounts WHERE user=’’ UNION SELECT * FROM Students—‘AND pass=’’AND eid=
```

The result of the first query in the example given above is null and the second one returns all the data in students table so the union of these two queries is the student table [8].

2.3.2.4 Piggy-Backed query

Attack Intent: Extracting data, adding or modifying data, performing denial of service, executing remote commands [6].

Description: In the query attack attacker tries to add an additional queries in to the original query string .In this injection the intruders exploit database by the query delimiter, such as “;”, to append extra query to the original query

Example:

```
SELECT*FROM Accounts WHERE user=’’; drop table Accounts—‘AND pass=’ ‘ AND eid=
```

The result of the example is losing the credential information of the accounts table because it would be dropped branch from database [8].

2.3.2.5 Stored Procedure

Attack Intent: Performing privilege escalation, performing denial of service, executing remote commands [6].

Description: In this technique, attacker focuses on the stored procedures which are present in the database system. Stored procedures run directly by the database engine. Stored procedure is nothing but a code and it can be vulnerable as program code. For authorized/unauthorized user the stored procedure returns true/false. As an SQL Injection Attack, intruder input “; SHUTDOWN; --" for username or password. Then the stored procedure generates the following query:

Example:

```
SELECT accounts FROM users WHERE login= '1111' AND pass='1234 '; SHUTDOWN;--;
```

This type of attack works as piggyback attack. The first original query is executed and consequently the second query which is illegitimate is executed and causes database shut down. So, it is considerable that stored procedures are as vulnerable as web application code [10].

2.3.2.6 Inference

Attack Intent: Identifying inject able parameters, extracting data, determining database schema [6].

Description: By this type of attack, intruders change the behavior of a database or application. There are two well known attack techniques that are based on inference: blind injection and timing attacks.

a) Blind Injection

At times developers hide the error details which help attackers to compromise the database. In this situation attacker face to a generic page provided by developer, instead of an error message. So the SQLIA would be very difficult but not impossible. An attacker can still steal data by asking a series of True/False questions through SQL statements.

Example:

```
SELECT accounts FROM users WHERE login= 'doe' and 1 =0 -- AND pass = AND pin=O SELECT accounts FROM users WHERE login= 'doe' and 1 = 1 -- AND pass = AND pin=O
```

If the application is secured, both queries would be unsuccessful, because of input validation. But if there is no input validation, the attacker can try the chance. First the attacker submits the first query and receives an error message because of "1 =0 ". So the attacker does not understand the error is for input validation or for logical error in query. Then the attacker submits the second query which always true. If there is no login error message, then the attacker finds the login field vulnerable to injection [10].

b) Timing Attacks

A timing attack lets attacker gather information from a database by observing timing delays in the database's responses. This technique by using if-then statement cause the SQL engine to execute a long running query or a time delay statement depending on the logic injected. This attack is similar to blind injection and attacker can then measure the time the page takes to load to determine if the injected statement is true. This technique uses an if-then statement for injecting queries. WAITFOR is a keyword along the branches, which causes the database to delay its response by a specified time. **Example**, in the following query:

```
declare @ varchar(8000)
 select @ = db_nameO if (ascii(substring(@, 1, 1)) & ( power(2, 0))) > 0 waitfor delay '0:0:5'
```

Database will pause for five seconds if the first bit of the first byte of the name of the current database is 1. Then code is then injected to generate a delay in response time when the condition is true. Also, attacker can ask a series of other questions about this character. As these examples show, the information is extracted from the database using a vulnerable parameter [10].

2.3.2.7 Alternate encoding

Attack Intent: Evading detection [6].

Description: In this type of attack the regular strings and characters are converted into hexadecimal, ASCII and Unicode. Because of this the input query is escaped from filter which scans the query for some bad character which results SQLIA and the converted SQLIA is considered as normal query.

Example:

```
SELECT * FROM Accounts WHERE user='user1'; exec(char(0x8774675u8769e)) - -'
AND pass=' ' AND eid=
```

The example char () function and ASCII hexadecimal encoding are used. The functions will get integer number as a parameter and return as a sample of that character. In the example it will return "SHUTDOWN", so whenever the query is interpreted the SHUTDOWN command is executed [8].

2.4 Result of SQLIA

2.4.1 Reports about the seriousness of SQLIA

The open web application security project (OWASP) ranks SQLI as the most widespread website security risk in 2011. The National Institute of Standards and Technology's National vulnerability Database reported 289 SQL vulnerabilities in websites including those of IBM, HP, and MICROSOFT. In December 2011, SANS Institute security experts reported a major SQL injection attack that affects approximately 160000 websites using Microsoft's Internet Information Services (IIS), ASP.NET, and SQL Server Frameworks [8].

Semiannual Report (July to December 2010) from the Web Hacking Incidents Database (WHID) shows that that SQL injection are consistently or near the top 21% of the reported vulnerabilities in 2010 ,consider as top 2 attack and recently in August, 2011, Hacker steals user records from Nokia Developer Site using "SQL injection". They are easy to detect and exploit; that is why SQLIAs are frequently employed by malicious user for different reasons. E.g. financial fraud, theft, confidential data, deface website, sabotage, espionage, cyber terrorism, or simply for fun. Throughout 2010, Government, Finance and Retail verticals faced different, but equally important, outcomes. Attacks against Government agencies resulted in defacement in 26% of SQL injection attacks, while Retail was most affected by credit card leakage at 27% of SQL injection and finance experienced monetary loss in 64% of attacks [44]. Furthermore, SQL Injection attack techniques have become more common more ambitious, and increasingly sophisticated, so there is a deep to need to find an effective and feasible solution for this problem in the computer security community [6].

2.4.2 Consequence of SQLIA

The code injected in the application can manipulate the information and structure of the database by using just few statements. Being common and efficient technique it is very important for the developers and administrators to consider it a major concern [11]. The result of SQLIA can be disastrous because a successful SQL injection can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administrative operations on the Database (such as shutdown the DBMS), recover the content on the DBMS file system and execute commands (xp cmdshell) to the operating system [6] The main consequences of these vulnerabilities are attacks on:

2.4.2.1 Authorization

A successful SQLIA can allow an attacker to change user privilege on the web application. The authorization to any certain operations on the database can be changed. Vital information stored on database may be altered if unauthorized access to the SQL database is gained through vulnerabilities in the database [12].

2.4.2.2 Authentication

When username and password are not validated properly, the consequences would be devastating. Anyone would be capable of gaining access to the system without knowing the right username and password [12].

2.4.2.3 Confidentially

A successful SQLIA would violate the confidentiality expected to be derived from storing data in a database because databases are usually used to store delicate and important information. This information must be kept out of the wrong hands. If a system fails, confidentiality of the database is lost and this becomes a problem [12].

2.4.2.4 Integrity

By a successful SQLIA not only an attacker reads sensitive information, but also, it is possible to change or delete this private information [6]. When an attacker is able to change or remove the contents of a database, this results to loss of system integrity. When integrity is compromised, false records can also be created [12].

2.5 SQLIA Defense Techniques

Researchers have proposed many different defense techniques to address the problems of SQLIAs. According to the [13] these different techniques can be categorized as:

2.5.1 By nature of defense

Nature of defense figures out how a technique is going to defend the application from injections. They can be classified into three types:

2.5.1.1 Prevention

Prevention technique averts or severely handicaps the possibility of success of SQLIAs by statically identifying vulnerabilities in the code, proposing a different development paradigm for generating SQL queries, or inspecting the application to enforces best defensive coding practices during development.

2.5.1.2 Detection

Detection technique discriminates SQLIA attempts and preparation from benign activity and alerts the system. It detects SQLIAs mostly during the operation time. After detection of attacks, it alerts the authorities so that they can perform certain actions such as rejecting and escaping of attacks.

2.5.1.3 Deflection

Deflection technique leads an attacker to believe that he has succeeded in an injection attempt whereas the reality is that he has been succeeded to compromise false information only. It is designed is such a way that attackers get easily attracted towards it. This technique helps in learning more about different SQLIAs and their attacking customs. Honeypot is the only one technique that falls under this category.

2.5.2 By detection principle

Each technique has its own criteria to detect the existence of vulnerabilities in the web application. They can be identified into four different detection principles:

2.5.2.1 Grammar based violation

The grammatical structure of SQL statement is the notion of this detection technique to detect SQL injection vulnerabilities. SQL injection occurs when the attacker provides malicious input that will change the structure of SQL query as intended by the application developer. Grammar-based violation detection technique detects the invalid structure of the SQL statement by comparing the parse tree or finite state machines (FSM) built with user input and without user input. A parse tree can be defined as a data structure for the parsed representation of a SQL statement. If the grammatical structures of parse trees are different, it implies that user input is malicious that will change the indented structure of query and will not allow SQL statement to execute.

2.5.2.2 Signature based

Signature can be as simple as a regular expression describing the known attack pattern. The signature-based detection systems maintain a list of possible attack signatures, and

then compare external input strings against the list of signatures at runtime to detect and block SQL Injection related patterns. The idea of signature based detection techniques is to look for known attack patterns to block. We identified two sub categories of signature based detection techniques.

» **Input signature:** This technique detect potential malicious characters by checking external input strings against white list or blacklist. White list is a set of safe (possible correct) values where as blacklist is set of unsafe (negative) values. In white list, external input strings are verified against a set of good input values/patterns/conditions and block anything that is not on the white list. In blacklist, external input strings are verified against set of the negative/bad values/patterns/conditions and sanitize the input by user defined action such as rejecting, escaping (adding a backslash) etc. The single quotation mark (') is one of an example of blacklist.

» **Output signature:** this technique detect potential malicious characters from the output of the web application execution before it will be sent to build the SQL query. It is essential to keep in mind that output often contains user input. Secure output handling is important to prevent from SQL injection attack vulnerabilities.

2.5.2.3 Tainted data flow

The key idea of tainted data flow detection is to detect whether tainted data will reach sensitive sinks in the application. A tainted data is the input from the user which should always be treated as malicious. Sensitive sinks is any point in the application which could lead to security issues when executed over any un-sanitized user input. Tainted data flow detection identifies user inputs and also untrustworthy sources and keeps track of all the data that is affected by those input data. Tainted data flow detection can be further divided in two sub categories.

» **Positive tainting:** this approach identifies and mark trusted data instead of untrusted data. It only allows trusted data to form the semantically correct SQL queries such as SQL keywords and operators.

» **Negative tainting:** this approach identifies and mark un-trusted data instead of trusted data. It basically keeps track of taintedness of data values and checking specifically for malicious contents only in the parts of output that came from un-trustworthy sources.

2.5.2.4 Anomaly detection

Anomaly detection techniques triggers alarm when run time behaviour of application diverges from normal system behaviour which was tracked during training period. It is challenging to identify abnormal behaviour of application at run time. The current state of application is periodically compared with the models of the normal system behaviour to detect anomalies. Anomaly detection techniques can only identify attacks which are modeled during training period.

» **Learning based:** This approach relies on training dataset to build profiles of the normal, benign behaviour of applications. It commonly uses data mining techniques, clustering techniques to characterize the network traffic and identify intrusion patterns. Some techniques use statistical analysis to characterize the user behaviour, while other uses artificial neural network (ANN) to train and learn the normal traffic pattern. Some techniques build legitimate libraries while training and detects the attack using that library.

» **Programmed based:** The description of accepted network behaviour is programmed by network administrator or user to detect anomalous events (which fall outside the model of accepted network behaviour). Thus the user defines the rules on what is considered abnormal enough for an application to alert for security violation. Programmed based anomaly detection uses trained specifications of normal behaviour and generate threshold values for different parameters. Such parameters can be the number of network connections, the number of unsuccessful logins etc. Threshold values define whether to raise the alarm or not. For example, alarm if the number of unsuccessful logins is greater than two.

2.5.3 By analysis method

SQL injection detection techniques use several different analysis methods to detect the existence of vulnerabilities in the web application. They can be identified into six different types:

2.5.3.1 Secure programming

Secure programming is a defensive coding approach to reduce injection vulnerabilities by implementing input validation routines or by using existing standard API or library classes to build the sentence in the source code of application during development. The main drawback of secure programming is that it requires developer training to learn the proper use of secure libraries.

2.5.3.2 Static analysis

Static analysis techniques analyze applications artifacts such as source code, binary code, byte code, and configuration files in order to get information about an application. Information can be how the data would flow at run time without executing the code. Such conservative static analysis can produce high number of false positives.

2.5.3.3 Dynamic analysis

Dynamic analysis techniques analyze the information acquired during program execution to detect SQL injection vulnerabilities. The information might be request and response patterns, structure of queries. Dynamic analysis can be performed at testing time during development or at run time after release. The drawback of dynamic analysis is that it only detects the vulnerabilities in the execution paths but it cannot detect which were not executed in the code.

2.5.3.4 Hybrid analysis

Hybrid analysis uses combination of both static and dynamic analysis to analyze the information obtained during program execution. Some techniques have used hybrid analysis to reduce the performance overhead and increase the efficiency to detect vulnerabilities.

2.5.3.5 Black box testing

Black box testing is a test design methods to detect vulnerabilities by testing application based on requirement specification. Requirement specification means what are the available inputs and the expected outputs that should result from each input. It is not concerned with application source code.

2.5.3.6 White box testing

White box testing is also a test design methods to detect vulnerabilities by testing application with test cases. Test cases are generated from the internal structure of the system i.e. source code.

2.5.4 By detection time

SQLIAs and their vulnerabilities can be detected at various times. They can be classified into three categories:

2.5.4.1 Coding time

If SQLIA vulnerabilities are detected during coding time of an application development cycle, then it is considered as coding time detection. Detecting vulnerabilities in this early stage helps in tumbling the cost caused by tardy detection. Static analysis techniques detect SQLIA vulnerabilities during coding time without the need of code execution.

2.5.4.2 Testing time

If SQLIAs and their vulnerabilities are detected during testing time of an application development cycle, then it is considered as testing time detection. The different testing approaches, such as Black-box testing and White-box testing can be used as analysis methods in testing time for detecting attacks and their vulnerabilities.

2.5.4.3 Operating time

If SQLIAs and their vulnerabilities are detected at run time in the real world field after product is released then it is considered as operation time detection. Run time defense techniques usually prevent SQLIAs by terminating the execution of attacks or

sanitizing them. However, in case of false positives, terminating the execution can lead significant inconvenience to users.

2.5.5 By detection location

SQLIAs and their vulnerabilities can be detected at various locations of the system. They can be classified into four categories:

2.5.5.1 Client- side application

Client-side application techniques detect SQLIAs by analyzing HTML pages. While client side scripts are also analyzed by some techniques which are used for detecting both SQLIA and cross-site scripting.

2.5.5.2 Client-side proxy

Client-side proxy acts as a gateway or intermediate server between a user and a web server. It intercepts user's requests and responses from web server in order to detect SQLIAs. After detecting malicious inputs, either it rejects the request or alters malicious inputs to benign inputs.

2.5.5.3 Server-side application

Server-side application technique detects SQLIAs by analyzing server side application written in programming and script languages.

2.5.5.4 Server-side proxy

Server-side proxy acts as supplementary server between an application server and a database server. It intercepts SQL queries from an application before reaching to database server. It aids in blocking the malicious query execution in database.

2.5.6 By response

Whenever the techniques detect the attacks, it responds to it. They can be classified into five different categories based on the reaction when the SQL injection vulnerabilities are detected.

2.5.6.1 Report

Some of the defense techniques report whenever it detects vulnerabilities in the application. The report often consists of the vulnerable line number of the source code in the application. Static analysis and vulnerability testing tools generates reports.

2.5.6.2 Reject

Some of the defense techniques reject the user requests whenever it figures out that the user input is malicious and blocks the execution.

2.5.6.3 Escape

Some of the defense techniques instead of rejecting the user requests, tries to sanitize by escaping the malicious input. However, escaping malicious input is still vulnerable to SQL second order injection attacks.

2.5.6.4 User defined action

Application developer defines the action whenever they detect malicious input. They can set rules which will escape or encode the user input when a malicious pattern is found. It can be rejecting or escaping.

2.5.6.5 Code suggestion

Some techniques collect information from source code containing SQL Injection vulnerabilities and generate replacement secure code that can maintain applications functional integrity. It suggests the secure code whenever it detects vulnerability.

2.5.7 By implementation

To deploy any techniques, developer needs to know if they require modifying the source code of the application. According to the implementation of the techniques, they can be identified into two categories:

2.5.7.1 Modification of code base

The developer need to modify the source code of the application to deploy the SQL injection defense techniques. Therefore, it is often laborious, time consuming and tedious.

2.5.7.2 No modification of code base.

The developers do not need to modify the source code of the application to deploy the SQL injection defense techniques. It provides flexibility and takes less effort in the implementation of the techniques.

2.6 Formal Specification Overview

2.6.1 Formal Specification Definition

A formal specification is a specification written in a formal language with a restricted syntax and well-defined semantics based on well-established **mathematical concepts**. Formal specifications use a language with **precise semantics**. This *avoids ambiguity* and may allow for proofs of properties about the specification. These languages support **precise descriptions** *of the behavior of system functions* and generally **eliminate implementation details** [14].

2.6.2 Advantages of Formal Specification

» The development of a formal specification provides insights and understanding of the **software requirements and the software design**.

» Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program **conforms to its specifications**.

» Formal specification may be automatically processed. Software tools can be built to assist with their **development, understanding, and debugging.**

» Depending on the formal specification language being used, it may be possible to animate a formal system specification to **provide a prototype system**.

» Formal specifications are mathematical entities and may be studied and analyzed **using mathematical methods.**

» Formal specifications may be used as a guide to the tester of a component in **identifying appropriate test cases** [15].

2.6.3 Formal Specification Languages

Two fundamental approaches to formal specification have been used to write detailed specifications for industrial software systems. These are:

1. An algebraic approach where the system is described in terms of operations and their relationships.

2. A model-based approach where a model of the system is built using mathematical constructs such as sets and sequences and the system operations are defined by how they modify the system state.

Different languages in these families have been developed to specify sequential and concurrent systems. Table 2.1 shows examples of the languages in each of these classes. You can see from this table that most of these languages were developed in the 1980s. It takes several years to refine a formal specification language, so most formal specification research is now based on these languages and is not concerned with inventing new notations [16].

	Sequential	**Concurrent**
Algebraic	Larch (Guttag, *et al.*, 1993), OBJ (Futatsugi, *et al.*, 1985)	Lotos (Bolognesi and Brinksma, 1987),
Model-based	Z (Spivey, 1992) VDM (Jones, 1980) B (Wordsworth, 1996)	CSP (Hoare, 1985) Petri Nets (Peterson, 1981)

Table 2.1 Formal Specification Languages **[16]**

2.7 Petri Nets Language

Petri nets were introduced in 1962 by Dr. Carl Adam Petri. Petri nets are a powerful modeling formalism in computer science, system engineering, and many other disciplines. Petri nets combine a well-defined mathematical theory with a graphical representation of the dynamic behavior of systems. The theoretic aspect of Petri nets allows **precise modeling and analysis of system behavior**, while the graphical representation of Petri nets **enables visualization of the modeled system state changes.** This **combination is the main reason for the great success of Petri nets**. Consequently, Petri nets have been used to model various kinds of dynamic event-driven such as computer networks, communication systems, manufacturing plants, command and control systems, real-time computing systems, logistic networks, and workflows to mention only a few important examples. This wide spectrum of applications is accompanied by wide spectrum different aspects, which have been considered in the research on Petri nets [17].

2.7.1 Basic Definitions of Graph Theory

A **graph** G= (V, E) is a mathematical structure consisting of two set V (vertices/nodes) and E (edges).

Each edge has a set of one or two vertices associated to it, which one called its **endpoints**.

A **loop** is an edge whose endpoints are equal. A non-simple graph with loops is depicted in Figure 2.4. (c).

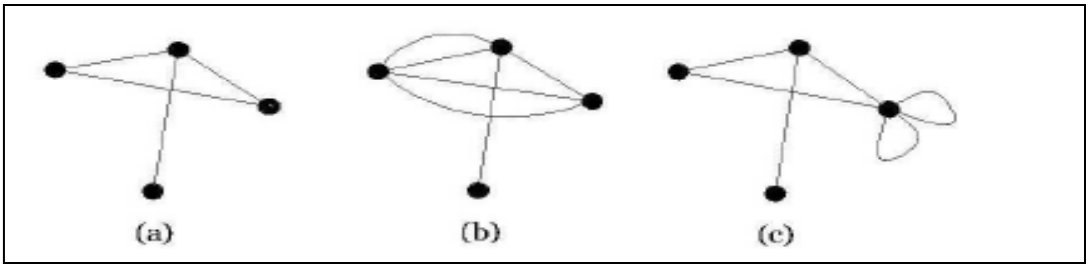

Figure 2.4: (a) A simple graph (b) Anon-simple graph with multiple edges (c) A non-simple graph with loops [18]

A **multi-edge** is a collection of two or more edges having identical endpoints.

A **simple graph** is a graph having no loops or multi-edges.

A **directed graph** is a graph each of whose edges is directed (Digraph).

A **weighted graph** is a graph in which each branch is given a numerical weight. A weighted graph is therefore a special type of labeled graph in which the labels are numbers (which are usually taken to be positive).Graphs with labels attached to edges or vertices are called **labeled graph**; see Figure 2.5.

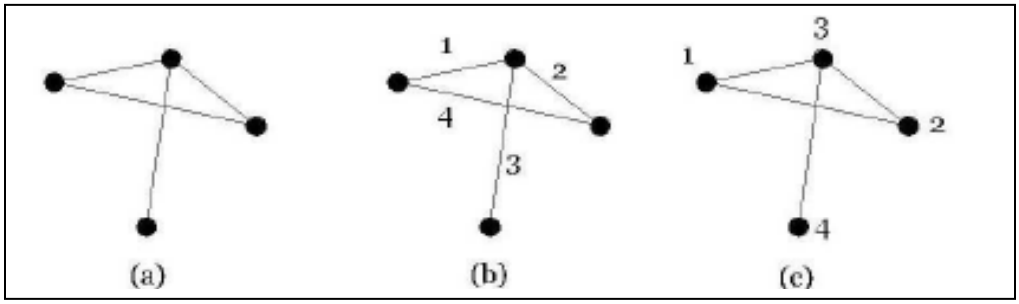

Figure 2.5: (a) Unlabeled graph (b) An edge-label graph (c) A vertex-labeled graph [18]

A **bipartite graph** G is a graph whose vertex set V can be partitioned into two subset U and W, such that each edge of G has one endpoint in U and one endpoint in W [18] see Figure 2.6.

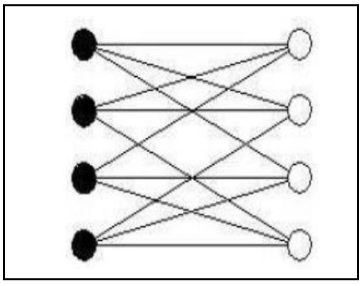

Figure 2.6: A bipartite graph [18]

2.7.2 Basics of Petri Nets

2.7.2.1 A Petri nets

A Petri net is a particular kind of bipartite directed graphs populated by four types of objects. These objects are places, transitions, directed arcs, and tokens [19]; see Figure 2.7, (a).

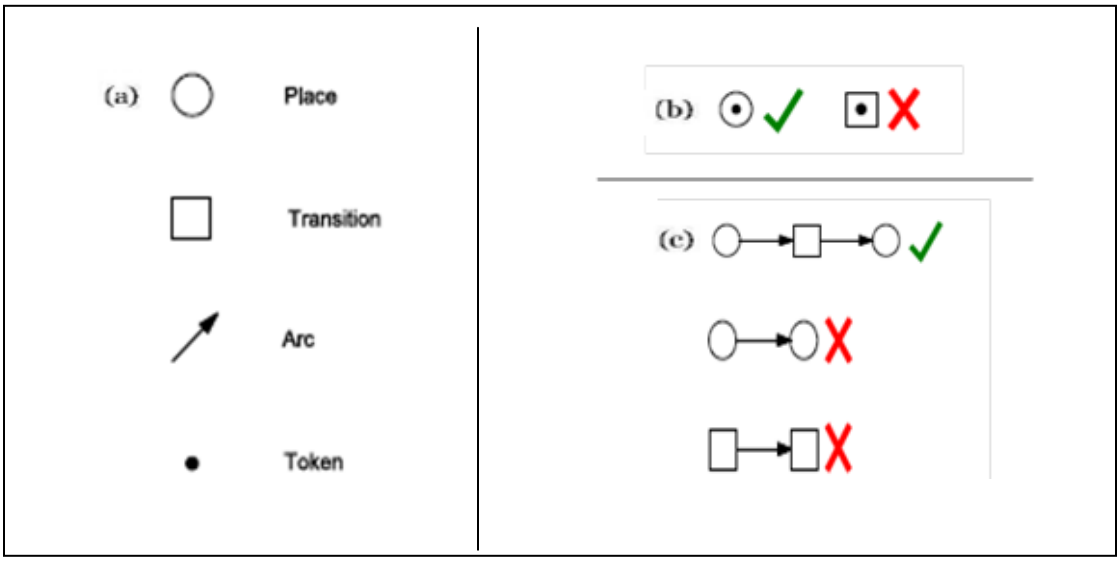

Figure 2.7: Petri Net Formalism. (a) Petri nets consist of places, transitions, arcs and tokens. (b) Just places are allowed to carry tokens. (c) Two nodes of the same type cannot be connected with each other [20].

» **Places** are **passive** nodes. They are indicated by circles and refer to c**onditions or states**. Only places are allowed to carry tokens [20]; see Figure 2.7, (b).

» **Tokens** are **variable elements of a Petri net**. They are indicated as dots or numbers within a place and represent the discrete value of a condition. **Tokens are consumed and produced by transitions.** A Petri net without any tokens is called "empty". The initial marking affects many properties of a Petri net [20], which are considered in section 2.7.3.

» **Transitions**: are **active** nodes and are depicted by squares. They describe **state shifts, system events and activities in a network**. If a place is connected by an arc

with a transition, the place (transition) is called pre-place (post-transition). If a transition is connected by an arc with a place, the transition (place) is called pre-transition (post-place); see Figure 2.8. Transitions consume tokens from its pre-places and produce tokens within its post-places according to the arc weights [20] (firing of transition concept) see example 2.2.

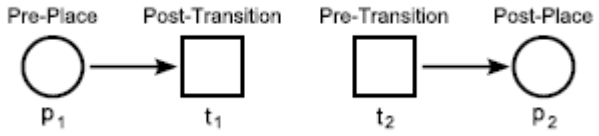

Figure 2.8: Places and Transitions. Place p1 is called pre-place of transition t1, and transition t1 is the post-transition of place p1. Place p2 is called post-place of transition t2, and transition t2 is the pre-transition of place p2 [20].

» **Directed arcs** are **inactive** elements and are visualised by arrows. They specify the **causal relationships between transitions and places** and **indicate how the marking is changed by firing of a transition** [20], which are consider the firing concept in section 2.7.2.3. Arcs connect only nodes of different types; see Figure 2.7, (c). Each arc is connected with an arc weight. **The arc weight sets the number of tokens that are consumed or produced by a transition** [20].

» **Inhibitor arc** is connects an input place to a transition, and is pictorially represented by an arc terminated with a small circle. The presence of an inhibitor arc connecting an input place to a transition changes the transition enabling conditions. In the presence of the inhibitor arc, a transition is regarded as enabled if each input place, connected to the transition by a normal arc (an arc terminated with an arrow), **contains at least the number of tokens equal to the weight of the arc**, and **no tokens are present on each input place connected to the transition by the inhibitor** arc [17]; see Figure 2.20.

2.7.2.2 Formal definitions of Petri Nets

A Petri net is a five-tuple PN = (P, T, F, W, M0) where:

P = { p1, p2, ..., pm} is a finite set of places

T = { t1, t2, ..., tn} is a finite set of transitions

$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs

$W : F \rightarrow \{ 1, 2, ... \}$ is a weight function

$M0 : P \rightarrow \{ 0, 1, 2, ... \}$ is the initial marking

$P \cap T = \Phi$ and $P \cap T = \Phi$

[21]

PN without the initial marking is denoted by N:

N = (P, T, F, W) , PN = (N, M0)

Some authors [22] prefer to use the input-output functions (I and O) in the Petri net definition instead of using set of arcs (F) and a weight function (W)

N = (P,T, I,O)

$I: (P \times T) \rightarrow No+$

$O: (P \times T) \rightarrow No+$

} Arc definition

Weight of arc is defined in the following way. If I (pi ,tj) = k , where k > 1is an integer, a directed arc from place pi to transition t j is drawn with the label (weight) k . If k = 1, an unlabeled arc is drawn and if it happens that k = 0 then no arc is drawn.

Example 2.1 (A Simple Petri net)

Figure 2.9 shows a simple Petri net. In this Petri net, we have

P ={p1, p2, p3, p4};

T ={t1, t2, t3};

I(t1, p1)=2, I(t1, pi)=0 for i=2, 3, 4;

$I(t_2, p_2)=1$, $I(t_2, p_i)=0$ for $i=1, 3, 4$;

$I(t_3, p_3)=1$, $I(t_3, p_i)=0$ for $i=1, 2, 4$;

$O(t_1, p_2)=2$, $O(t_1, p_3)=1$, $O(t_1, p_i)=0$ for $i=1, 4$;

$O(t_2, p_4)=1$, $O(t_2, p_i)=0$ for $i=1, 2, 3$;

$O(t_3, p_4)=1$, $O(t_3, p_i)=0$ for $i=1, 2, 3$;

$M_0=(2\ 0\ 0\ 0)$.

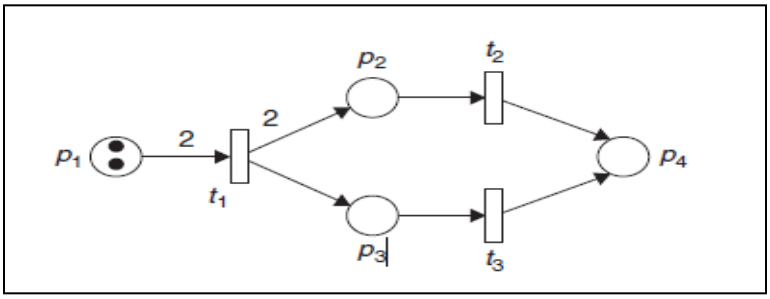

Figure 2.9: A simple Petri net [17]

2.7.2.3 Enabling and firing rules

The execution of a Petri net is controlled by the number and distribution of tokens in the Petri net. By changing distribution of tokens in places, which may reflect the occurrence of events or execution of operations, for instance, one can study the dynamic behavior of the modeled system. A Petri net is executed by *firing* transitions. We now introduce the enabling rule and firing rule of a transition, which govern the flows of tokens:

1) *Enabling Rule*: A transition *t* is said to be *enabled* if each input place *p* of *t* contains at least the number of tokens equal to the weight of the directed arc connecting *p* to *t*, i.e., $M(p) \geq I(t, p)$ for all *p* in *P*. If $I(t, p)=0$, then *t* and *p* are not connected, so we do not care about the marking of *p* when considering the firing of *t*.

2) *Firing Rule*: Only enabled transitions can fire. The firing of an enabled transition *t* removes from each input place *p* the number of tokens equal to *I*(*t*, *p*), and deposits in each output place *p* the number of tokens equal to *O*(*t*, *p*).

Mathematically, firing *t* at *M* yields a new marking

$M'(p) = M(p) - I(t, p) + O(t, p)$ for all p in P

Note that since only enabled transitions can fire, the number of tokens in each place always remains nonnegative when a transition is fired. Firing a transition can never try to remove a token that is not there.

The transition firings rule to inhibitor arc the same for normally connected places. The firing, however, does not change the marking in the inhibitor arc connected places [17]. A Petri net with an inhibitor arc is shown in Figure 2.19.

Example 2.2 (Firing of Transition)

Consider the simple Petri net shown in Figure 2.9. Under the initial marking,

M0= (2 0 0 0), only t1 is enabled. Firing of t1 results in a new marking, say M1. It follows from the firing rule that

M1 = (0 2 1 0)

The new token distribution of this Petri net is shown in Figure 2.10. Again, in marking M1, both transitions of t2 and t3 are enabled. If t2 fires, the new marking, say M2, is

M2 = (0 1 1 1)

If t3 fires, the new marking, say M3, is

M3 = (0 2 0 1)

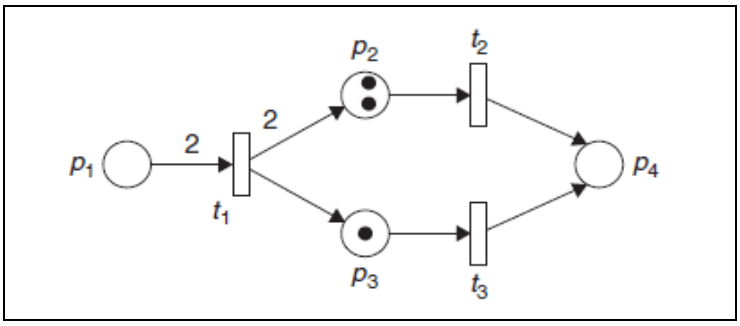

Figure 2.10: Firing of Transition t1. [17]

A transition without any input place is called a *source transition*, and one without any output place is called a *sink transition*. Note that a source transition is unconditionally enabled, and that the firing of a sink transition consumes tokens, but does not produce tokens [19]; see Figure 2.11.

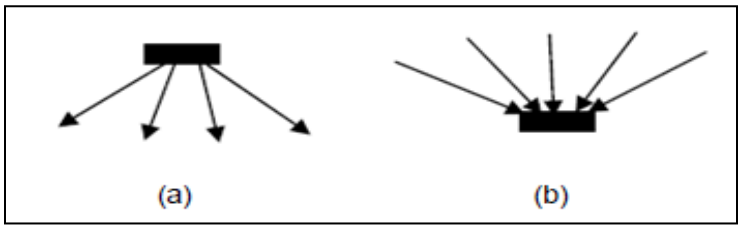

Figure 2.11: (a) Source Transition (b) Sink Transition [22]

A pair of a place *p* and a transition *t* is called a *self-loop*, if *p* is both an input place and an output place of *t*. A Petri net is said to be *pure* if it has no self-loops [19]. Any impure Petri net (Petri a net having self-loops) can be made pure by ad**ding appropriate dummy places and transitions to it [22]**; see Figure 2.12.

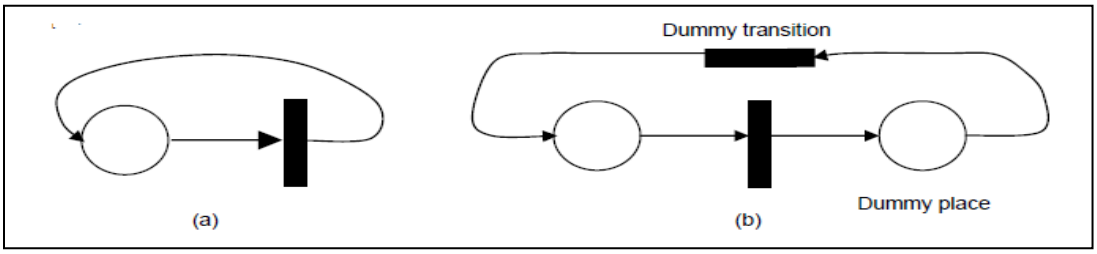

Figure 2.12: (a) Impure Petri net, (b) Pure Petri net [22]

2.7.3 Properties of Petri Net

As a mathematical tool, Petri nets possess a number of properties. These properties, when interpreted in the context of the modeled system, allow system designer to identify the presence or absence of the application domain specific functional properties of the system under design. Two types of properties can be distinguished, behavioral and structural ones. The **behavioral properties are those which depend on the initial state or marking of a Petri net**. **The structural properties, on the other hand, do not depend on the initial marking of a Petri net**. They depend on the topology, or net structure, of a Petri net. Here we provide an overview of some of the most important, from the practical point of view, behavioral properties: **reachability, safeness, and liveness** [17]. For more details about the rest of the properties can review [22] [18] [21].

2.7.3.1 Reachability

An important issue in designing event-driven systems is whether a system can reach a specific state, or exhibit a particular functional behavior. In general, the question is whether the system modeled with a Petri net **exhibits all desirable properties as specified in the requirement specification, and no undesirable ones.**

To find out whether the modeled system can reach a specific state as a result of a required functional behavior, it is necessary to find such a transition firing sequence that would transform its Petri net model from the initial marking M_0 to the desired marking M_j, where M_j represents the specific state, and the firing sequence represents the required functional behavior. In general, a marking M_j is said to be *reachable* from a marking M_i if there exists a sequence of transition firings that transforms M_i to M_j. A marking M_j is said to be *immediately reachable* from M_i if firing an enabled transition in M_i results in M_j. The set of all markings reachable from marking M is denoted by $R(M)$ [17].We will explain how to get $R(M)$ in section 2.9.1.

2.7.3.2 Safeness

The Petri net property, which helps to identify the existence of overflows in the modeled system, is the concept of *boundedness*. A place *p* is said to be *k-bounded* if the number of tokens in *p* is always less than or equal to *k* (*k* is a nonnegative integer number) for every marking *M* reachable from the initial marking M_0, i.e.,$M \in R(M_0)$. It is *safe* if it is 1-bounded.

A Petri net $N = (P, T, I, O, M_0)$ is *k*-bounded (safe) if each place in *P* is *k*-bounded (safe). It is *unbounded* if *k* is infinitely large. For example, the Petri net of Figure 2.9 is 2-bounded, but the net of Figure 2.10 is unbounded [17].

2.7.3.3 Liveness

The concept of liveness is closely related to the *deadlock* situation, which has been situated extensively in the context of computer operating systems.

A Petri net modeling a deadlock-free system must be *live*. This implies that for any reachable marking *M*, any transition in the net can eventually be fired by progressing through some firing sequence. This requirement, however, might be too strict to represent some real systems or scenarios that exhibit deadlock free behavior. For instance, the initialization of a system can be modeled by a transition (or a set of transitions) that fires a finite number of times. After initialization, the system may exhibit a deadlock-free behavior, although the Petri net representing this system is no longer live as specified above. For this reason, different levels of liveness are defined. Denote by $L(M_0)$ the set of all possible firing sequences starting from M_0. A transition *t* in a Petri net is said to be

(1) *L*0-live (or dead) if there is no firing sequence in $L(M_0)$ in which *t* can fire.

(2) *L*1-live (potentially firable) if *t* can be fired at least once in some firing sequence in $L(M_0)$.

(3) *L*2-live if *t* can be fired at least *k* times in some firing sequence in $L(M_0)$ given any positive integer *k*.

(4) $L3$-live if t can be fired infinitely often in some firing sequence in $L(M_0)$.

(5) $L4$-live (or live) if t is $L1$-live (potentially firable) in every marking in $R(M_0)$.

For example, Transitions t0, t1, t2, t3 are L0 live (dead), L1 live, L2 live and L3 live respectively in the net of Figure 2.13

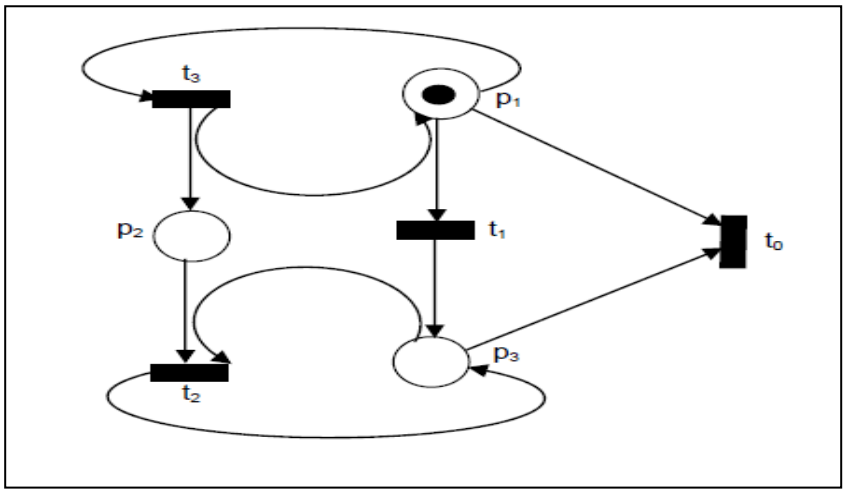

Figure 2.13: Transitions t_0, t_1, t_2, t_3 are L0 live (dead), L1 live, L2 live and L3 live respectively [22]

2.8 Modeling with Petri Nets

The success of any model depends on two factors: its modeling power and its decision power. Modeling power refers to the ability to correctly represent the system to be modeled; decision power refers to the ability to analyze the model and determine properties of the modeled system [23]. The modeling power of Petri Nets has been examined in this section and in next section we take into consideration the analysis techniques of Petri Nets.

2.8.1 Basic Modeling Constructs

In this section, some basic situations are taken which are encountered often during modeling a physical system. This section describes how Petri net handles these real life modeling situations, thus revealing the modeling power and ease of representation of Petri nets [22].

2.8.1.1 Sequential execution

Sequential execution poses a precedence constraint among the activities (transitions). In Figure 2.14 transition t2 can fire only after the firing of t1.

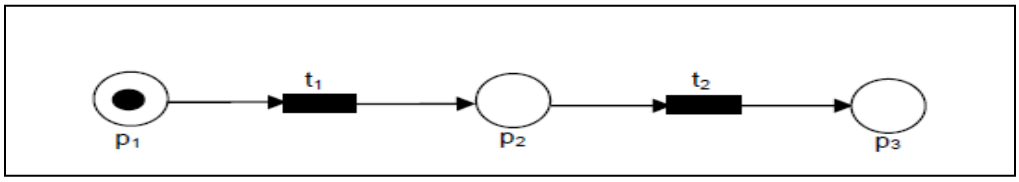

Figure 2.14: Transition t1 occurs first and then transition t2 occurs [22]

2.8.1.2 Synchronization

Petri nets can successfully capture the synchronization mechanism in the modeling phase. In Figure 2.15 transition t1 will fire only when the empty input place gets a token. Thus, the three input places of t1 are synchronized for the firing of transition t1.

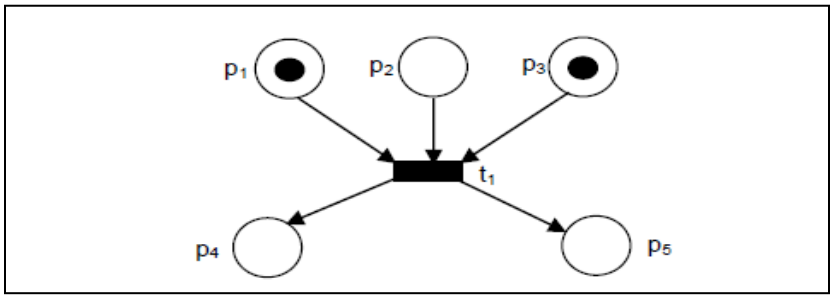

Figure 2.15: Transition t1 fires when the place p2 gets a token so that all the input places of transition t1 have tokens [22].

2.8.1.3 Conflict

In Figure 2.16 transitions t_1, t_2 and t_3 are in conflict. All three transitions are enabled but only one can fire at a time. Hence, choice has to be made regarding which transition will be fired. Firing one will lead to the disabling of other transitions.

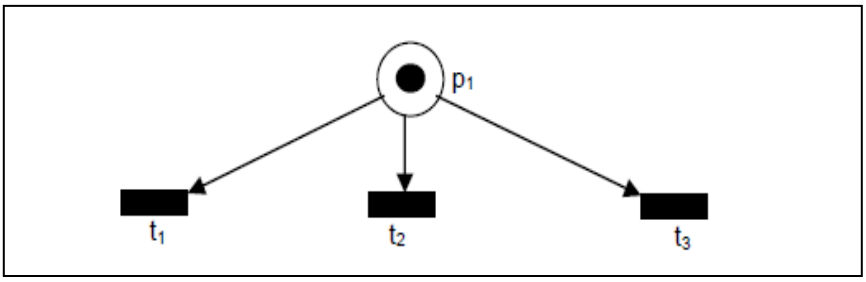

Figure 2.16: Transition t1 occurs first and then transition t2 occurs [22]

2.8.1.4 Concurrency

In Figure 2.17 transitions t1, t2 and t3 are concurrent. Concurrency is characterized by the existence of a forking transition that deposits tokens simultaneously in two or more output places. In Figure 2.16 t_0 is the forking transition.

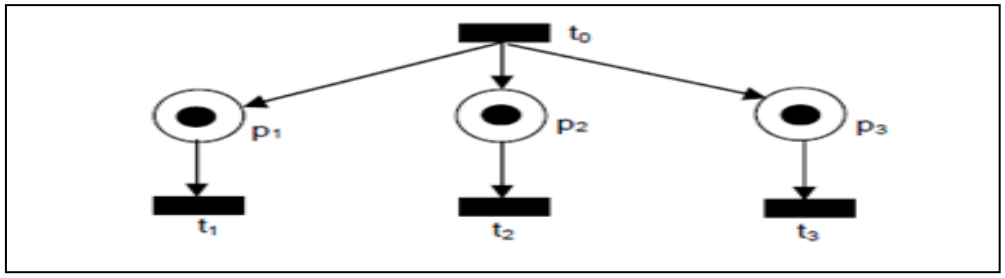

Figure 2.17: Transitions t1, t2 and t3 are concurrent [22]

2.8.1.5 Confusion

Confusion occurs when conflict and concurrency co-exist. In such a situation, it is not clear that whether a conflict is needed to be resolved or not, in going to the new state (marking). In Figure 2.18 transitions t_1 and t_3 are concurrent whereas transitions t_1 and t_2 are in conflict. Also t_2 and t_3 are in conflict.

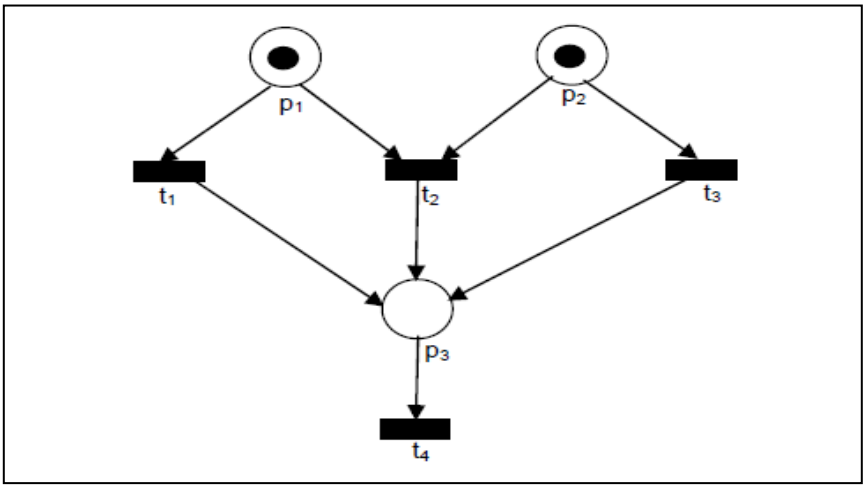

Figure 2.18: Transitions t1, t2 and t2, t3 are in conflict but t1, t3 are concurrent [22]

Confusions can be of two types: Symmetric Confusion and Asymmetric Confusion. Figure 2.19 (a) shows Symmetric Confusion where t_1 and t_3 are concurrent (both enabled and firable) and at the same time they are in conflict with t_2.

In Figure 2.19 (b), t_1 and t_2 are concurrent and if t_1 fires first, then t_3 and t_2 will be in conflict. This situation is called Asymmetric Confusion. Asymmetric confusion occurs when one place feeds to a set of transitions via output arcs from it and there exists another place in the net which feeds to a subset of those transitions. In Figure 2.19 (b) the place p_2 feeds to a set of transitions $\{t_2, t_3\}$ via output arcs from p_2 and there exists a place p_3 in the net which feeds to $\{t_3\} \subseteq \{t_2, t_3\}$.

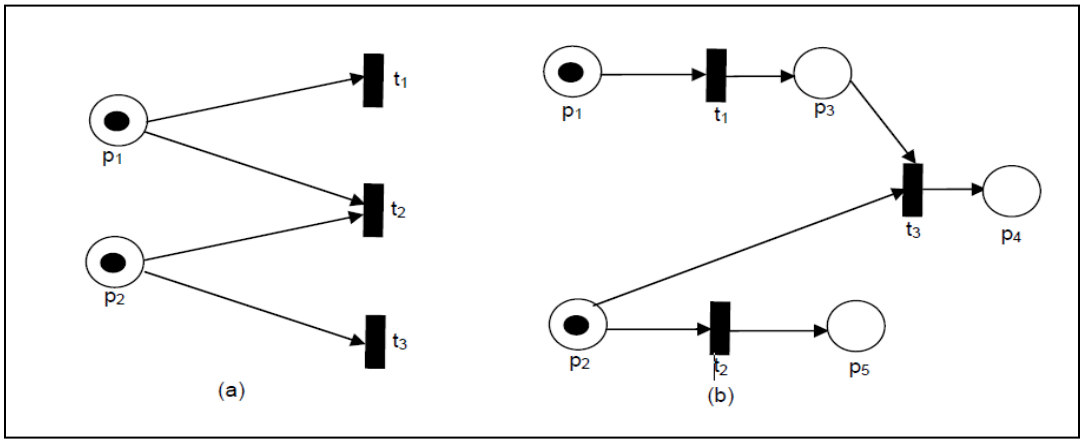

Figure 2.19: (a) Symmetric Confusion (b) Asymmetric Confusion [22]

2.8.2 Primitives for Programming Constructs

This section describes basic programming constructs in Petri net formalism. This, in turn, will express the modeling power of Petri nets and these constructs will be used in subsequent modeling examples [22].

2.8.2.1 Selection (if – else)

a) If condition A then do activity X, else do activity Y.

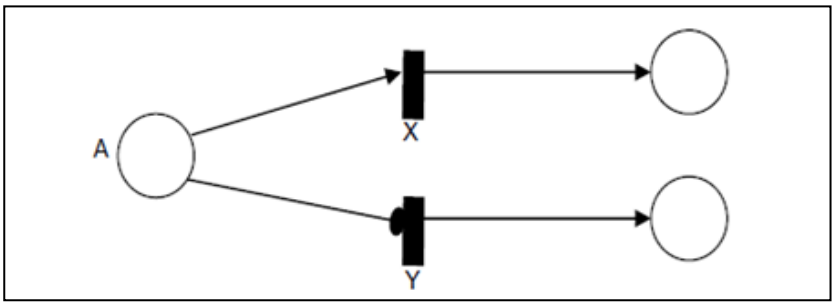

Figure 2.20: If – else condition [22]

b) If condition A and condition B hold, then do activity X.

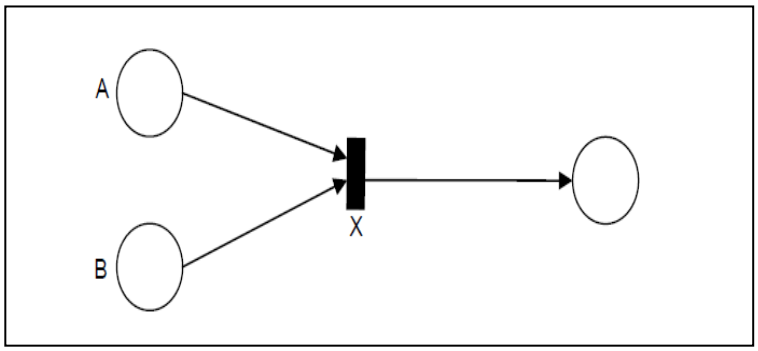

Figure 2.21: If – else with and operator [22]

2.8.2.2 Case (Switch) statement

If *Case A* do activity P, if *Case B* do activity Q, if *Case C* do activity R, if *Case D* do activity S.

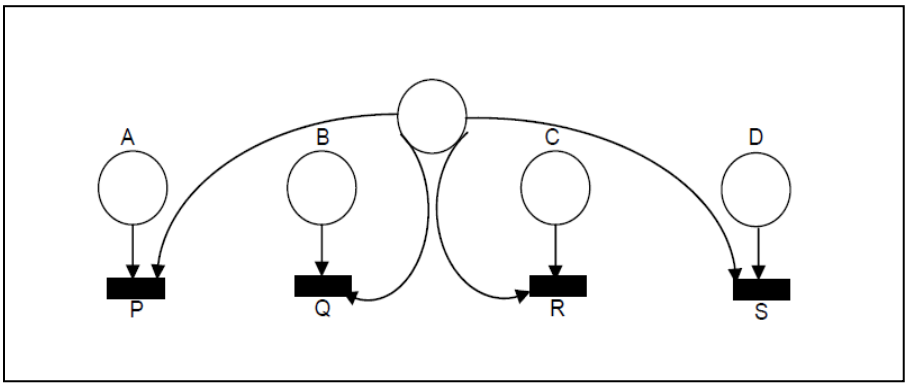

Figure 2.22: (a) Switch statement [22]

2.8.2.3 While loop

While condition A holds, *do* activity X.

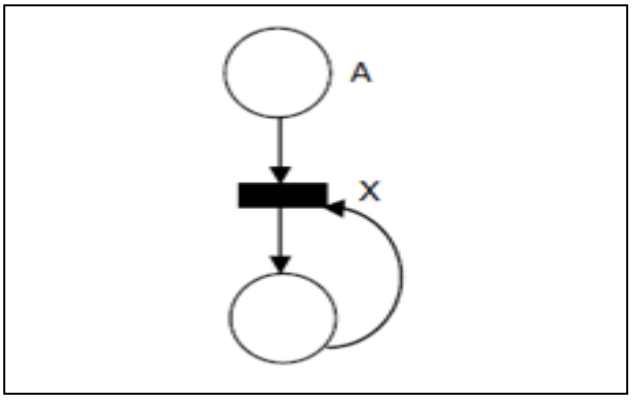

Figure 2.23 While loop [22]

2.8.2.4 Repeat (for) loop

For condition A, do activity X.

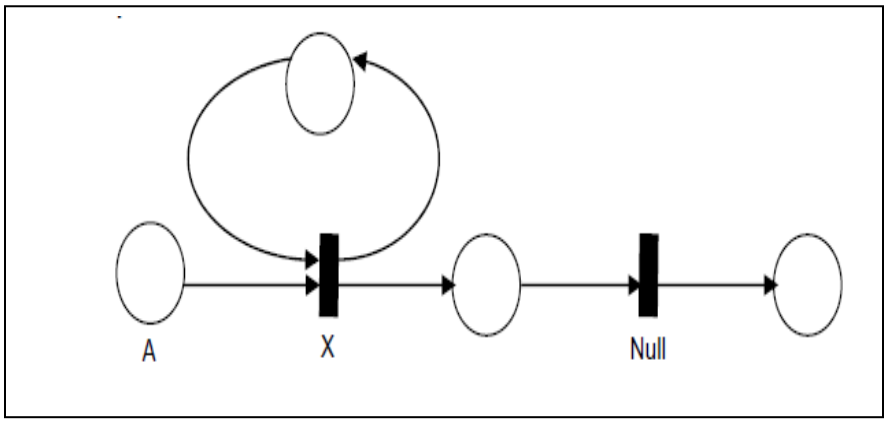

Figure 2.24: For loop [22]

2.8.2.5 Precedence

Activity X should *precede* activity Y.

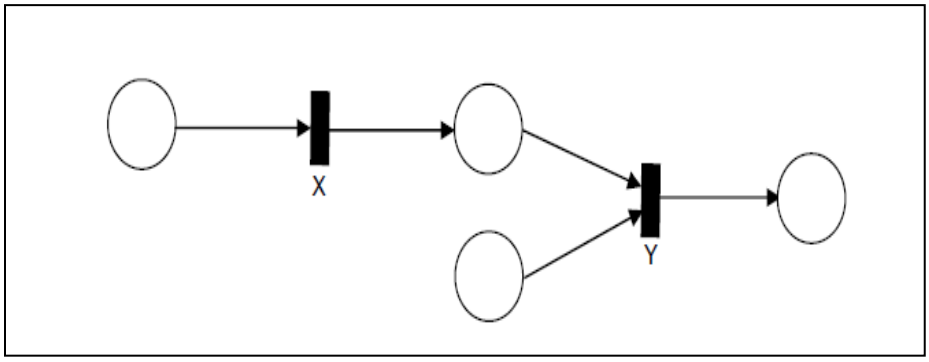

Figure 2.25: Precedence relation [22]

2.8.2.6 Timed occurrence

After k seconds do activity X

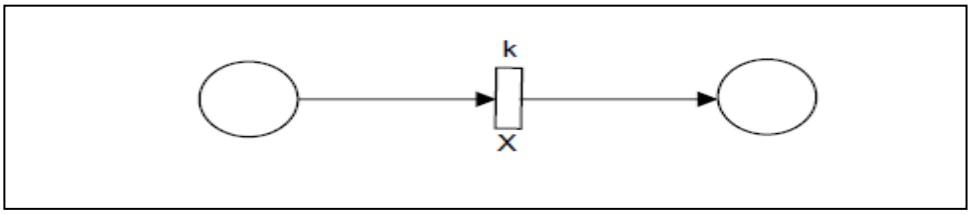

Figure 2.26: Timed transition [22]

2.8.2.7 Either – or (Mutual exclusion)

a) Either do activity X or do activity Y.

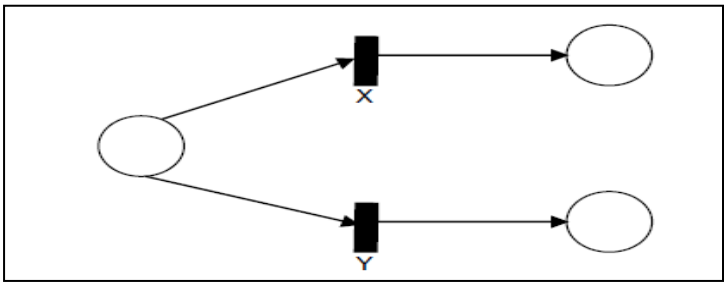

Figure 2.27: Either – or statement [22]

b) Either do activity X or do activity Y with preference to activity X (preferential either - or)

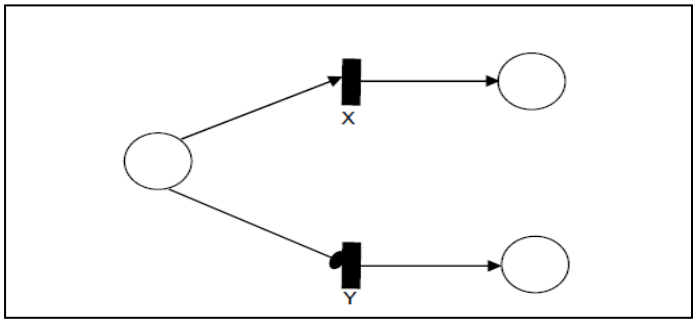

Figure 2.28: Preferential either – or statement [22]

2.9 Analysis of Petri Nets

We have introduced the modeling power of Petri nets in the previous sections. However, modeling by itself is of little use. It is necessary to *analyze* the modeled system. This analysis will hopefully lead to important insights into the behavior of the modeled system [17].

Some of the methods used for modeling and analyzing systems with Petri nets are the reachability tree and incidence matrix [21].

2.9.1 Reachability Analysis

Reachability analysis is conducted through the construction of reachability tree if the net is bounded. Given a Petri net *N*, from its initial marking M_0, we can obtain as many "new" markings as the number of the enabled transitions. From each new marking, we can again reach more markings. Repeating the procedure over and over results in a tree representation of the markings. Nodes represent markings generated from M_0 and its successors, and each arc represents a transition firing, which transforms one marking to another.

The above tree representation, however, will grow infinitely large if the net is unbounded. To keep the tree finite, we introduce a special symbol ω, which can be thought of as "infinity." It has the properties that for each integer n,$\omega > n$, $\omega + n = \omega$, and $\omega \geq \omega$. Generally, we do not know if a Petri net is bounded or not before we perform the **reachability** analysis. However, we can construct a ***coverability tree*** if the net is unbounded or a ***reachability tree*** if the net is bounded according to the following general algorithm:

1. Label the initial marking M0 as the root and tag it **"new."**

2. For every new marking M:

   2.1 If M is identical to a marking already appeared in the tree, then tag M **"old"** and go to another new marking.

   2.2 If no transitions are enabled at M, tag M **"dead-end"** and go to another new marking.

   2.3 While there exist enabled transitions at M, do the following for each enabled transition t at M:

      2.3.1 Obtain the marking M_ that results from firing t at M.

2.3.2 On the path from the root to M if there exists a marking M" such that M' (p)≥ M" (p) for each place p and M'≠M", i.e., M" is coverable, then replace M' (p) by ω for each p such that M' (p)>M" (p).

2.3.3 Introduce M' as a node, draw an arc with label t from M to M', and tag M' "new."

If ω appears in a marking, then the net is unbounded and the tree is a coverability tree; otherwise, the net is bounded and the tree is a reachability tree. Merging the same nodes in a coverability tree (Reachability tree) results in a coverability graph (*reachability graph*) [17].

Example 2.3 (Reachability analysis)

Consider the Petri net shown in Figure 2.9. All reachable markings are M_0=(2, 0, 0, 0), M_1=(0, 2,1, 0), M_2=(0, 1, 1, 1), M_3=(0, 2, 0, 1), M_4=(0, 0, 1, 2), M_5=(0, 1, 0, 2), and M_6 =(0, 0, 0, 3).

The reachability tree of this Petri net is shown in Figure 2.29(a), and the reachability graph is shown in Figure 2.29(b).

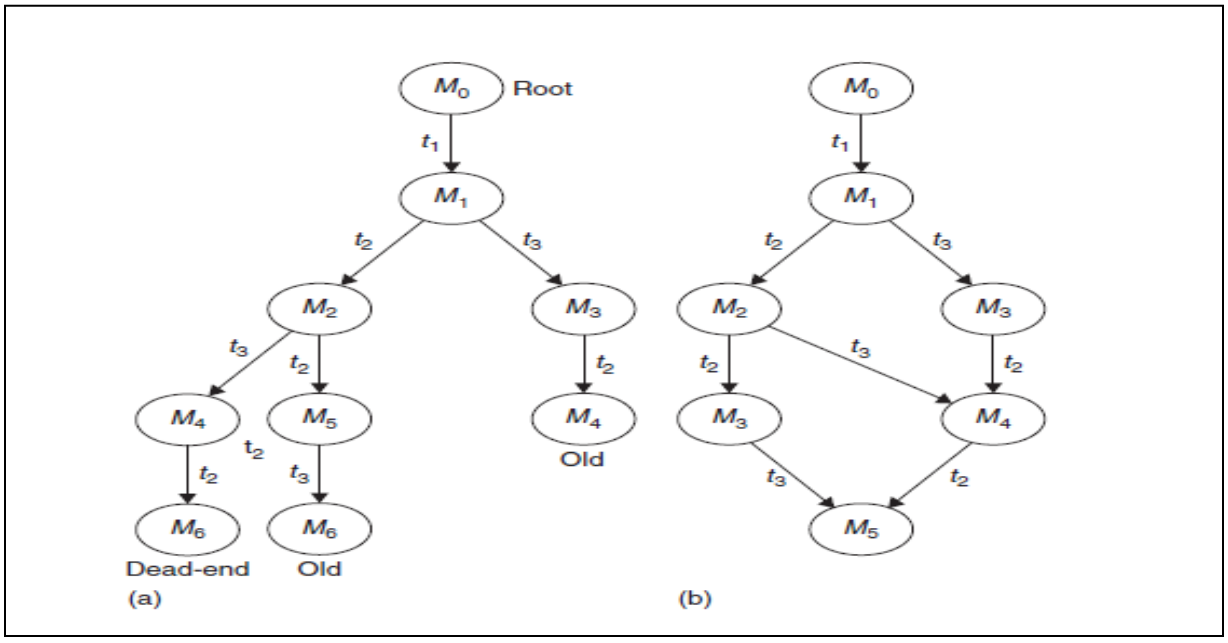

Figure 2.29: (a) Reachability tree. (b) Reachability graph. [17]

2.9.2 Incidence Matrix Analysis

An alternative method for representation and analysis of Petri nets is based on matrix equations used to represent the dynamic behavior of Petri nets. The method involves constructing the **incidence matrix** that defines all possible interconnections between places and transitions. The incidence matrix of a Petri net is an matrix, where is the number of transitions and is the number of places [21].

Incidence Matrix: For a Petri net PN with n transitions and m places, the incidence matrix $A = [a_{ij}]$ is an $n \times m$ matrix of integers and its typical entry is given by;

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-}$$

where $a_{ij}^{+} = w(i, j)$ is the weight of the arc from transition i to its output place

j and $a_{ij}^{-} = w(i, j)$ is the weight of the arc to transition i from its input place j.

Transition i is enabled at marking M iff

$$a_{ij}^{-} <= M(j), j = 1, 2, \ldots, m \text{ [18]}$$

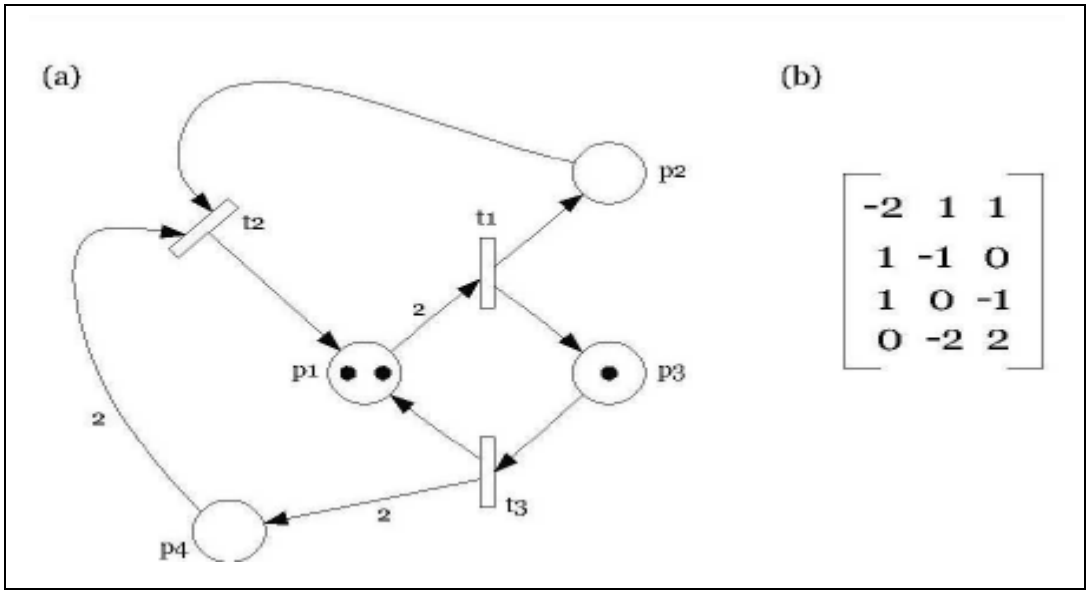

Figure 2.30: (b) The incidence matrix of a given Petri net in (a). [18]

2.10 Related works

Jaskanwal Minhas and Raman Kumar [24] proposed a technique to detect SQLIAs, which uses combined static and dynamic analysis technique. In this, work stored the valid query structure (static queries) in a database. And in runtime removed attribute values of dynamic queries and compared with previously stored static queries having the same number of tokens as in dynamic query. If a match is found requested dynamic query is valid query otherwise it is SQL Injection Attack.

The advantages of a proposed system are: Firstly, reduce false positives and a false negative by using a model is combined static and dynamic analysis technique. Secondly, it can improve response time by comparing dynamic queries only with that static query having the same number of tokens. Thirdly, it simplicity framework because is complexity of the algorithm is divided into two parts- first token calculation and second searching for dynamic query. Fourthly, it defines and detects a new type of SQLIA known as white space manipulation. Fifthly, SQL query independent of the database by removing of attribute values from SQL query.

This research didn't refer to any limitation but also didn't refer to an ability of detect new SQLIA forms.

Diksha Gautam Kumar and Madhumita Chatterjee [25] proposed a block model against SQL injection attacks. The model works both on client and server side. Client side implements a filter program that checks the length and data type of the submitted variables, and detects the injection-sensitive characters and keywords. Server side is based on entropy in information theory, and it works in two phases training and detection. In training phase first to compute the static entropy of each query in the source code based on complexity the entropy and is derived from token's probability distribution. Next, apply Message authentication code (MAC) on entropy. Finally, this entropy is stored in a database. In detection, Phase first created entropy to dynamic query in run time. Next, apply MAC on entropy calculated from first step. Finally

compare entropy stored in a database (static) with dynamic query to detect an attack. Client side and server side are shown in figure 2.31 & figure 2.32 respectively.

The advantages of a proposed system are: Firstly, client side reduces CPU cycles since it avoids a number of round trips to the server. Secondly, it can detect all known SQLI attacks. Thirdly, it can reveal several unknown vulnerabilities. It does not rely on the specific type of attack inputs. Fourthly, does not require tainted data flow analysis or complex static analysis. Fifthly, can be applied for a wide variety of scripting languages and by applying Mac; we provide an additional layer of security.

The Limitations of a proposed system are: In the client, sides are firstly limiting the size of input and restricting the use of special characters cannot be imposed on user in all applications. Secondly, the protection provided by client side scripts can be easily bypassed. This approach does not address the SQLIA in stored procedures.

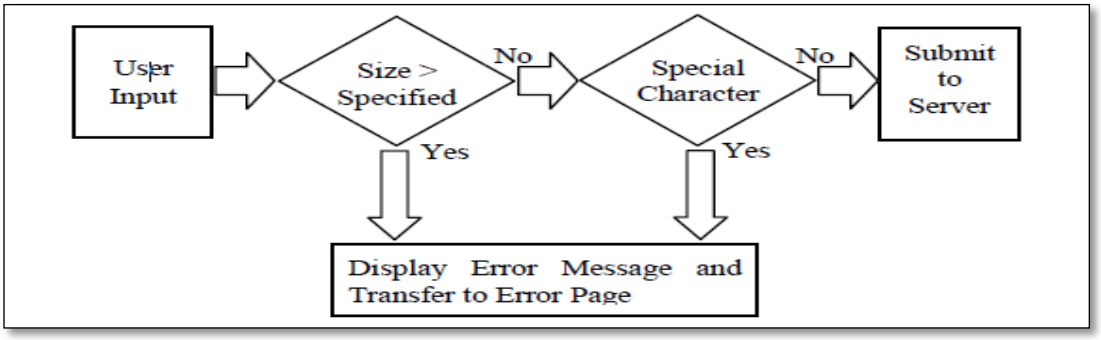

Figure 2.31: Client Side Framework [25]

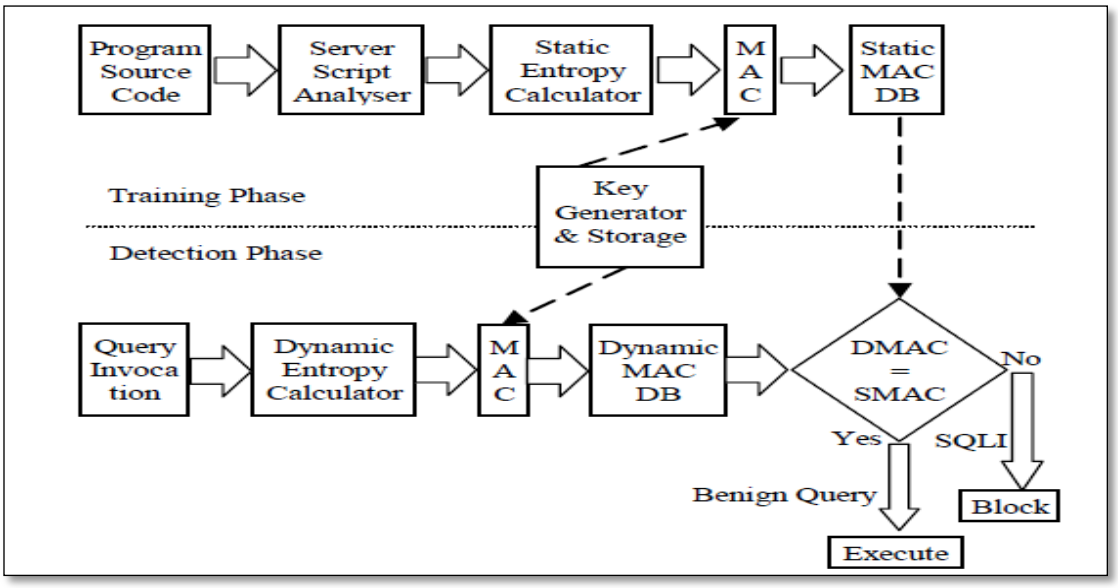

Figure 2.32: Server Side Framework [25]

Witt Yi Win and Hnin Hnin Htun [3] proposed a framework to detect SQL injection attacks. The main idea of this framework is combined static analysis and Runtime Monitoring as shown in figure 2.33. In its static analysis, part uses program analysis technique to automatically build the abstract legitimate queries that could be generated by the application and after that store, these abstract legitimate queries separately according to the query statement in a master database. In its dynamic part, monitors the dynamically generated queries at runtime and checks them with the statically-generated queries pervious stored in a master database. In case not matched, then it is flagged as SQLIA, else it is passed.

The advantages of a proposed system are**:**, this framework is eliminated the problem of false negatives that may result from the incomplete identification of all untrusted data sources because is based on positive tainting, which explicitly identifies trusted (rather than untrusted) data in a program. Secondly, it can reduce the runtime scanning overhead by restricting the number of queries that need to be scanned along any execution path that is taken in the program.

This research didn't refer to any limitation but also didn't refer to an ability of detect all types of SQLIA; it was referring to five types in evaluation.

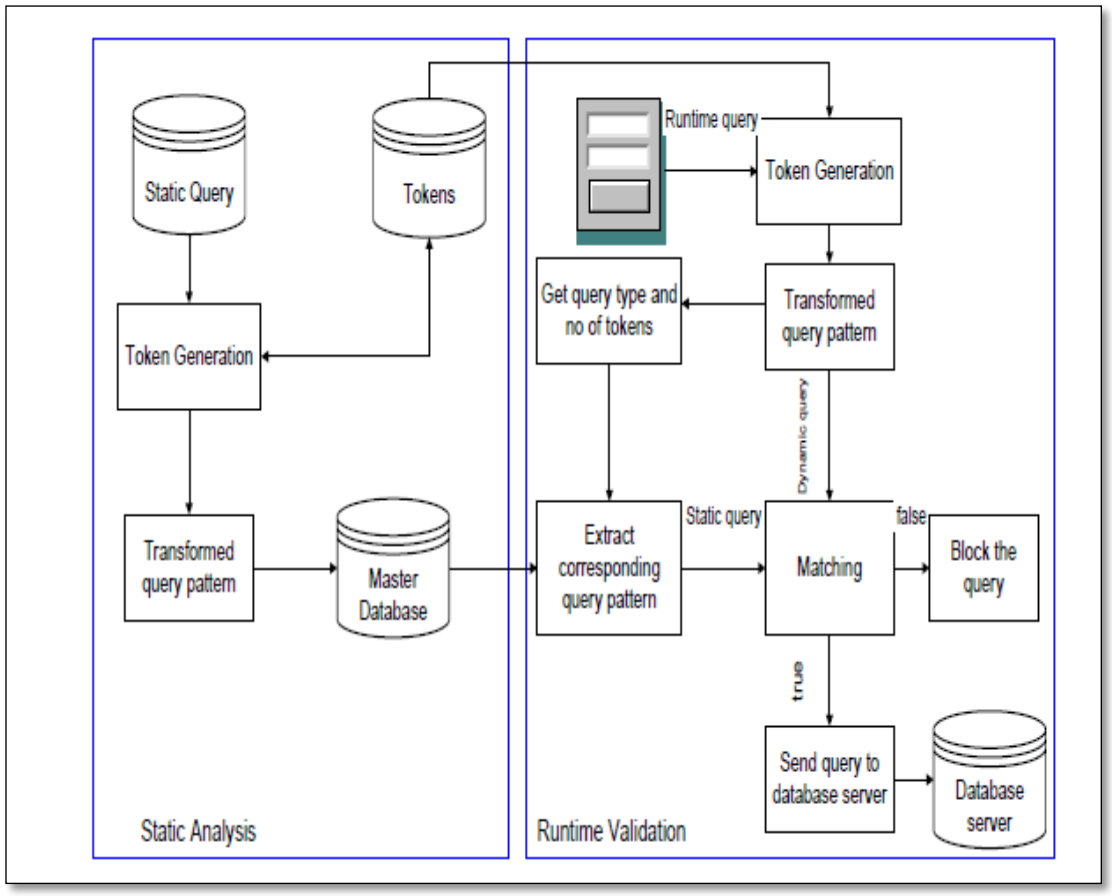

Figure 2.33: Overview of the Proposed System [3]

Reshma Rai and Jitendra Jadhav [26] proposed a technique that uses a concept of filter called "Smart Filter," that avoid the SQL injections with static matching and dynamic signature based intrusion-detection mechanism. This smart filter actually works in between the web application & database server. Therefore, before sending SQL queries to the database, the smart filter will analyze the query to check the vulnerability. If it found any, it reported else it forwards the query to the database server. Apart from the checking, the SQL query by smart filter, it also reports the new vulnerabilities found in SQL queries. This technique implements in three modules injection parser module, signature based detection module and threat recorder module all of these modules are shown in Figure 2.34.

a) **Injection parser module:** It used a recursive descent parser to ensure the administrator that; the query does not contain any vulnerable character.

b) Signature based detection module: It is the core part of the proposed technique. It works when: Query may have special characters or injected query cannot detect by the injection parser module. It can upgrade the knowledge using supervised-learning; the administrator can update the knowledge of the system periodically.

c) Threat recorder module: It is developed for the auditing purpose, as it generates the reports that help the administrator to identify the errors, choose the signatures to upgrade the system knowledge. This module and log file recording is also important to keeping track of applications that have little to no human interaction.

The advantages of a proposed system are: Firstly, it provides a standard and common guideline for the evaluation process of detection and prevention of SQL injection attack tools in general without any restriction or limitations. Secondly, database and operating system independent. Thirdly, it provides a complete evaluation by analyzing different aspects of the tool.

The Limitation of a proposed system is: a language dependent; one has to migrate the logic to other language.

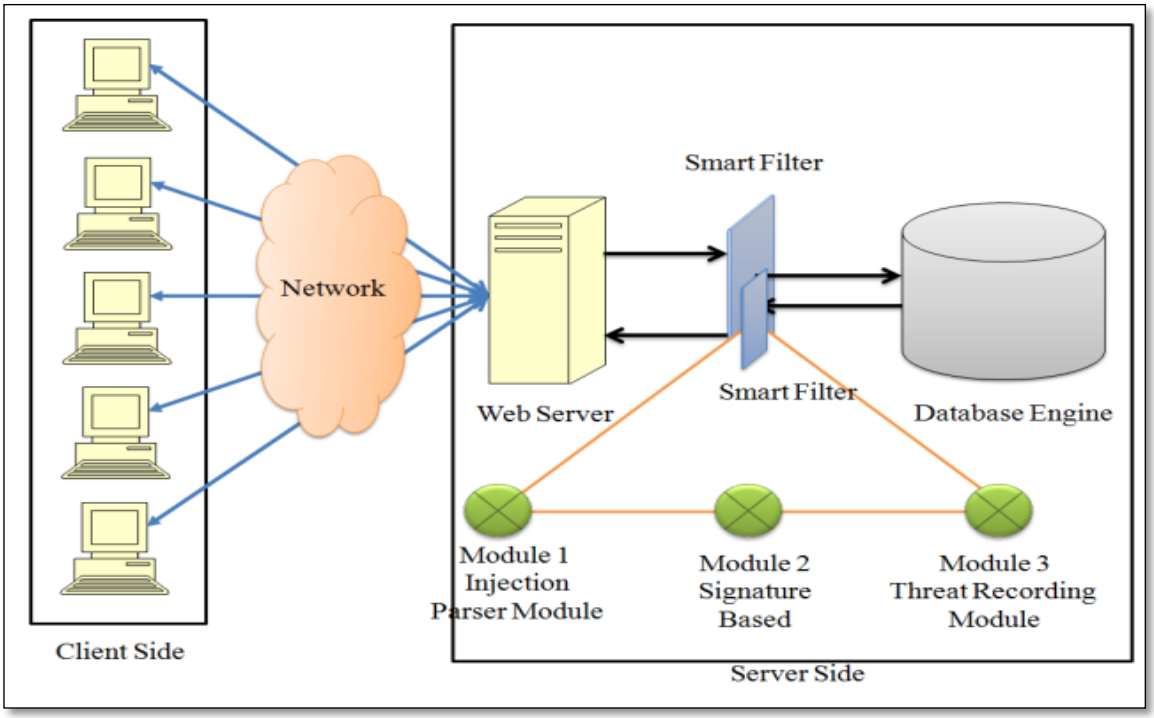

Figure 2.34: Details of Smart Filter [26]

Jalal Omer Atoum and Amer Jibril Qaralleh [27] proposed a hybrid technique combined static and runtime SQL queries analysis to create a defense strategy that can detect and prevent various types of SQL injection attacks. This technique is done in two main phases: runtime analysis, and static analysis. The first phase is a dynamic/runtime analysis method that depends on applying tracking methods to trace and monitor the execution processes of all received queries. The result of affected objects of this monitoring will be compared with a prepared set of expected changes that the developer had created before, and the result of this comparison process will decide if there is an existence of any type of SQLIA, and if so they will be forwarded to the following phase. The next phase is a static analysis phase that is performing a string comparison between the received SQL queries and previous expected SQL queries to prevent any query that is described as a suspicious query. This technique is based on different stages to reject any malicious query from being passed through the database engine before its execution process.

The advantages of a proposed system are: Firstly, it can detect and prevent SQLIAs that are performed through the system or through a direct SQL query to the database. Secondly, it is the only one that can detect and prevent SQLIAs that are using Built-In functions to perform such attacks.

The Limitations of a proposed system are: the time delay that the database recovery takes after the SQLIA is detected is needed to increase, also didn't refer to an ability of detect new SQLIA forms.

Pranita Talekar, et al [28] proposed a technique for detecting and preventing SQLIA using both static phase and dynamic phase. This technique uses static Anomaly Detection using Aho–Corasick Pattern matching algorithm. In Static Phase, the user generated SQL Queries Compared with the stored in Static Pattern List (list of known Anomaly Pattern), If the pattern is exactly match with one of the stored pattern then the SQL Query is affected with SQL Injection Attack. In Dynamic Phase, if any new

anomaly is occur then Alarm will indicate and new Anomaly Pattern will be generated. The new anomaly pattern will be updated to the Static Pattern List.

The results of this technique show that model protects against 100% of tested attacks before reaching the database layer.

The Limitation of a proposed system is not eliminated the problem of false negatives that may result from the incomplete identification of all Patterns because is based on known Anomaly Pattern.

Ammar Alazab, et al [29] proposed a general model for protecting web applications based on SQL syntax at the web application layer, and negative taint at the database layer. It performs negative taint by storing untrusted markings, based on the evasion methods, at the database layer. Also, performs syntax-aware evaluation in web application server of query strings, before executing the query in the database, by validating queries whose input matches with untrusted markings that contain one or more characters without trust markings, the matching process done with SQL keywords and operators.

Applying negative taint in database layer helps us to identify untrusted data in the database layer. Also, able to detect maliciousness caused by tricky data and obfuscation techniques while and minimizing false negatives. The main challenge, is that if the username and password correct not always led to a legitimate query.

The major advantage of a proposed system is: apart from efficiently is that it does not change the web architecture.

The Limitation of a proposed system is: test the model with a larger dataset and with a more comprehensive vulnerabilities lookup for various other obfuscations.

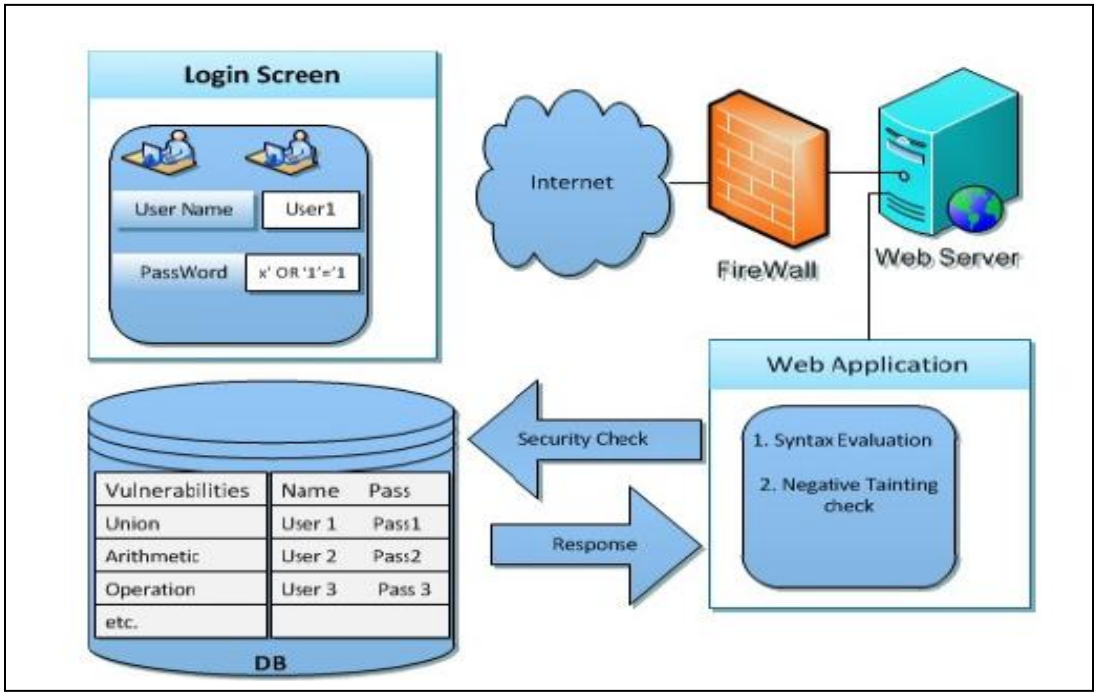

Figure 2.35: Methodology of the Proposed System [29]

In our model we integrate hybrid techniques, combined static and dynamic analysis, which are used in studies 1, 2 and 5, positive tainting technique is used in study 3, negative tainting technique is used in both studies 6 and 7, and Signature based technique is used in study 4. We integrate all these techniques to provide more than one level of defense in order to build a high secured system. This system not only detects known attack, but it has the ability to detect unknown attack by using learned based technique -Anomaly Detection and the ability to reduce false positives and false negative alarm.

Chapter 3 - Research Methodology

3.1 Introduction

This chapter explains the research methodology applied in this thesis. It describes the overall phases to build secure system.

3.2 The Research Methodology

This thesis involves the main phases and outputs for each phase as shown in Figure 3.1.

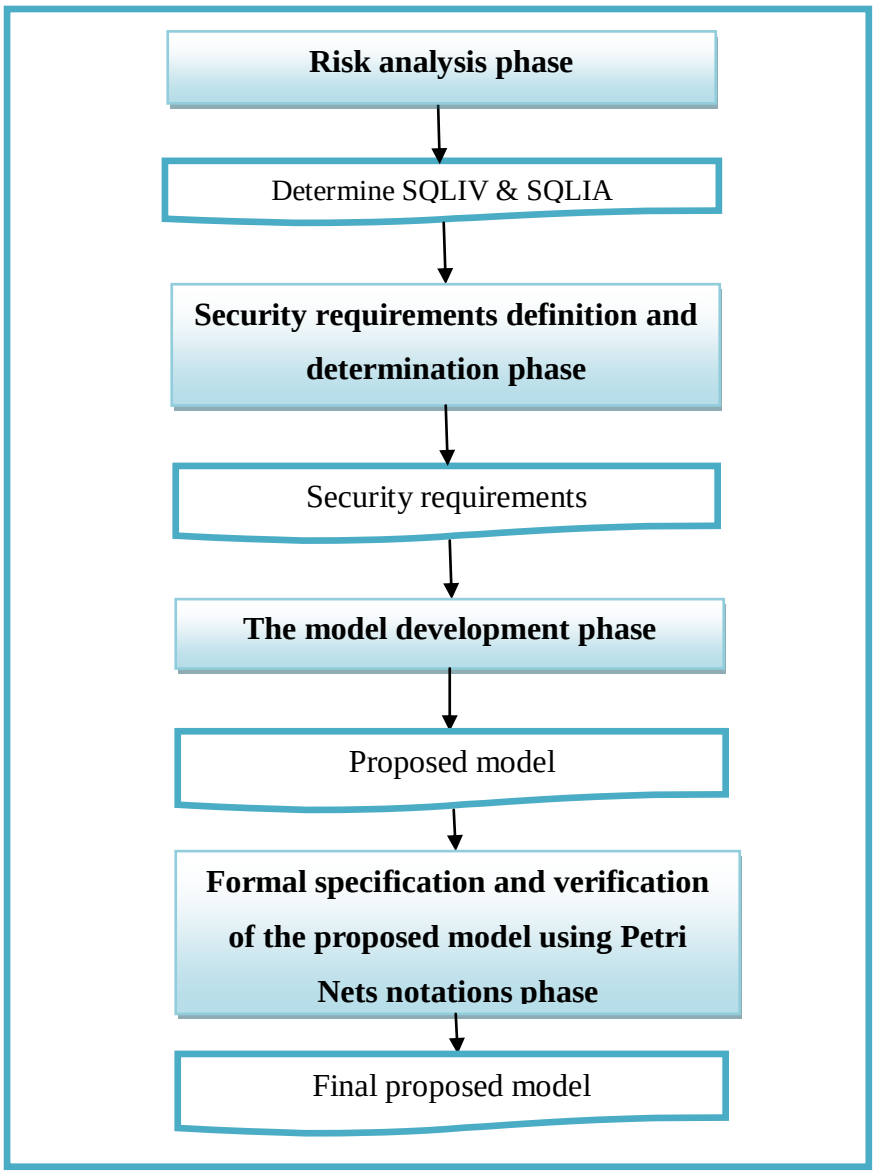

Figure 3.1: Main Phases for Research Methodology

3.2.1 Risk analysis phase

To determine security's requirements for building a secure system, firstly, we need to analyze the risk to identify vulnerabilities (SQLIVs). We use these SQLIVs to identify attacks (SQLIAs) in section 2.3; we present these SQLIAs. The outputs from this phase are SQLIAs.

3.2.2 Security requirements definition and determination phase

The first step to determine security's requirements is SQLIAs mentioned in the previous section. The next step, in order to identify the security services, we use these SQLIAs, which illustrated in section 2.4, the last step, in order to achieve security services; we use security mechanisms as illustrated in section 2.5. The outputs from this phase are security's requirements.

3.2.3 The model development phase

In this research, we propose a model by using many of the security mechanisms to achieve security requirements illustrated in previous section to build protected system. In chapter 4, we will present this phase in more details. The output from this phase is a propose model.

3.2.4 Formal specification and verification of the proposed model using Petri Nets notations phase

At this phase, we present formal specification and verification to propose a model, using Petri nets notations. In chapter 5, we will present this phase in more details. The output from this phase is a final propose model.

Chapter 4 – The Proposed Model

4.1 Introduction

This chapter explains the proposed model. It describes the overall phases of the model used in this study.

4.2 Risk analysis & Security requirements

Table 4.1 shows the abstract needed to build Entire model depending on risk analysis & critical review in section (2.3, 2.4).

Security requirements / Risk Analysis	S.R 1: Authorization	S.R 2: Authentication	S.R 3: Confidentially	S.R 4: Integrity
R.A 1: Tautology		*	*	
R.A 2: Illegal/Logical Incorrect queries			*	
R.A 3: Union queries		*	*	
R.A 4: Piggy-Backed query			*	*
R.A 5: Stored Procedure	*		*	*
R.A 6: Inference			*	
R.A 7: Alternate encoding			*	

Table 4.1 Risk Analysis & Security Requirements

4.3 Proposed Model

Depend on previous stage 4.2 we build our model by combining a number of defense techniques. We categorized them as based on literature in section 2.5 as the following:

- **By nature of defense** (Prevention, detection and deflection)
- **By detection principle** (Grammar based violation, Signature based, Tainted data flow [positive tainting, negative tainting] and anomaly detection [learning based]).
- **By analysis method** (hybrid analysis (static analysis, dynamic analysis))
- **By detection time** (coding time, testing time, operation time)
- **By detection location** (server-side application, server-side proxy)
- **By response**(report, user defined action)
- **By implementation**(no modification of code base)

Figure 4.1 shows the abstract levels of the model.

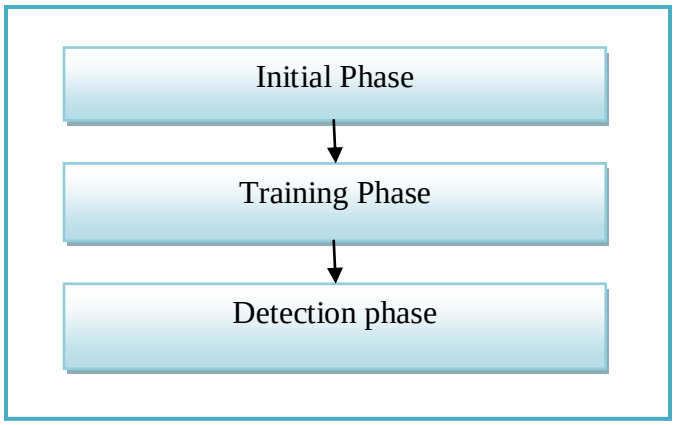

Figure 4.1 Main Phases of Proposed Model

Figure 4.2, Figure 4.3 & Figure 4.4 respectively, shows the abstract levels for all phases of the model.

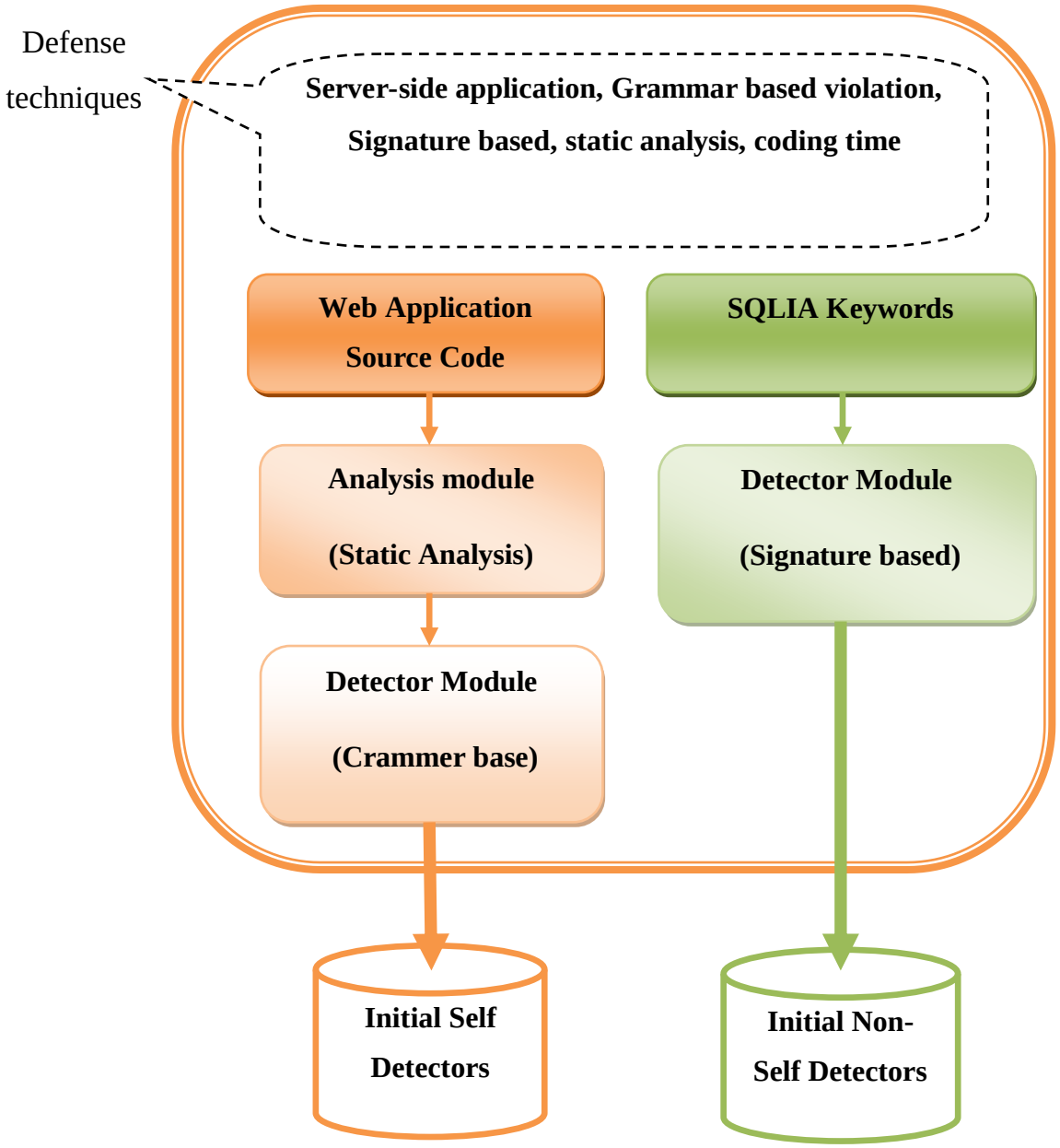

Figure 4.2 Abstract Level of Initial Phase

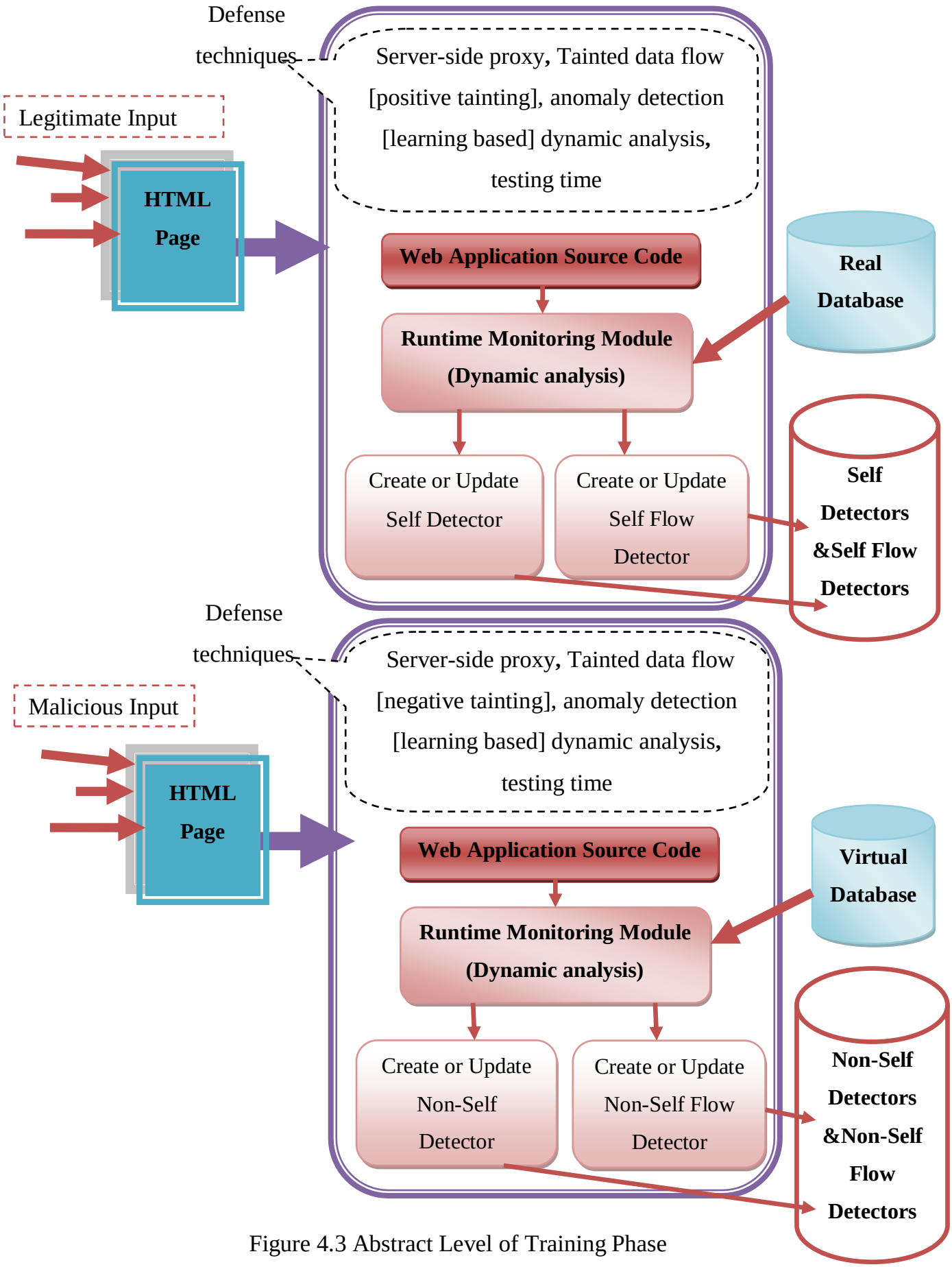

Figure 4.3 Abstract Level of Training Phase

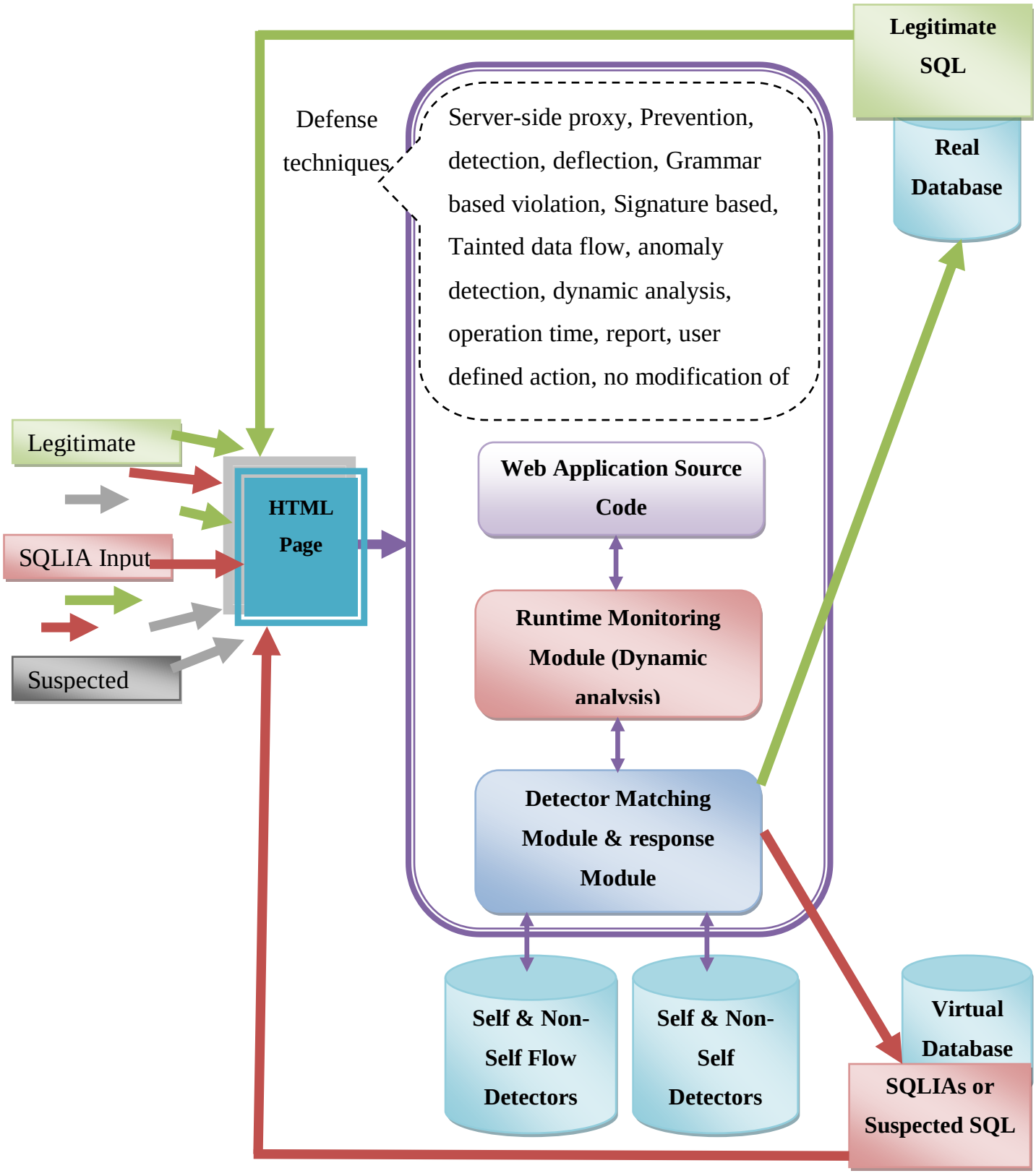

Figure 4.4 Abstract Level of Detection Phase

4.3.1 Initial phase (create initial values & initial detectors)

In this phase, we create initial detectors used to identify normal query from injected query. These detectors divided into two types self and non-self detectors; self-detector represent normal queries, and non-self detectors represent injected queries.

The initial detector is used as first line of defense, this detector generally is rule contain some elements that can be used to recognize query.

At this phase should also define the threshold value.

4.3.1.1 Convert SQL query to detector format

a. Determine the main token in SQL query, SEVEN keyword (SELECT, INSERT, UPDATE, DELETE, CREAT, DROP, ALTER)

b. Convert SQL query to tokens using space as a delimiter by token in query.

c. Calculate numbers of tokens in SQL query.

d. Create a fitness flag to represent the number of occurrences of this query, initial values are zero.

e. Put this SQL query in detector format; see Figure 4.5.

Figure 4.5 Detector Format

The main steps of this phase illustrated in Figure 4.6.

4.3.1.2 Create initial self detectors

In this sub-phase, we create self-detectors as the following:

1. Extract all possible SQL queries in web application source code using static analysis technique.

2. Convert all SQL query to detector Format.

3. Store initial self detector in disk.

The main steps of this phase illustrated in Figure 4.7.

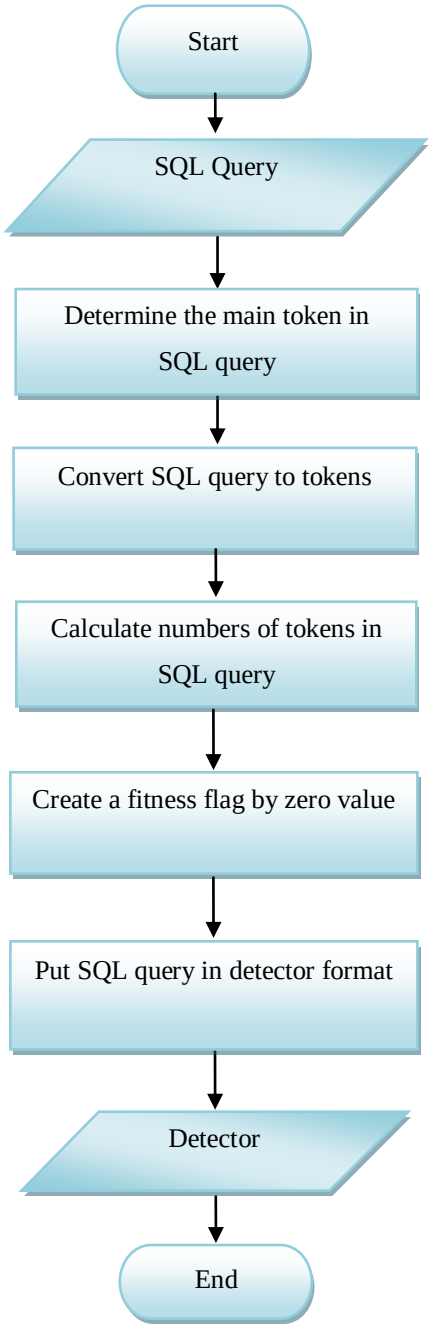

Figure 4.6 Convert SQL query to detector format

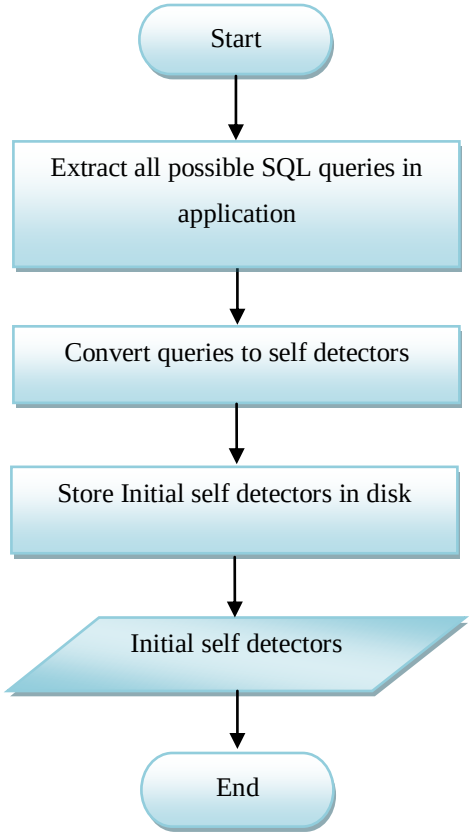

Figure 4.7 Create Initial Self Detectors

4.3.1.3 Create initial non-self detectors

In this phase, we adopt the sensitive characters/keywords of the SQLIAs and convert them to detector format as initial non-self detectors. According to [25] the sensitive characters/keywords of the SQLIA include: "exec", "xp_", "sp_", "declare", "Union ","+","//",".."," ;","'","-- ","%"," 0x ", which are not to be bound to use in the general structure query statement. After determine the sensitive keywords of the SQLIA, we represent these keywords as non-self detectors.

The main steps of this phase illustrated in Figure 4.8.

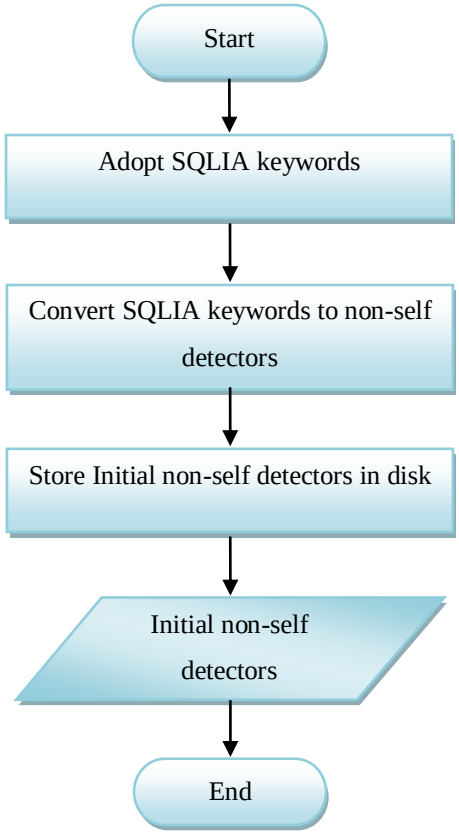

Figure 4.8 Create Initial Non-self Detectors

4.3.2 Training Phase (Update detectors and Create flow detectors)

We used normal and injected query at runtime application and compare runtime query with initial detectors to update them and create flow detectors.

This phase we spited to two sub-phases as follows:

4.3.2.1 Update self detectors and create self flow detectors

After create initial self-detectors, we used them to identify normal query from injected query. We monitor the web application in runtime at training time. At this time, we request normal query to application and compared with initial self detectors, if a detector is matched, then increase fitness of this detector and store flow of this query as self-flow detectors. If no matches found, then add this query to self-detectors to reduce false-positive alarms in detection phase; see Figure 4.9.

4.3.2.2 Update non-self detectors and create non-self flow detectors

After create initial non-self detectors we used them to identify normal query from injected query. We monitor the web application in runtime at training time. In this time we request injected query to application and compared with non-self detectors, if detector is match then increase fitness of this detector and store flow of this query as non-self flow detectors. If no matches found then add this query to non-self detectors to reduce false negative in detection phase; see Figure 4.10

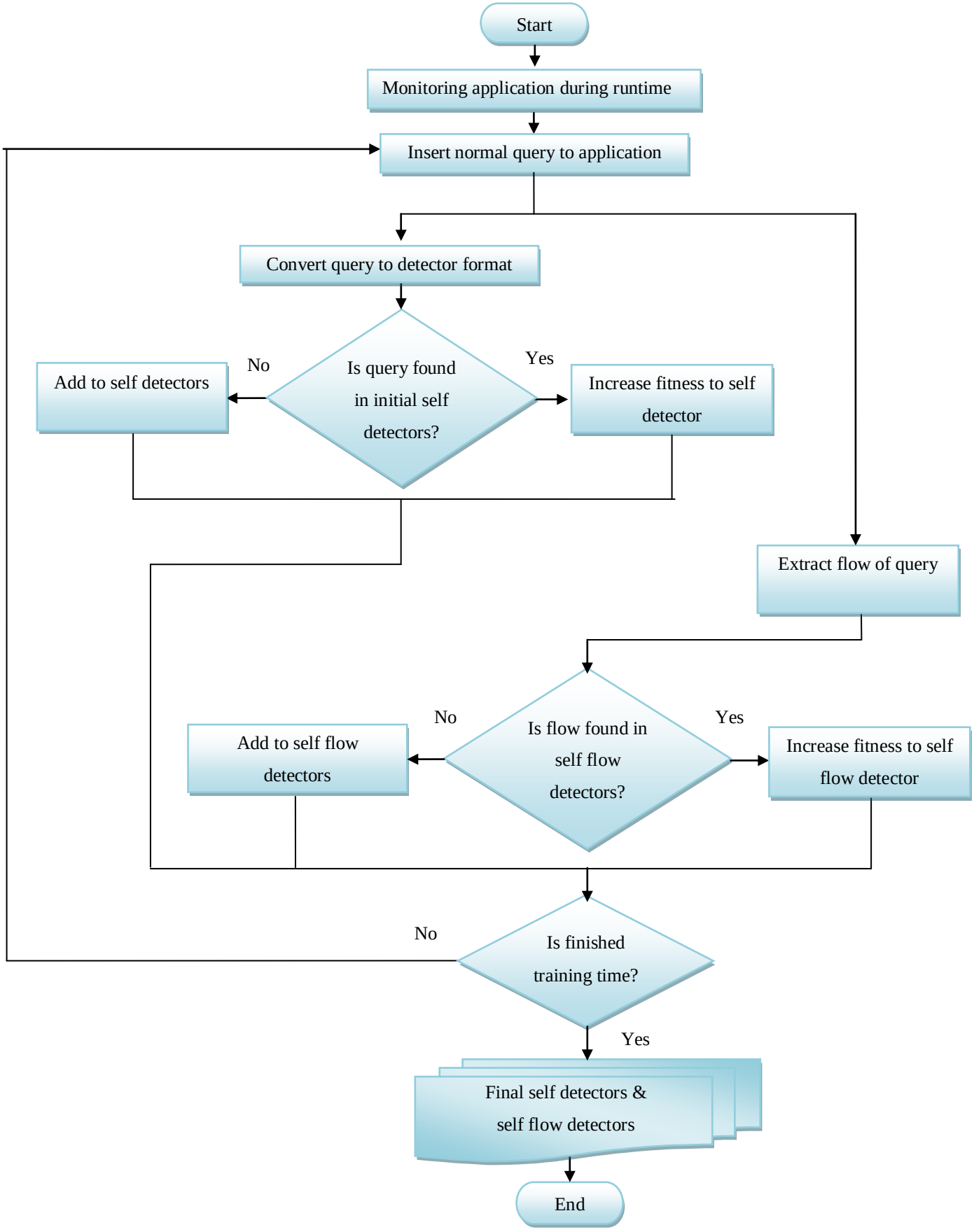

Figure 4.9 Update Self Detectors & Create Self Flow Detectors

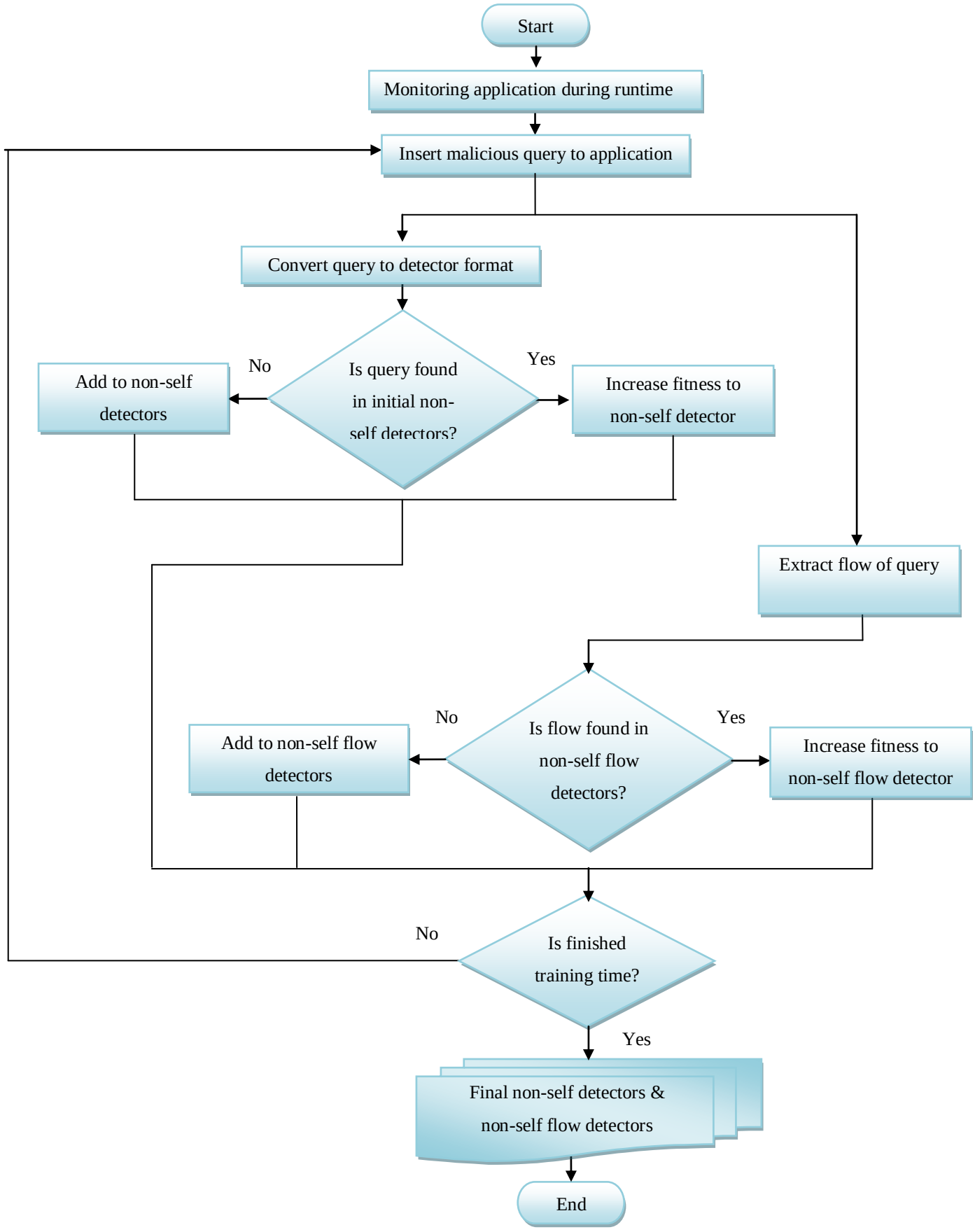

Figure 4.10 Update Non-self Detectors & Create Non-self flow Detectors

4.3.3 Detection phase

After create detectors and flow detectors we run the model in the reality in detection phase; see Figure 4.11

This phase we can summarized as follow:

Start Step(S):

1) Monitoring application during runtime.

2) Execute query.

3) Convert query to detector format.

4) Extract flow of query.

5) Compare query with self detectors

» **If found**: marked detector & flow detector as self then go to step (A).

» **If not found** : Compare query with non-self detectors

✓**If found**: marked detector & flow detector as non-self then go to step (A).

✓**If not found**: marked detector & flow detector as suspected then go to step (B).

Step A:

Is this detector, holds self mark?

» **If yes**: compare flow of query with self flow detectors

- **If found**: increase the fitness of self detector and self flow detector and execute the query in real database then go to step (S).
- **If not found** : compare flow of query with non-self flow detectors

- **If found**: go to step (C).

- **If not found**: Compare fitness of self detector with threshold value

  - If (fitness >= threshold) execute the query in real database then go to step (S).

    - If (fitness < threshold) then go to step (D).

» **If No**: compare flow of query with non-self flow detectors

- **If found**: increase the fitness of non-self detector and non-self flow detector and execute the query in virtual database then go to step (S).

- **If not found** : compare flow of query with self flow detectors

  - **If found**: go to step (C).

  - **If not found**: Compare fitness of non-self detector with threshold value

    - If (fitness >= threshold) execute the query in virtual database then go to step (S).

    - If (fitness < threshold) then go to step (D).

Step C:

Is this detector, holds self mark?

» **If yes**: Compare fitness of self detector with threshold value

- **If (fitness >= threshold)**: Compare fitness of non-self flow detector with threshold value

  - **If (fitness < threshold)** execute the query in real database then go to step (S).

  - **If (fitness >= threshold)** execute the query in virtual database then go to step (S).

- **If (fitness < threshold)**: execute the query in virtual database then go to step (S).

» **If No**: Compare fitness of non-self detector with threshold value

- **If (fitness >= threshold)**: Compare fitness of self flow detector with threshold value
  - **If (fitness < threshold)** execute the query in virtual database then go to step (S).
  - **If (fitness >= threshold)** execute the query in real database then go to step (S).
- **If (fitness < threshold)**: execute the query in real database then go to step (S).

Step B:

Compare flow of query with self flow detectors

» **If found**: Compare fitness of self flow detector with threshold value

- **If (fitness >= threshold)** execute the query in real database then go to step (S).
- **If (fitness < threshold)** go to step (D).

» **If not found**: Compare flow of query with non-self flow detector.

- **If found**: Compare fitness of non-self flow detector with threshold value
  - **If (fitness >= threshold)** execute the query in virtual database then go to step (S).
  - **If (fitness < threshold)** go to step (D).
- **If not found**: go to step (D).

Step D:

Send a message to system administrator and wait for the response until the end of waiting time.

- **If admin response**: add this query as selected by admin and execute it then go to step (S).
- **If admin not response**: execute the query in virtual database then go to step (S).

Table 4.2 shows the probabilities of query classification in detection phase.

Detector / Flow Detector		Self		Non-self		Suspected
		F>=T	F<T	F>=T	F<T	
Self	F>=T	Self	Self	Self	Self	Self
	F<T	Self	Self	Non-Self	Self	*Admin Selected/ Non-Self
Non-self	F>=T	Non-Self	Non-Self	Non-Self	Non-Self	Non-Self
	F<T	Self	Non-Self	Non-Self	Non-Self	*Admin Selected/ Non-Self
Suspected		Self	*Admin Selected/ Non-Self	Non-Self	*Admin Selected/ Non-Self	*Admin Selected/ Non-Self

Table 4.2 Query Classification Probabilities in Detection Phase

(F) Fitness, (T) Threshold

(*)Suspected Query (As selected by admin Or Non-Self if not selected)

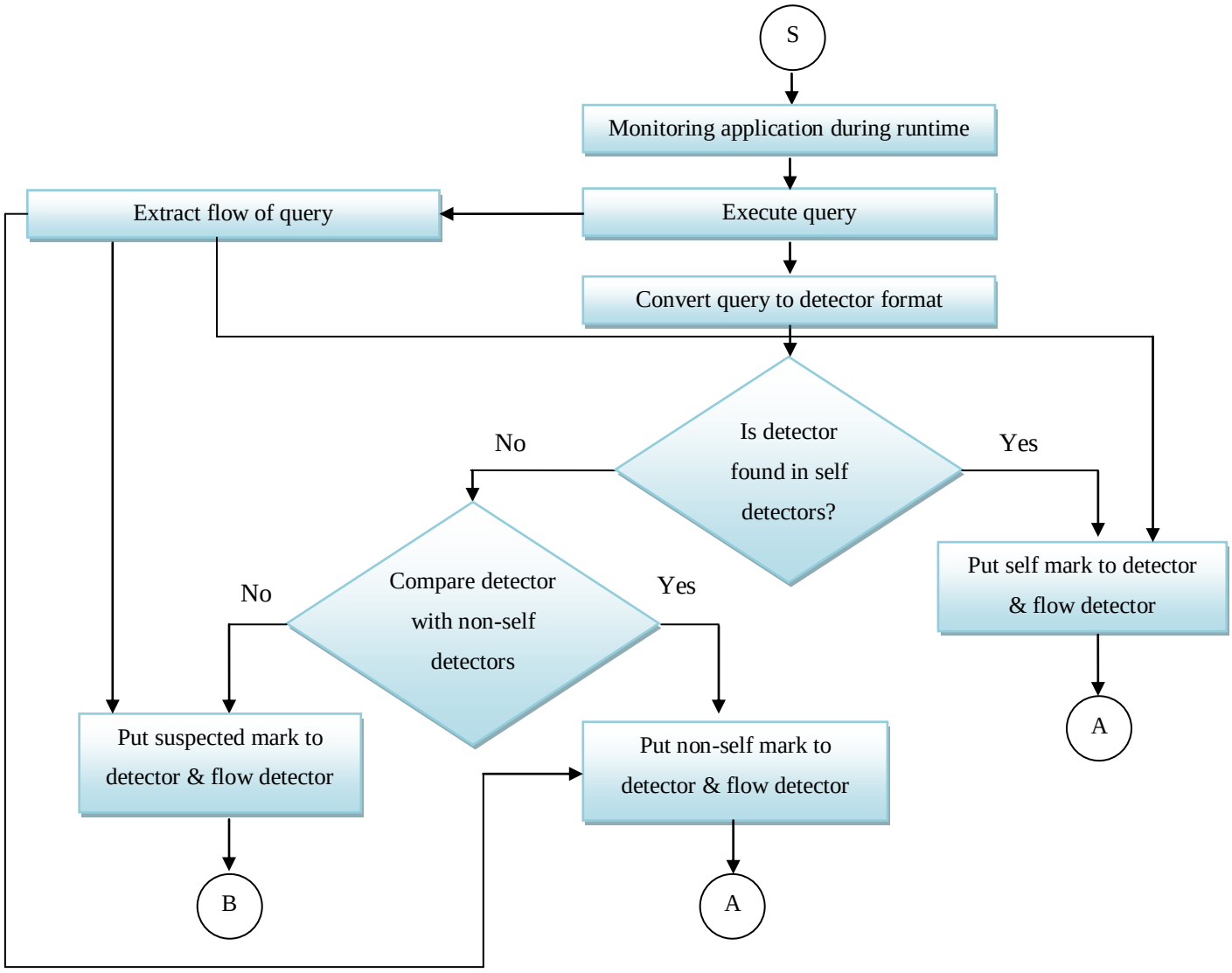

Figure 4.11 Start Step(S) in Detection Phase

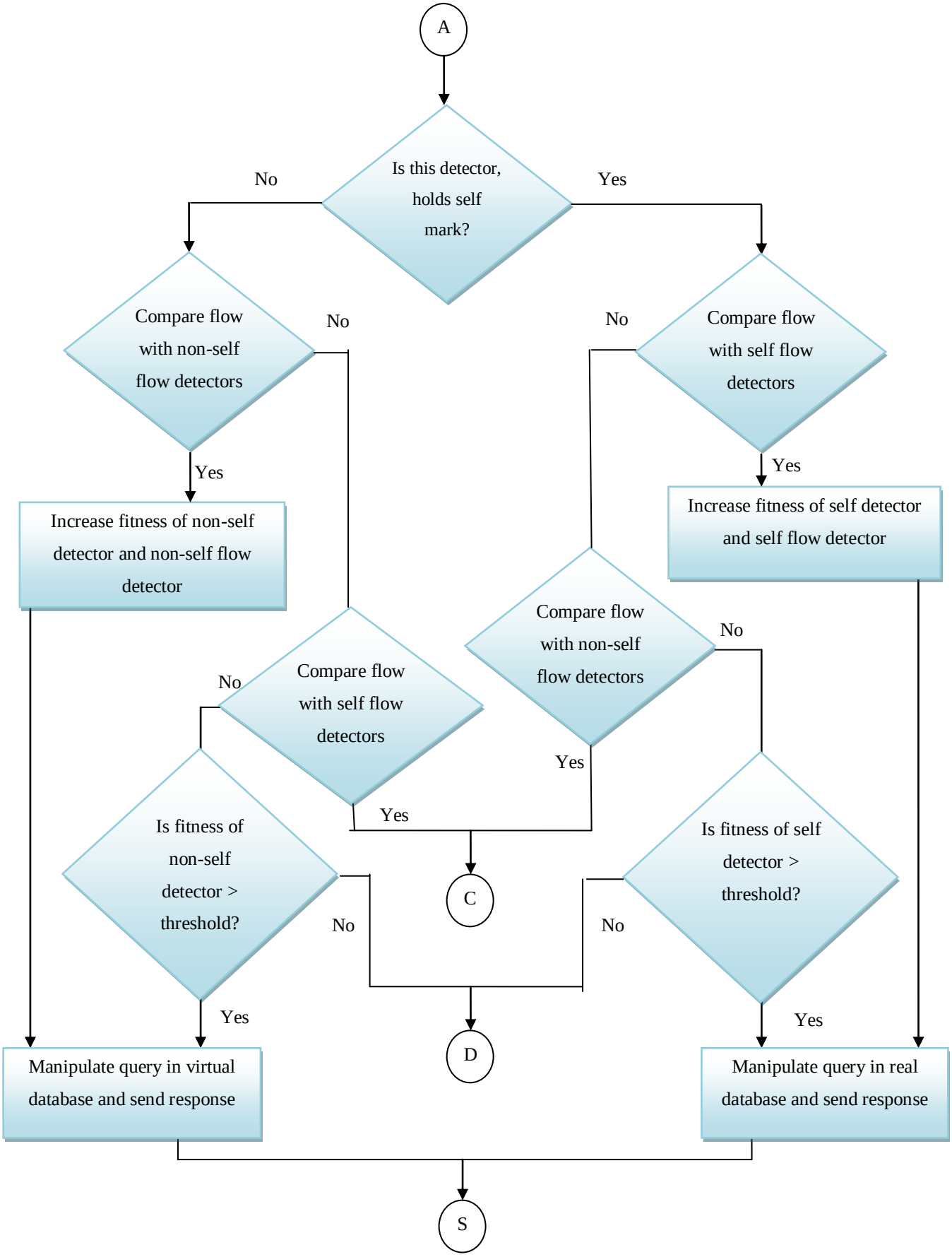

Figure 4.12 Step A in Detection Phase

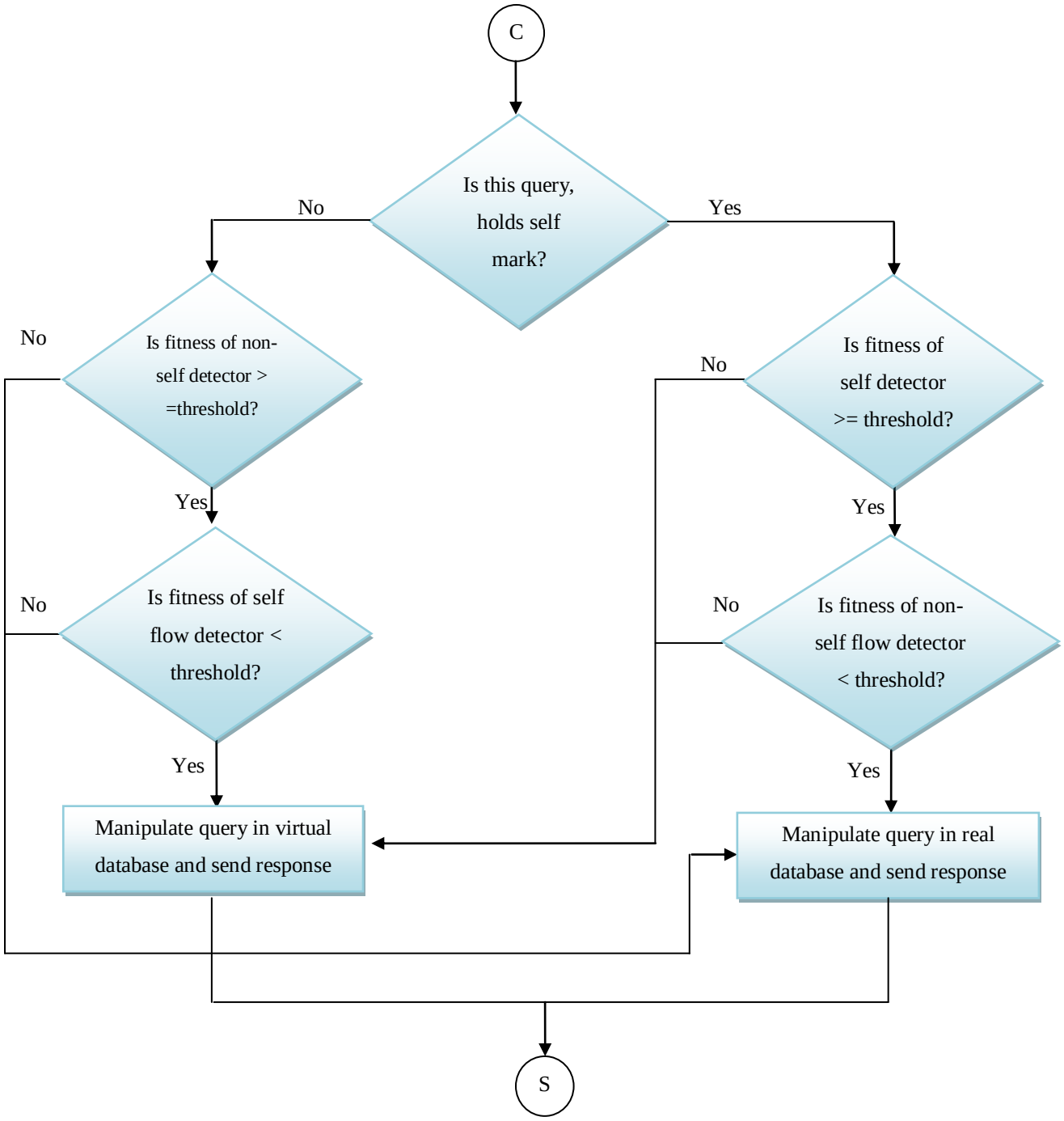

Figure 4.13 Step C in Detection Phase

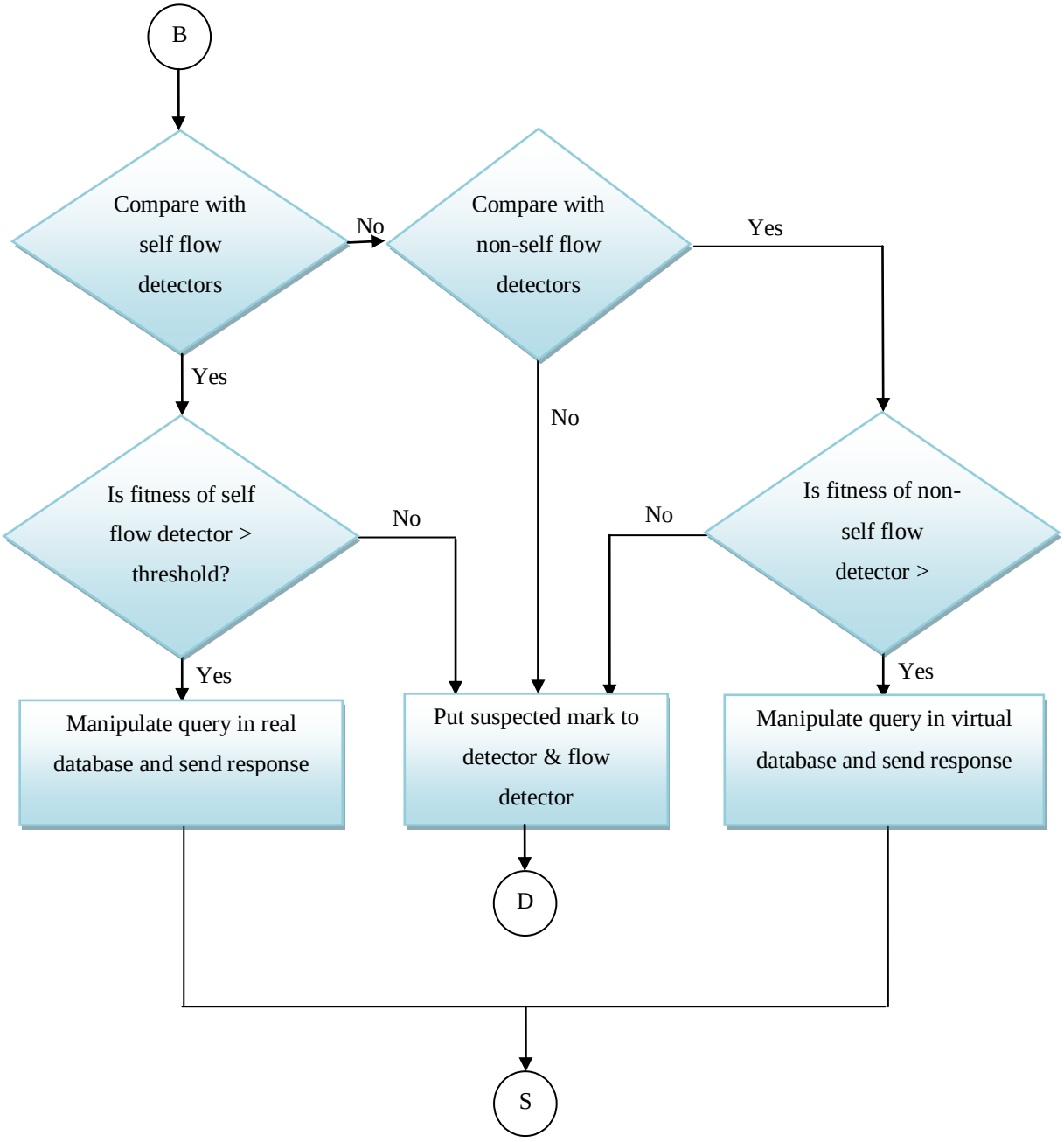

Figure 4.14 Step B in Detection Phase

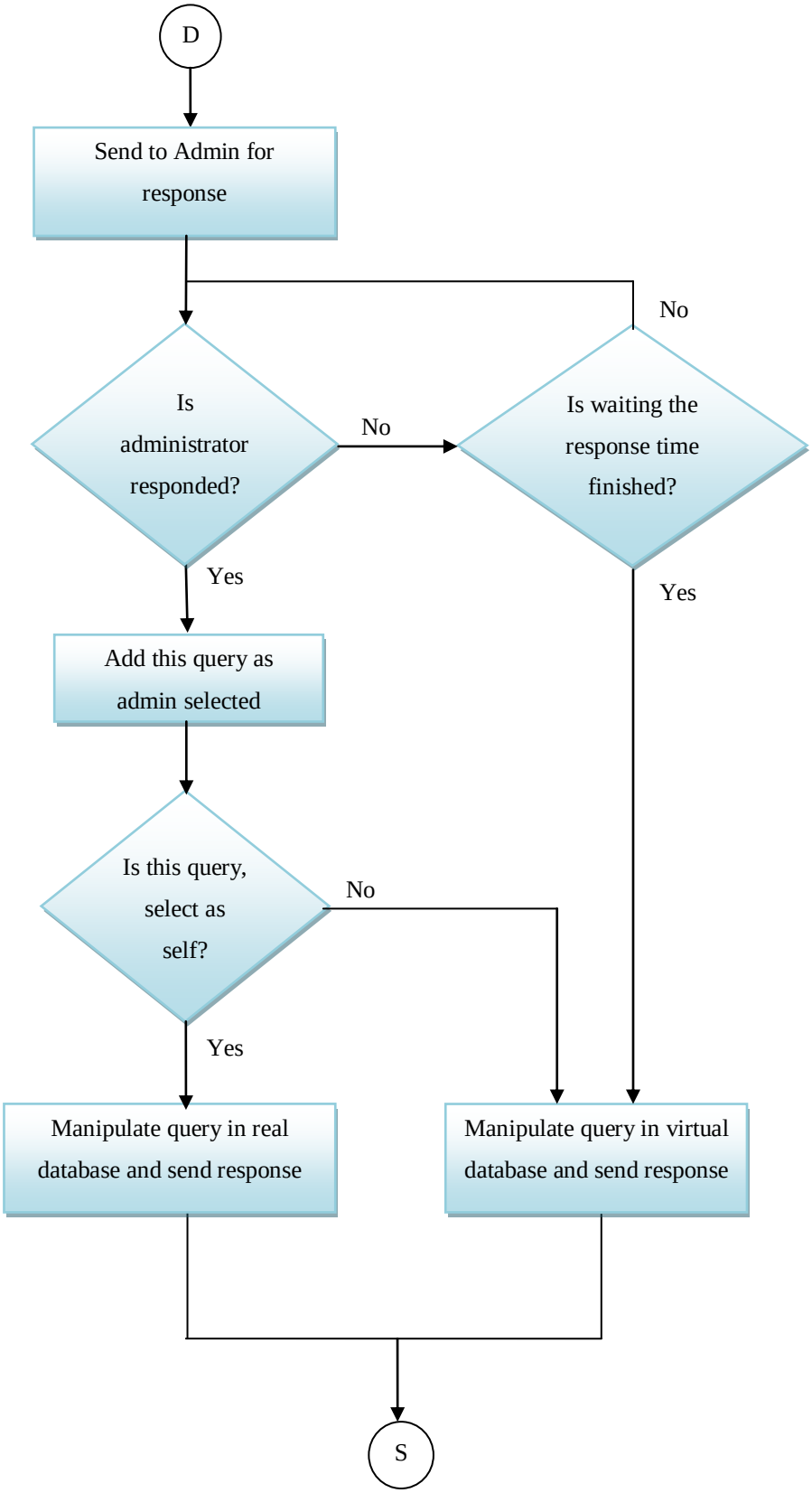

Figure 4.13 Step D in Detection Phase

Chapter 5 - Formal Specification and Verification

5.1 Tina Tool

Tina (TIme petri Net Analyzer) is a toolbox for the editing and analyzing Petri nets and Time Petri nets. The toolbox includes an editor for graphical or textual description of Petri nets and Time Petri nets. TINA can perform construction of reachability graphs, perform structural and path analysis. TINA was developed by the OLC group at LAAS (Laboratory of Analysis and Architecture of Systems) which is a research unit in the CNRS (National Center for Scientific Research) at Toulouse Cedex, France. OLC is a group that carries on research activities in the design of communication software.

The TINA toolbox contains set of tools in our model we use the following:

» **nd (NetDraw):** Editor and GUI for Petri nets, Time Petri Nets and Automata. Handles graphically or textually described nets or automata. Includes drawing facilities for nets and automata and a stepper simulator for nets.

» **play: Stepper simulator:** Allows to simulate interactively and step by step net descriptions in all formats accepted by tina. Its capabilities are similar to those of the nd stepper except that it is faster and may also simulate Time Transition Systems [30].

5.2 Formal Specification Using Tina Tool

In our model we use **nd** in Tina Tool to draw all models by using Petri Nets notation shown from Figure 5.1 to Figure 5.10.

5.2.1 Initial Phase (Create initial Detectors)

5.2.1.1 Convert SQL Query To Detector

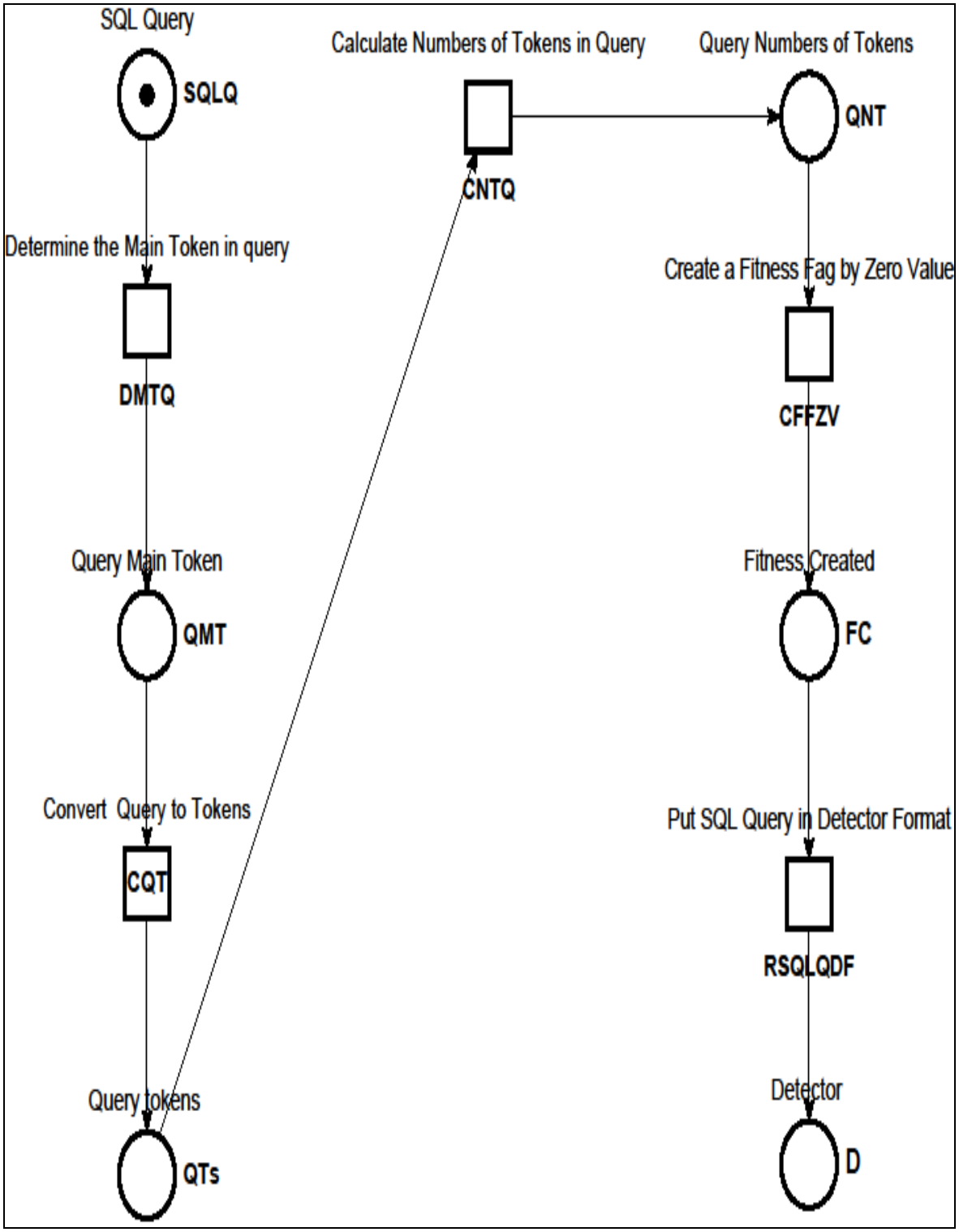

Figure 5.1 Convert SQL Query to Detector Using Petri Nets Notation

5.2.1.2 Create initial self detectors

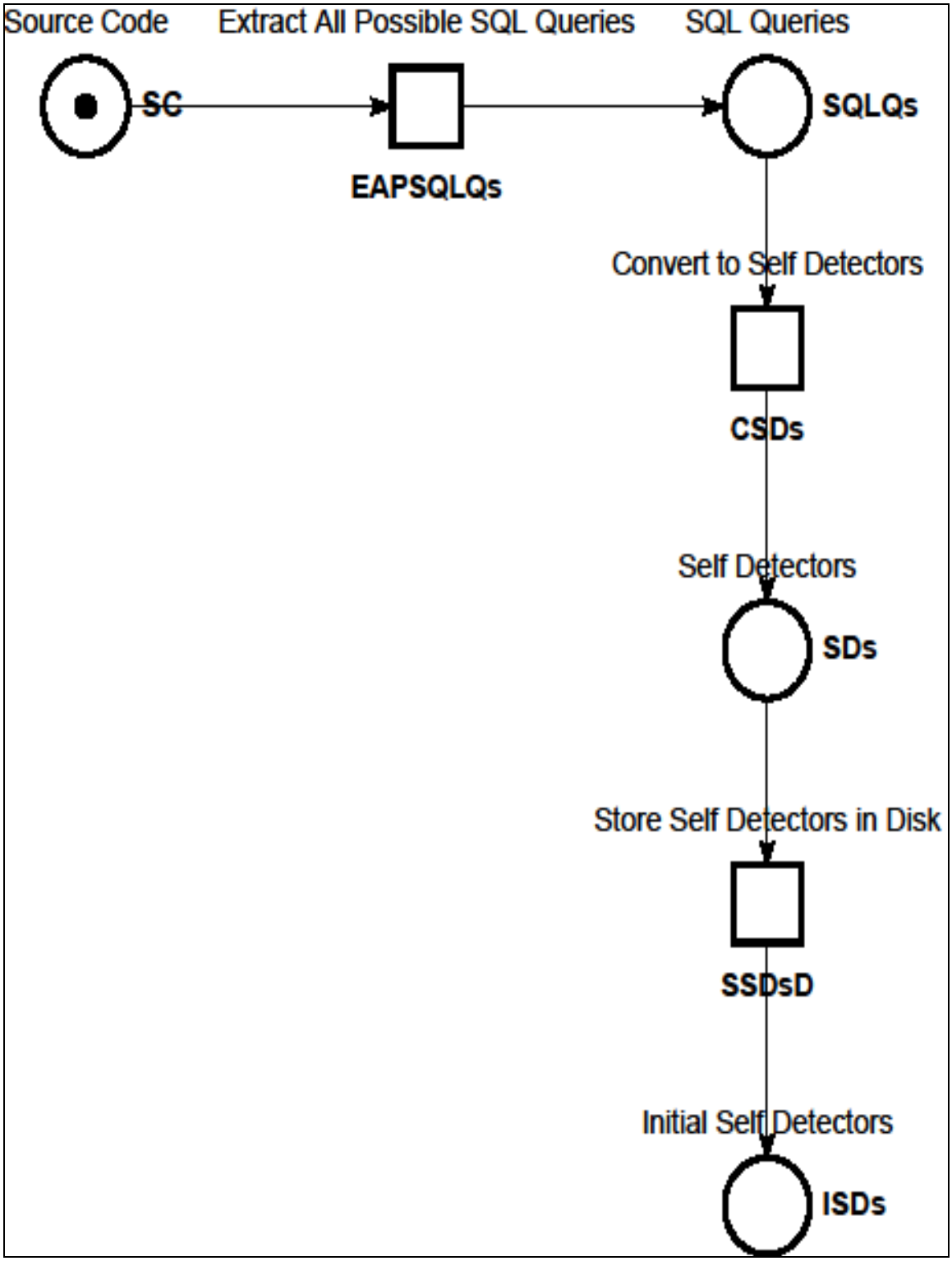

Figure 5.2 Create Initial Self Detectors Using Petri Nets Notation

5.2.1.3 Create initial non-self detectors

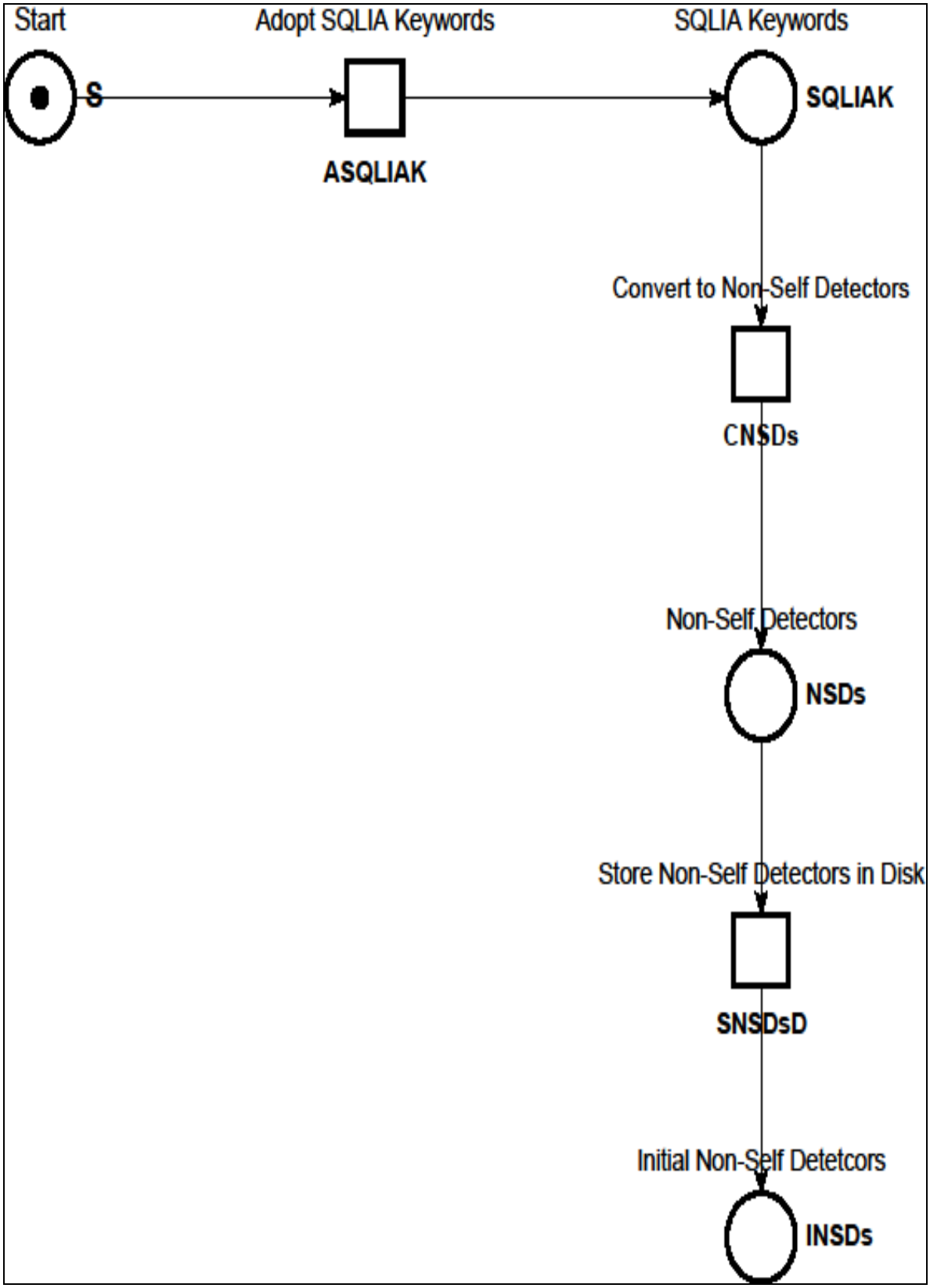

Figure 5.3 Create Initial Non-Self Detectors Using Petri Nets Notation

5.2.2 Training Phase (Update detectors and Create flow detectors)

5.2.2.1 Update self detectors and create self flow detectors

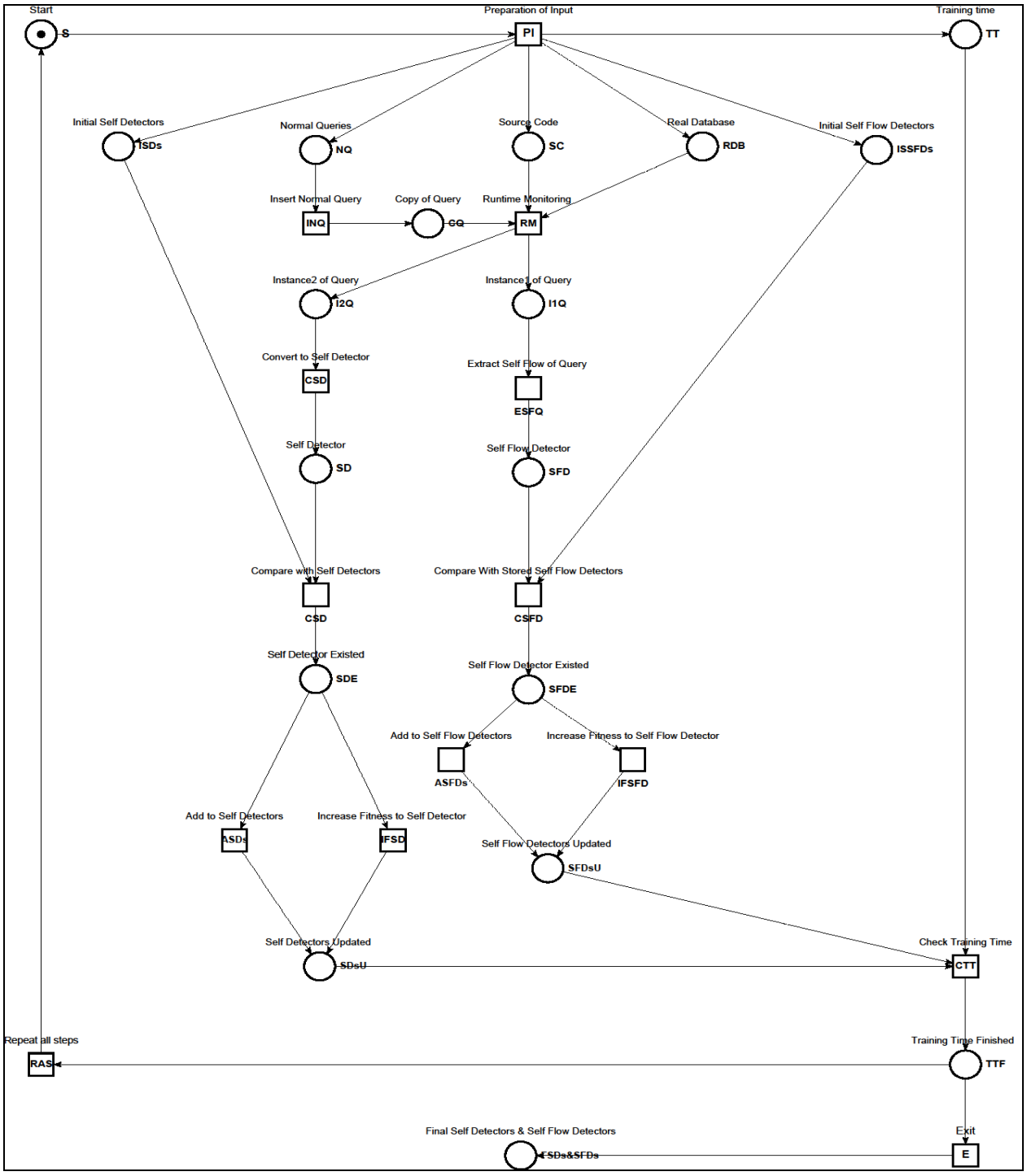

Figure 5.4 Update Self Detectors & Create Self Flow Detectors Using Petri Nets Notation

5.2.2.2 Update non-self detectors and create flow non-self detectors

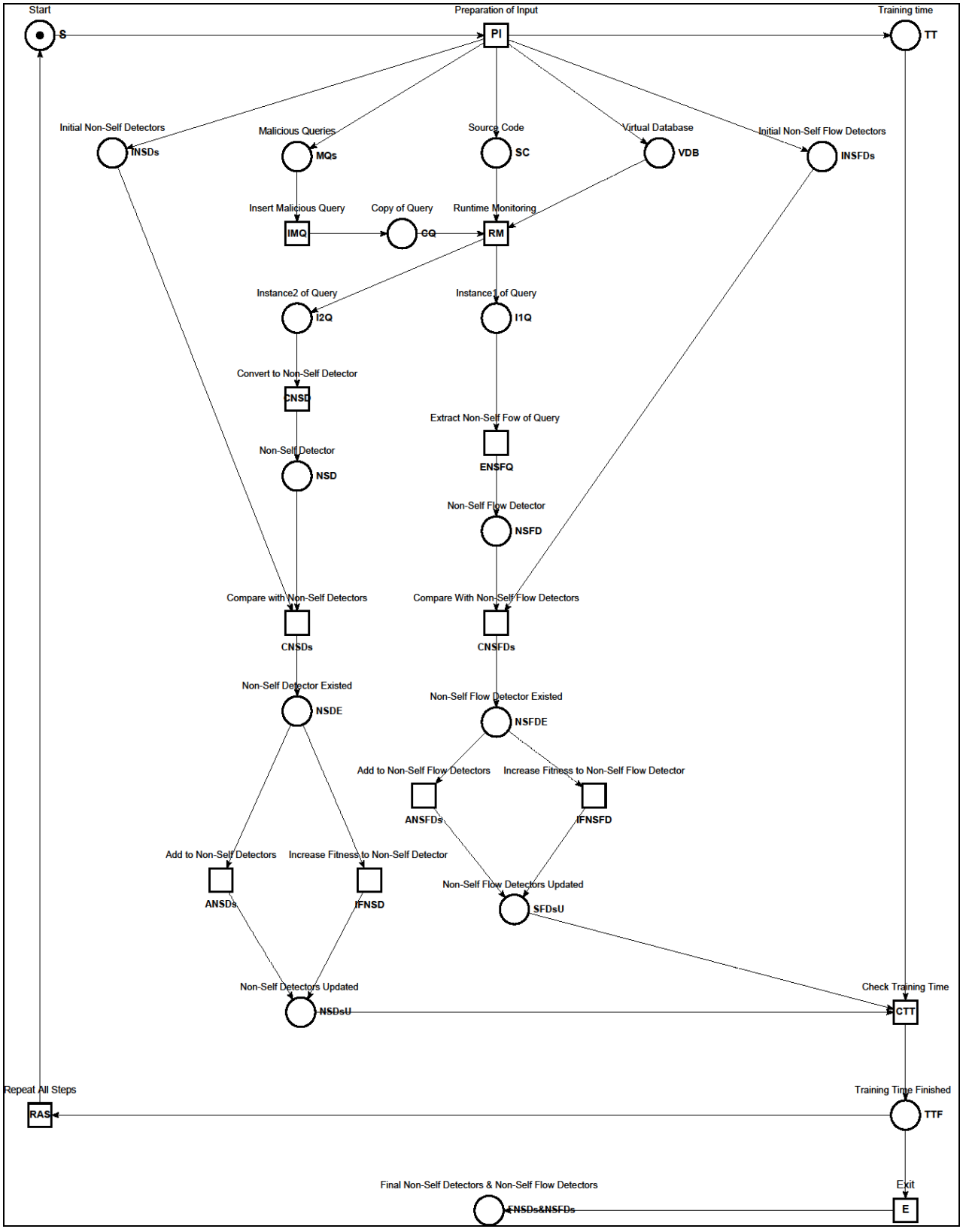

Figure 5.5 Update Non-Self Detectors & Create Non-Self Flow Detectors Using Petri Nets Notation

5.2.3 Detection phase

5.2.3.1 Start Step (S) in detection phase

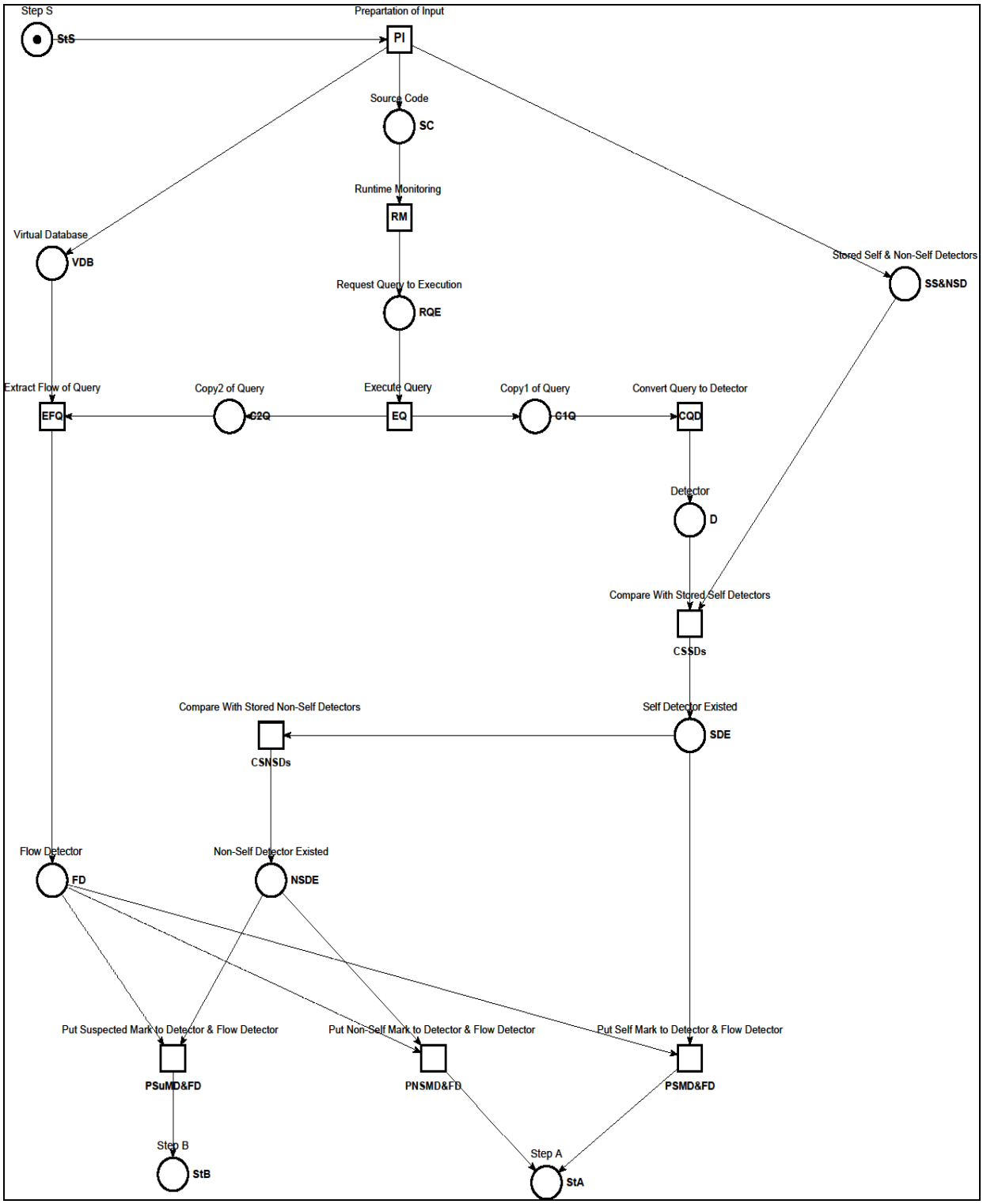

Figure 5.6 Start Step(S) in Detection Phase Using Petri Nets Notation

5.2.3.2 Step A in detection phase

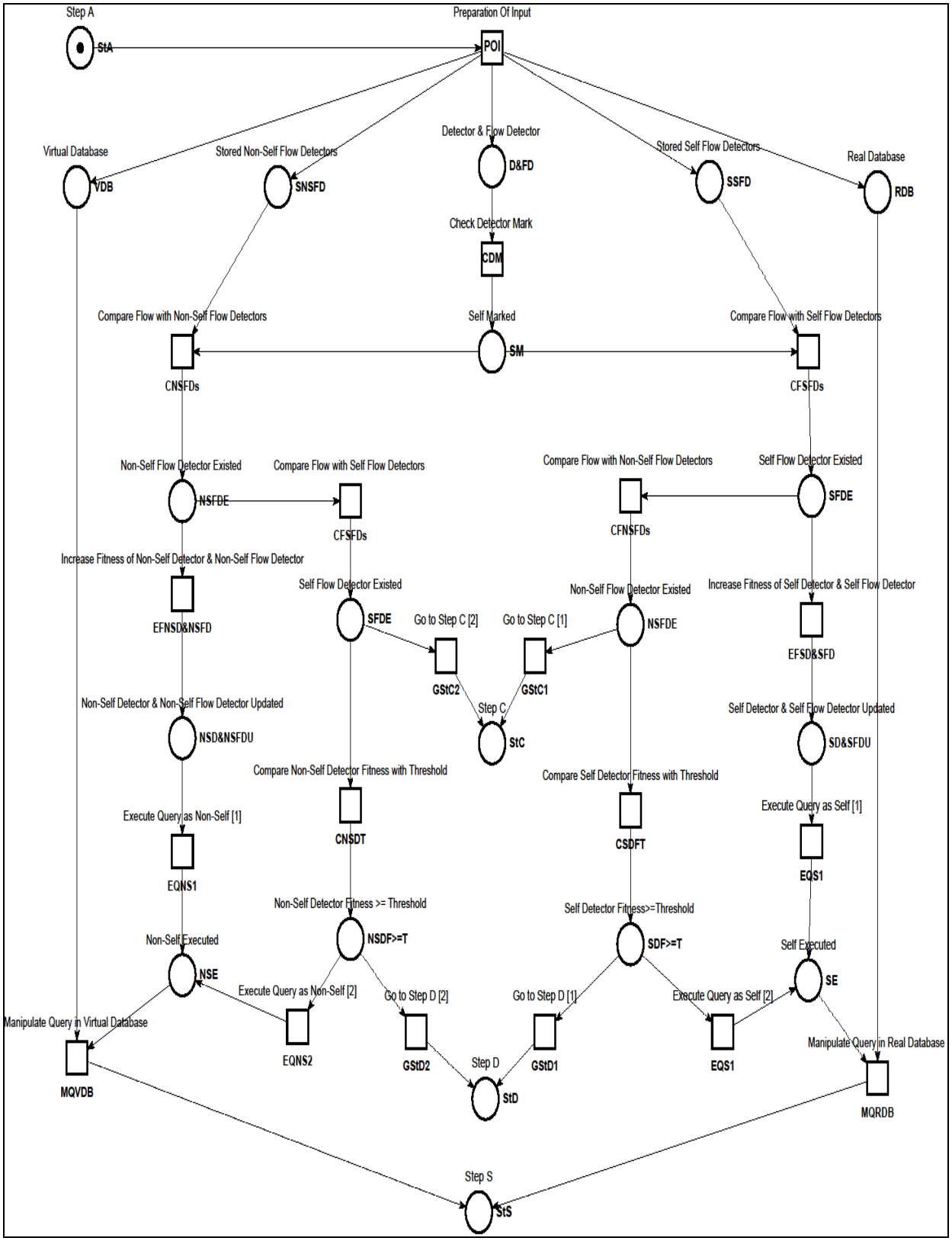

Figure 5.7 Step A in Detection Phase Using Petri Nets Notation

5.2.3.3 Step B in detection phase

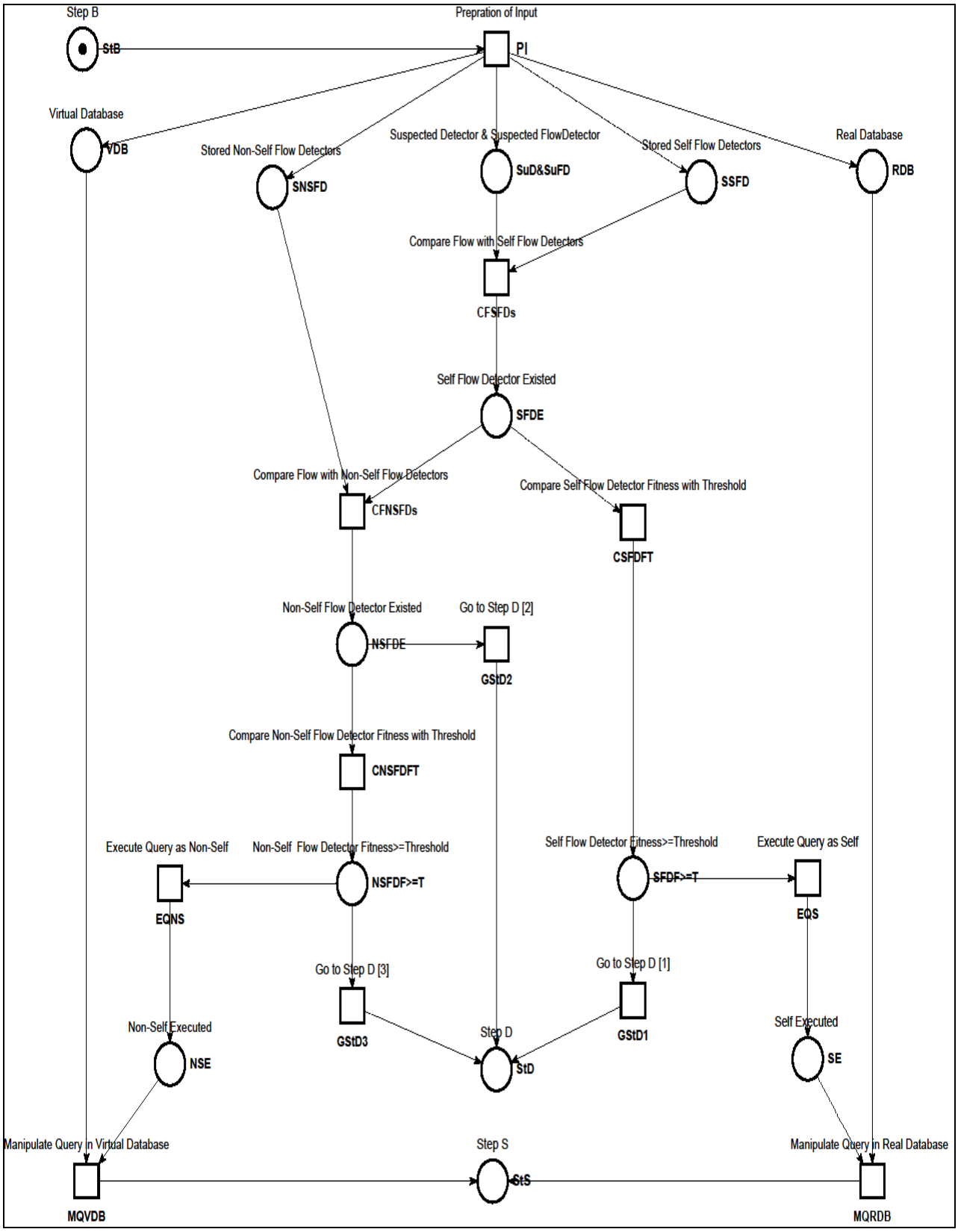

Figure 5.8 Step B in Detection Phase Using Petri Nets Notation

5.2.3.4 Step C in detection phase

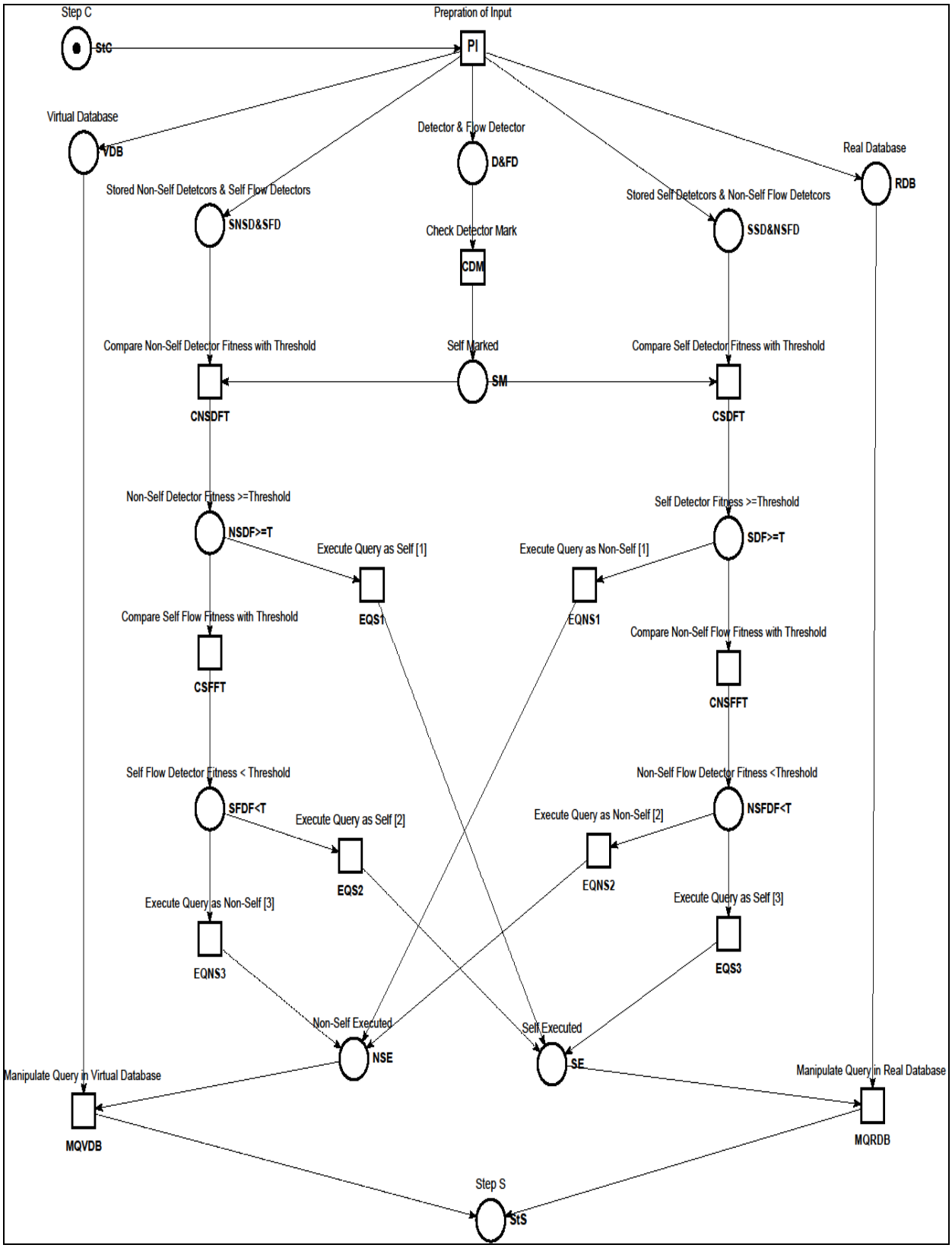

Figure 5.9 Step C in Detection Phase Using Petri Nets Notation

5.2.3.5 Step D in detection phase

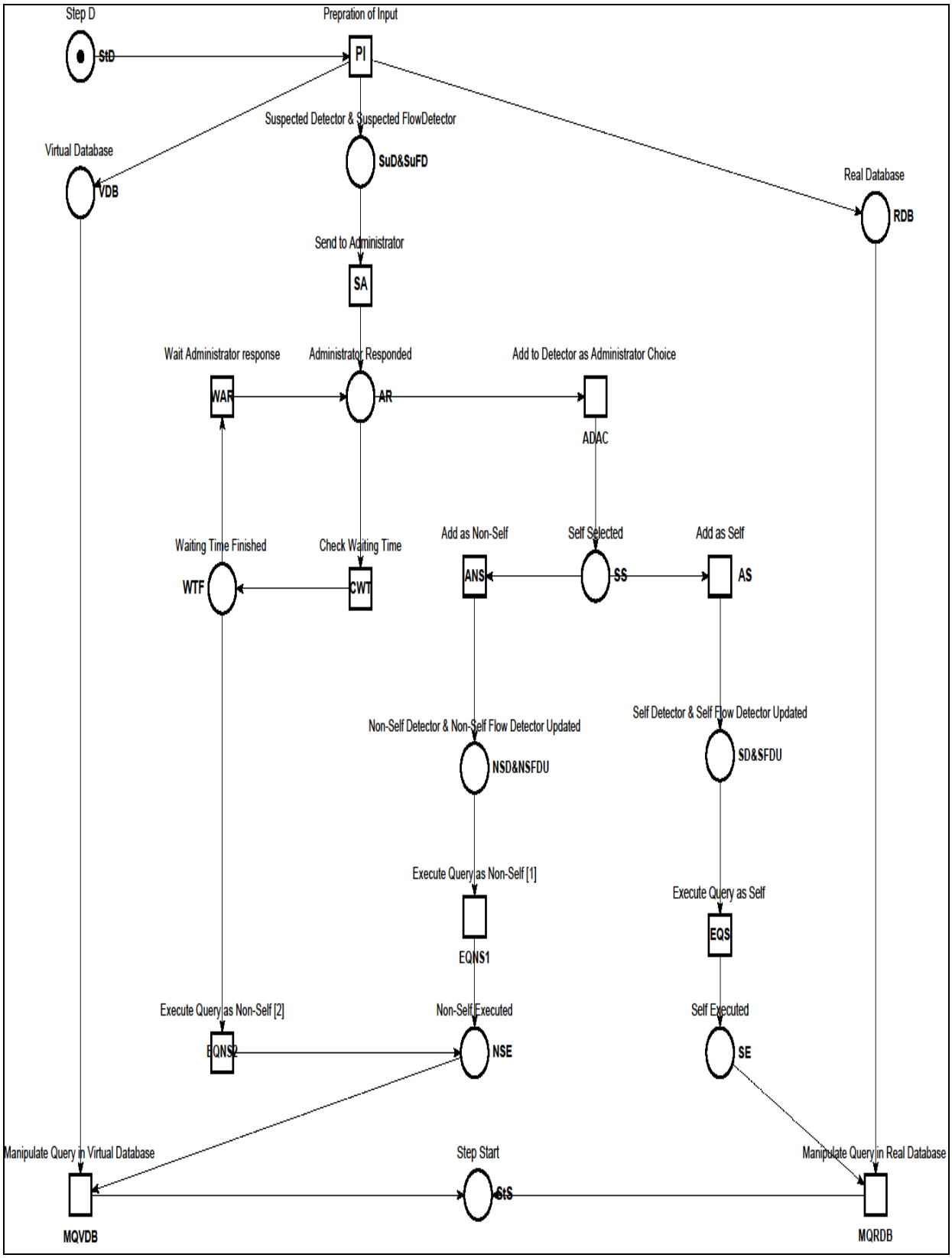

Figure 5.10 Step D in Detection Phase Using Petri Nets Notation

5.3 Verification Using Tina Tool

In our model we use **Stepper simulator** for all parts of model to follow the firing of Petri Nets transitions (events execution) to check dynamic behavior of model.

For instance, suppose we have these sequence of messages (EAPSQLQs - Extract All Possible SQL Queries, CSDs - Convert to Self Detectors, SSDsD - Store Self Detectors in Disk) these transition of Create Initial Self Detectors shown in Figure 5.2, transition EAPSQLQs is only enabled transition at the beginning of execution because the input place SC is contains number of tokens equal to weight of the directed arc connected them; this transition should be fired by removes token form input place SC and deposits token to output place SQLQs; to make enable the CSDs transitions and so on. We show the flow of transition & safeness of the model in Figure 5.11 and Figure 5.12 respectively.

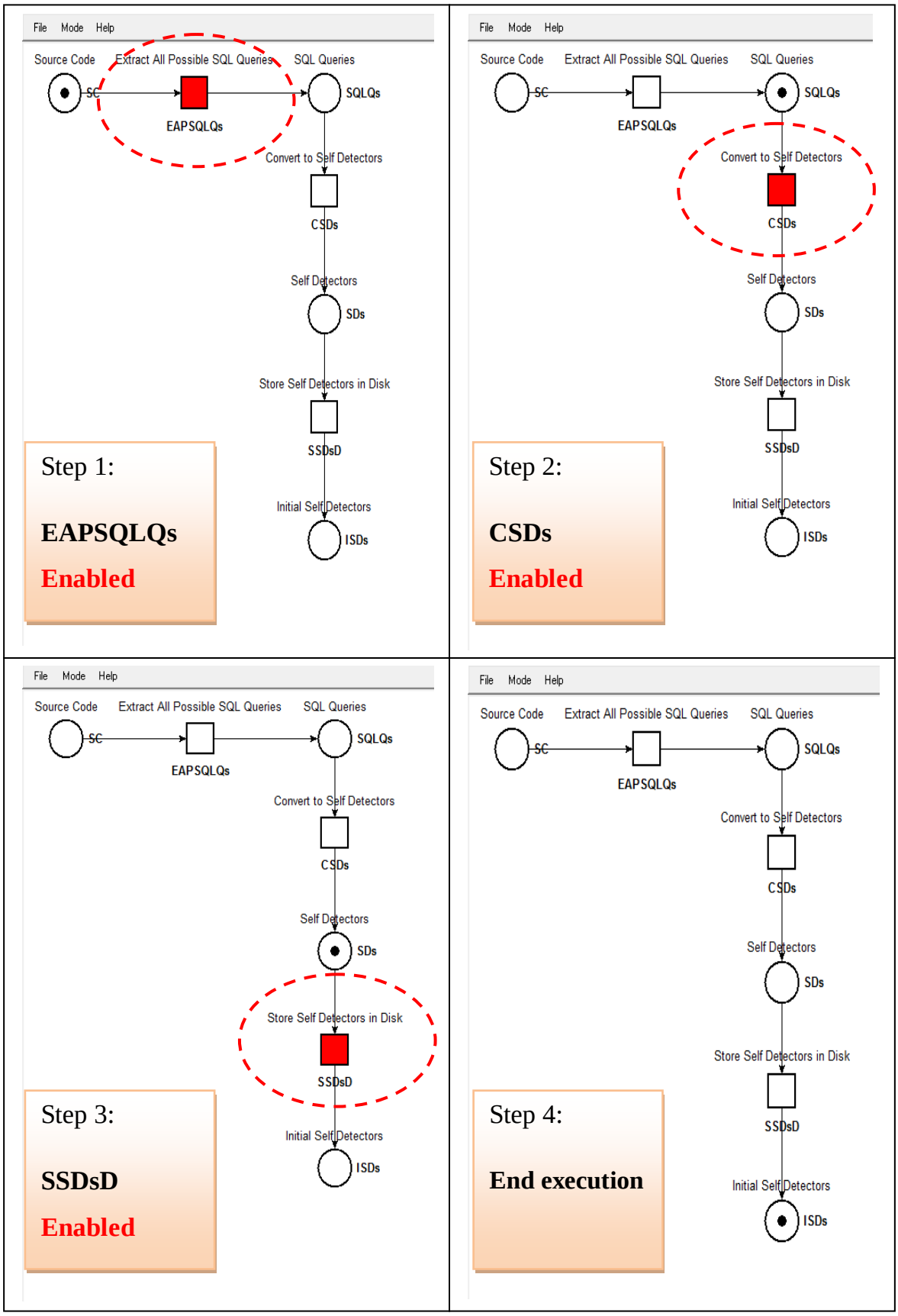

Figure 5.11 Flow of Transition Using Stepper Simulator Manually

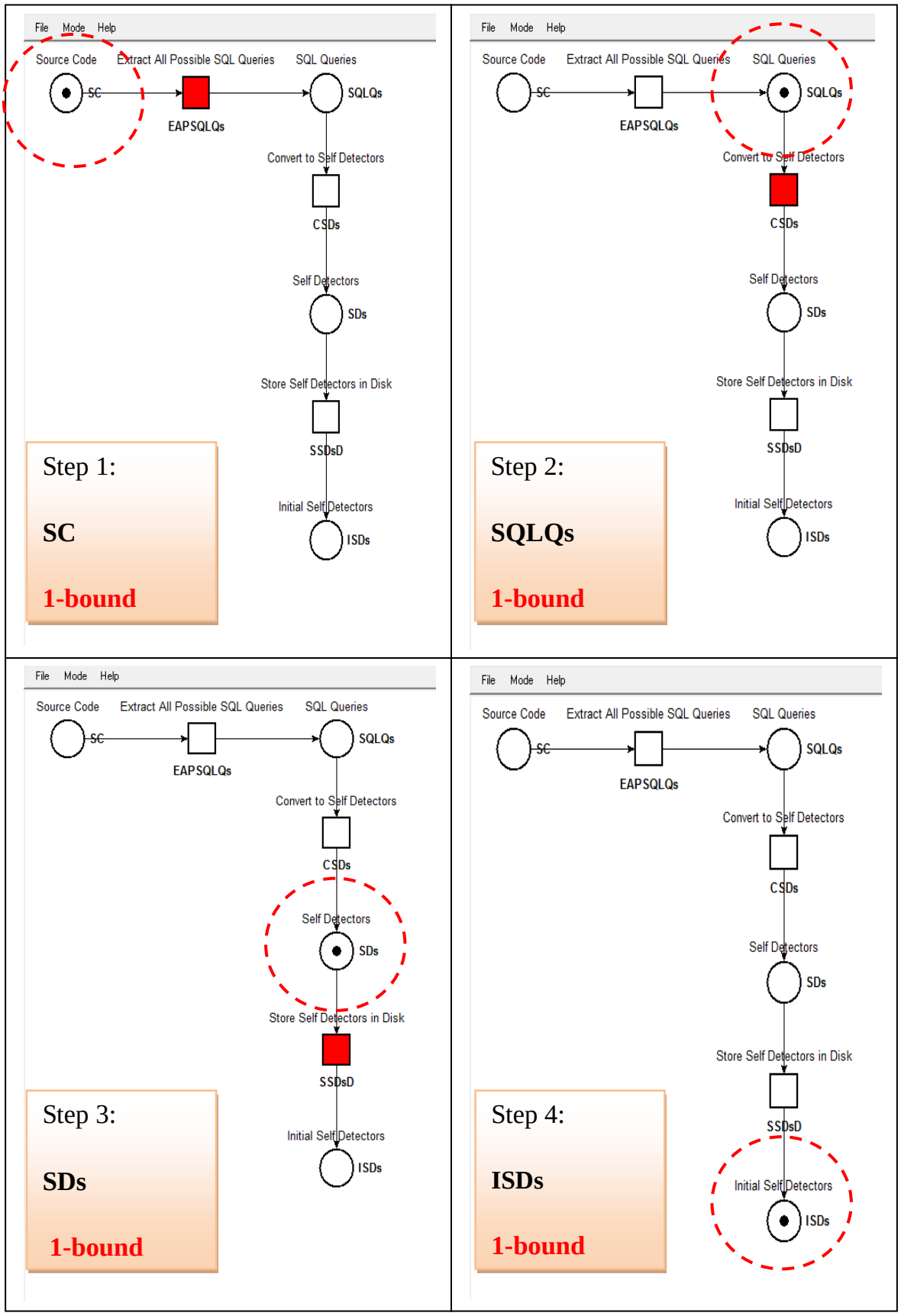

Figure 5.12 1-bound & Safe Model Using Stepper Simulator Manually

Chapter 6 - Conclusion and Future work

6.1 Conclusions

In this research we have presented an extensive review of the different types of SQLIA with descriptions and examples of how attacks of that type could be performed. We also provide and analyze existing detection and prevention models against SQLIA and we discuss its strengths and weaknesses and its differences with our model. The model has been presented in flowcharts form and subsequently we demonstrate its formal specifications using Petri net language and verify it.

As a consequence, the results showed the effectiveness of the model in terms of the correct syntax and safeness.

6.2 Future work

In the future studies we recommend the following:

» Apply formal verification to assess the readiness of the proposed model for execute in real word.

» Increase training time and comprehensive data set will reduce the proportion of positive and negative false alarms and also reduces the probability of suspicious query.

References

[1] M. e. a. Muntjir, "Security Issues and Their Techniques in DBMS-A Novel Survey," *International Journal of Computer Applications*, vol. 85, no. 13, pp. 39-43, Janeruary 2014.

[2] N. a. S. G. Mishra, "Defenses To Protect Against SQL Injection Attacks," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 2, no. 10, October 2013.

[3] W. a. H. H. H. Win, "A Simple and Efficient Framework for Detection of SQL Injection Attack," *IJCCER*, vol. 1, no. 2, pp. 26-30, July 2013.

[4] Jamal, Md. Shahid Sagar and Aafrin, "BASIS OF FORMAL SPECIFICATION IN DISCRETE MATHEMATICS," *VSRD international journal of computer science & information*, vol. 2, no. 11, November 2012.

[5] "Formal specification languages," [Online]. Available: http://www.liacs.nl/~mtbeek/re-formal.pdf. [Accessed 16 February 2014].

[6] V. Nithya, "A Survey on SQL Injection attacks, their Detection and Prevention Techniques," *IJECS*, vol. 2, no. 4, pp. 886-905 , April 2013.

[7] Pathan, Diallo Abdoulaye Kindy and Al-Sakib Khan, "A Detailed Survey on various aspects of SQLInjection in Web Applications: Vulnerabilities,Innovative Attacks and Remedies," *International Journal of Communication Networks and Information Security (IJCNIS)*, vol. 5, no. 2, pp. 80 - 92, August 2013.

[8] Manish Kumar et al, "Detection and Prevention of SQL Injection attack," *International Journal of Computer Science and Information Technologies (IJCSIT)*, vol. 5, no. 1, pp. 374-377, 2014.

[9] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, "A Classification of SQL Injection Attacks and Countermeasures," *Proceedings of the IEEE International Symposium on Secure Software Engineering,* vol. 1, pp. 13-15, 2006.

[10] Neha Mishra and Sunita Gond, "Defenses To Protect Against SQL Injection Attacks," *International Journal of Advanced Research in Computer and Communication Engineering,* vol. 2, no. 10, pp. 3829 - 3833, October 2013.

[11] Yash Tiwari ,Mallika Tiwari, "A Study of SQL of Injections Techniques and their Prevention Methods," *International Journal of Computer Applications,* vol. 114, no. 17, March 2015.

[12] MA Lawal, ABM Sultan, AO Shakiru, "Systematic Literature Review on SQL Injection Attack," *International Journal of Soft Computing* , vol. 11, no. 1, pp. 26-35, 2016.

[13] Anup Shakya and Dhiraj Aryal, "A Taxonomy of SQL Injection Defense Techniques," Doctoral dissertation, Master's Thesis Computer Science, School of Computing Blekinge Institute of Technology, Sweden, June 2011.

[14] M. Bishop, "Formal Specification," in *Computer Security: Art and Science*, 2003, p. 548.

[15] A. D. Boyd, "A Formal Definition of the Object-Oriented Paradigm for Requirements Analysis," AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH, 1991.

[16] I. Sommerville, "Formal Specification," in *Software Engineering 8th*, 2006, pp. 246-267.

[17] JiacunWang, "Petri Nets for Dynamic Event-Driven System Modeling," in *Handbook of Dynamic System Modeling*, Gainesville, U.S.A., Edited by Paul A.

Fishwick ,University of Florida, 2007.

[18] B. YILMAZ, "APPLICATIONS OF PETRI NETS," Doctoral dissertation, Izmir Institute of Technology, October 2008.

[19] J. e. Wang, "Petri Nets," in *Handbook of Finite State Based Models and Applications*, Boca Raton, FL, USA: CRC Press, Taylor and Francis Group, 2013, pp. 298-299.

[20] M. A. M. H. a. W. M. Blätke, "Tutorial Petri Nets in Systems Biology.," Otto von Guericke University and Magdeburg, Centre for Systems Biology, 2011.

[21] R. L. Paturca, "BIOMODELING WITH PETRI NETS," TURKU UNIVERSITY OF APPLIED SCIENCES ,BACHELOR´S THESIS, 13.01.2014.

[22] A. Halder, "A study of Petri nets modeling, analysis," Department of Aerospace Engineering Indian Institute of Technology Kharagpur, India, 2006..

[23] A. Bobbio, "SYSTEM MODELLING WITH PETRI NETS," *Systems reliability assessment. Springer Netherlands*, pp. 103-143, 1990.

[24] J. M. a. R. Kumar, "Blocking of SQL Injection Attacks by Comparing Static and Dynamic Queries," *International Journal of Computer Network and Information Security (IJCNIS),* vol. 5, no. 2, pp. 1-9, 2013.

[25] D. G. a. M. C. Kumar, "An Approach for SQL Injection Attack Prevention," *VESIT , International Technological Conference*, pp. 23-27, January 2014.

[26] Reshma Rai and Jitendra Jadhav , "IMPLEMENTATION OF SMART FILTER TO AVOID SQL INJECTIONS WITH SIGNATURE BASED INTRUSION DETECTION," *International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE),* vol. 2, no. 1, pp. 100-107, January 2013.

[27] J. O. A. a. A. J. Qaralleh, "A HYBRID TECHNIQUE FOR SQL INJECTION ATTACKS DETECTION AND PREVENTION," *International Journal of Database Management Systems (IJDMS)*, vol. 6, no. 1, pp. 21-28, February 2014.

[28] e. a. Pranita Talekar, "WEB APPLICATION PROTECTION AGAINST SQL INJECTION ATTACK.," *Bimonthly International Journal*, vol. 2, no. 05/06, pp. 096-100, 2014.

[29] e. a. Ammar Alazab, "Web application protection against SQL injection attack.," *Proceedings of the 7th International Conference on Information Technology and Applications*, pp. 1-7, 2011.

[30] "TIme petri Net Analyzer," OLC Group (Tools and Software for Communicating systems) Laboratory for Analysis and Architecture of Systems (LAAS)National Center for Scientific Research (CNRS), [Online]. Available: http://projects.laas.fr/tina//. [Accessed 15 10 2015].