uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Olfa Chemli

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Master of Computer Science

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Requirement Based Test Suite Generation Using Dependence Analysis

TITRE DE LA THÈSE / TITLE OF THESIS

Hasan Ural

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Daniel Amyot

Chung-Horng Lung

Gary W. Slater

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

Requirement Based Test Suite Generation Using Dependence Analysis

Olfa Chemli

A thesis submitted to

The Faculty of Graduate and Postdoctoral Studies of the University of Ottawa

in partial fulfillment of the requirements of the degree of

Masters in Computer Science*

School of Information Technology and Engineering

University of Ottawa

Ottawa, Ontario, Canada

---

* The Masters program in Computer Science is a joint program with Carleton University, administrated by the Ottawa Carleton Institute for Computer Science

Your file Votre référence
ISBN: 0-494-14889-6
Our file Notre référence
ISBN: 0-494-14889-6

Abstract

Requirement-based automated test case generation has been advocated as a model-based technique for generating a test suite related to individual requirements. The method proposed in this thesis consists in an approach of automating the generation of requirement-based test suites from requirements modeled by an Extended Finite State Machine (EFSM) and using EFSM Dependence Analysis. Two types of dependencies are identified between elements of the EFSM model and used to identify interactions among these elements. Patterns of interactions that affect a requirement under test are then developed to generate a test suite related to this requirement.

This thesis proposes algorithms to identify data and control dependencies in the EFSM model, algorithms to use EFSM dependencies in the generation of patterns of interactions related to a requirement under test and algorithms to derive a test suite related to these patterns of interactions. Also, a *Test Suite Generation* program (TSG) has been developed based on these algorithms and case studies have been performed to confirm the generation of efficient test suites for individual requirements.

Acknowledgements

First and foremost, I would like to thank Dr. Ural for his guidance and financial support throughout the course of my research. His exemplary practices have opened my eyes to how stimulating research can be. My special thanks go to Dr. Probert for his valuable support and encouragement and for giving me the opportunity to have my first work experience in IBM Markham. I wish to express my sincerest appreciation for Dr. Boyd and Dr. Amyot for their much needed understanding and help. Warmest thanks go out to my parents for instilling in me confidence and a drive for pursuing my graduate studies, for my close by and overseas dear family and friends and for my fellow ASERT members: Tuong, Panitee, Bo and Yan, for their support on many levels and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the past few years, research has been widely conducted in an attempt to improve the quality and reliability of large scale software systems. Although progress has been made on the formal proof of program correctness, proving large scale software systems correct by formal proof is very challenging, if not infeasible [Cla96]. Automated software tools have been found to be valuable in improving software reliability and attacking the high cost of developing and testing software systems. Product testers are often placed under severe pressure by the short release cycles expected in today's software markets. In the telecommunications domain, customers contract for large, custom-built systems and demand high reliability for their software. Issues like these have encouraged product test organizations to search for techniques that improve upon the traditional approach of manually writing test cases. In fact, test automation techniques offer a great deal of hope for testers.

In this thesis, we focused on system level automated test generation, which is often referred to as Model-Based Testing. Model-based test generation techniques are used to automatically generate tests for large software systems [Vay02]. This approach offers considerable promise in reducing the cost of test generation, increasing the efficiency of tests, and shortening the testing cycle. Automated test generation can be especially efficient for systems that are changed frequently, as testers can update the model and then rapidly regenerate a test suite, avoiding tedious and error-prone editing of a suite of hand-crafted tests.

1.2 Contribution of the Thesis

Generation of requirement-based test suites using EFSM dependence analysis is studied in this thesis. A novel method to automatically generate test suites related to individual requirements is proposed. The research conducted in this thesis is inspired from [Vay02] and from the work on data flow testing illustrated by Ural and Yang in [UY91, UY93]. We list four main contributions of the thesis.

Vaysbourg et al. [Vay02] proposed a method for test suite reduction based on dependence analysis. This method can be summarized as follows:

1. Given a set of system requirements expressed as an EFSM model, data and control flow dependencies of the EFSM model are captured in a directed graph called Static Dependence Graph (SDG) where EFSM transitions are represented as nodes, data and control dependencies are represented as plain and dashed directed edges, respectively.

   The first contribution of the thesis consists in writing an algorithm to formalize the generation of an SDG for a given EFSM model.

2. Given a test suite for each requirement under test in the EFSM model,
   a. Build a static interaction pattern (SIP) for each test sequence in this test suite
   b. Compare the obtained SIPs for the test sequences in the test suite to identify equivalent test sequences in the test suite (Two test sequences are considered to be *equivalent* wrt a requirement under test if these test sequences correspond to the same SIP)
   c. Remove all equivalent test sequences in the test suite except for one, to obtain a reduced test suite wrt to the requirement under test.

It is observed in [Vay02] that the size of the reduced test suite is bounded by the number of possible SIPs wrt a requirement under test. Interpreting this observation as for any EFSM model, the number of SIPs is bounded for a requirement under test, we wanted to demonstrate that the set of SIPs for a requirement under test, can be automatically generated.

This led to the second and most significant contribution of the thesis which consists in a method to automatically generate the set of SIPs for a requirement under test. This method takes as inputs an EFSM model and its corresponding SDG, and generates all possible SIPs from the SDG of the EFSM model. We call this method *Requirement based test generation using static dependence analysis*.

The third contribution of the thesis consists in an algorithm to form a test suite that maps to the generated set of SIPs for a requirement under test. Finally, the fourth contribution is a test suite generation tool which has been implemented using C++ under Sun Solaris. In addition, case studies have been performed to confirm that the reduced test suites can be generated by using the proposed method.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows: Chapter 2 reviews the related test generation techniques. Chapter 3 introduces the concepts related to dependence analysis in the EFSM model and gives an algorithm to generate the Static Dependence Graph from a given EFSM model. The test generation method we propose in this thesis is presented in Chapter 4 along with related algorithms. Chapter 5 gives the details of the Test Suite Generation program, which has been developed based on algorithms introduced in Chapter 4. This program has been used in three case studies, whose results

are presented and analyzed in Chapter 6. Chapter 7 gives our conclusions, with a summary of contributions and directions for future research.

Chapter 2

Related Work

2.1 A Review of Test Generation Techniques

To test the implementation of a given system, a *test suite* is needed. A test suite is a set of test cases, each test case being a realization of a test purpose. Test suites can be generated by using the information provided in specifications (black-box testing [Bei90]), designs (grey-box testing [Pro82]) or source code (white box testing [Mye04]). Test generation techniques utilize different criteria to select the set of test cases that will be used to exercise the system under test. Numerous test generation techniques are now available for generating test cases. Test case generation can be classified according to their mechanism of generating test cases into three main categories: random, path-oriented, and goal-oriented. Random test case generation techniques generate test cases by randomly selecting input values [Bir83]. Path-oriented test case generation techniques first select a path (in the program to be tested or in a control flow graph representing the program[1]) that will meet a testing requirement, and then attempt to find input values that cause that path to be executed. [Rap85] distinguishes two types of path-oriented techniques: those based on symbolic execution (or static), and those that are execution-based (or dynamic). With goal-oriented techniques, the path selection step is omitted and test cases covering a selected goal are identified without considering the path taken. Figure 2.1 below illustrates this classification of test case generation techniques.

[1] More details on the control flow graph of a program are given in Section 2.3

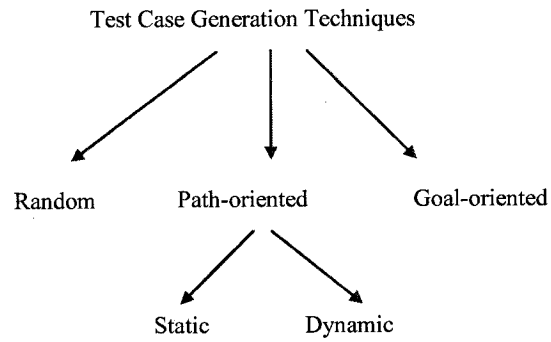


Figure 2.1 A classification of test generation techniques

2.1.1 Random Test Generation Techniques

Random test generation consists in selecting test data uniformly at random from the input domain of the program. Random testing is the simplest test generation method; however, it does not perform well in terms of coverage. Since it merely relies on probability, it has rather low chances finding semantically small faults, and thus accomplishing high level of coverage [Off96]. A semantically small fault is a fault that is only revealed by a small percentage of the program input.

For example, consider the following piece of code:

```
void foo(int a, int b) {
  if (a == b) then
     write(1);
  else
     write(2);
}
```

The probability of exercising the statement `write(1)` is $1/n$, where n is the maximum integer[2], as in order to execute this statement variables `a` and `b` must be equal.

[2] The maximum integer value is 2^{31} - 1, or 2 147 483 647 in C/C++

2.1.2 Path-oriented Test Generation Techniques

This technique consists in constructing a control flow graph for the program or the specification under test, and then selecting a set of paths in this flow graph that satisfy some selected testing requirement. Once a set of test paths is determined, then for every path in this set input data is sought that results in the execution of the selected path. Input data is defined as the values given to variables that appear in input statements of the program. In general, it is undecidable to determine whether input data exists to execute the program to follow a selected path. Two types of path-oriented approaches have been proposed to find input data to execute a selected path: the static approach, through the use of symbolic execution, and the dynamic approach, which obtains the necessary input data by executing the program under test.

2.1.2.1 Symbolic Execution Approach for Test Data Generation

Symbolic execution consists in replacing input parameters by symbolic values and in statically evaluating the statements along the control flow path. The goal of symbolic execution is to identify the constraints (either equalities or inequalities) called *path conditions* on symbolic input values under which a selected path is executed.

Symbolic execution is a useful technique for verification and testing, which has been widely studied during the past 30 years and used in various activities such as path domain checking, program proving, program reduction, symbolic debugging and test data generation [Kin96]. Symbolic execution, also referred to as symbolic evaluation, does not "execute a program" since the actual execution of a program requires real values (e.g., $x == 3$, $y == 10$) as the input data. It essentially uses symbols as values of variables and represents values of program variables as symbolic expressions, like $x == a0$, $y == b0$. A

set of expressions, one expression per output variable, is produced where the output values computed by a program are expressed as a function of the input symbolic values. For example, consider statement $z = x + 2y$. With ordinary program execution, the result of z will be a concrete value like 23. But with symbolic execution, the result is typically some symbolic expression like $a0 + 2b0$. After many steps of execution, a variable's value may become a very complex expression.

Symbolic execution is used for input data generation in the following way: after performing symbolic evaluation on a control flow path, the output variables will be represented by expressions in terms of symbolic values of input variables and constants. The symbolic input values in the expressions produced for each output variable can be substituted with actual values [Kin96]. The values substituted constitute an input data and the evaluation of the expression provides the corresponding output value. The creation of such values may be automated by using a numerical optimizer such as CPLEX[3]. One drawback of this approach is that symbolic execution may have difficulties handling complex expressions. Execution-based techniques [Kor90] alleviate some of these difficulties by incorporating dynamic execution information into the search for inputs.

2.1.2.2 Execution Based Approach for Test Data Generation

This approach is based on i) actual execution of a program under test (hence called dynamic approach), ii) dynamic data flow analysis and iii) function minimization methods [Kor90]. In this method, input data is determined as follows: when the program to be tested is executed on some input data, the program execution flow is monitored. If, during program execution, an undesirable execution control flow at some branch is

[3] CPLEX Optimization software products. CPLEX reads linear and integer problems in several formats, including MPS format and CPLEX LP formats and interfaces with several modeling languages, including GAMS, AMPL, and MPL.

observed, then a real-valued function is associated with this branch. This function is positive when a branch predicate (an edge labeled with a condition in the flow graph) is false and negative when the branch predicate is true. Function minimization search algorithms are used to automatically locate values of input variables for which the function becomes negative. In addition, dynamic data flow analysis is used to determine input variables which are responsible for the undesirable program behavior, leading to significant speed-up of the search process [Kor90].

2.1.3 Goal-oriented Test Generation Techniques

Two methods using goal-oriented technique have been found: the chaining approach and assertion-oriented approach (The latter is an extension of the chaining approach). Both techniques are considered dynamic as they involve program execution. The chaining approach and the assertion-oriented approach have been presented in [Fer96, Kor90] and implemented in the TESTGEN system [Fer96, Kor90].

The chaining approach uses data dependence (or data flow information) to find solutions to branch predicates. The purpose is to identify a chain of nodes in the flow graph representing the program under test that are vital to the execution of the goal node. This chain is built up iteratively during execution. The next step consists in finding a path that traverses this chain.

In assertion-oriented testing, certain conditions, called *assertions* are either manually or automatically inserted in the code. An assertion is defined as a statement containing Boolean expressions that are added to the code to be tested. When an assertion is executed it is supposed to hold, otherwise there is an error in the program (or possibly in

the assertion). The goal of assertion-oriented test generation is to find a path (i.e., any path) to an assertion that does not hold. For instance, consider the following code.

```
void fie(int a) {
  int b = (a+1)*(a-1);
  assert(b != 0);
  write(1/b);
}
```

The assertion `assert(b!=0)` is used to stress that before executing `1/b` the variable `b` must not be zero.

Table 2.1 summarises the test generation techniques described above and provides related examples.

Table 2.1 A classification of test generation techniques

	Technique Evaluation	Examples
Random	- Randomly generates input data - Considered as the weakest of the three approaches - Generates a large number of test cases but does not guarantee test suite coverage	– Pure random – Guided by number of cases – Error guessing
Path-Oriented	- A set of paths that satisfy some selected testing requirement is determined then input data is derived for every path which results in the execution of the selected path. - Considered as the strongest in the three approaches **Input data generation based on symbolic execution** - This method leads to several problems : the growth of intermediate algebraic expressions, and the difficulty to deal with arrays - Using symbolic execution corresponds to an exhaustive exploration of all paths going through a selected point **Execution-oriented input data generation** these techniques take advantage of the actual variable values obtained during execution to try to solve problems with complex expressions	– Decision/condition Coverage – Path Coverage – All-definitions – All-uses – All-du-paths – Branch Coverage – Statement Coverage
Goal-Oriented	- Focus on the final goal rather than on a specific path - Concentrate on executions that can be determined (e.g. through the use of data dependence information) to possibly influence progress toward the goal.	– The chaining approach – The assertion-oriented approach

2.2 Model Based Testing

Model-based test generation techniques automate test generation for software systems based on a system model. These techniques are used for system testing where a model representing the system behavior is developed, from which a test suite is generated automatically [Vay02]. In order to eliminate the problems of ambiguity during the construction and test of a system, requirements specification should be formalized in a precise notation with a unique interpretation. A system model may be generated by using formal specification languages like EFSM [Bou97], SDL [Sar87] and LOTOS [ISO89], or may be designed by software engineers through a modeling notation like UML [Dal98].

2.2.1 Formal Specification

Formal specification representations are powerful abstraction tools, often used to describe behavioral specification for model-based techniques. Formal specification is part of a more general collection of techniques that are known as 'Formal Methods'. Formal methods include Formal specification; Specification analysis and proof; Transformational development and Program verification, all are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics. Formal methods may be used in all phases of computer system development, including requirements analysis, system design and system testing. Within the context of software testing, a formal method typically provides a formal specification language like FSM, EFSM, LOTOS, etc... for specifying system properties to be tested, and a set of guidelines for developing the specifications in a systematic manner, and often facilitates the construction of tools capable of automatically translating the specifications into executable tests. The intended system

behavior is sometimes described in specification documents that exhibit a tendency to be incomplete and ambiguous. The idea of model-based testing is to make the intended system behavior explicit, in the form of behavior models. Once these models have been determined to accurately reflect the actual requirements, the issue of test generation can be addressed.

The Finite State Machine (FSM) model is very popular in expressing the control aspects of state-based systems and many related analysis methods have been developed [Gil62]. FSMs support formal test derivation techniques which can be used for validation and testing purposes. They are often used to describe the control structure of a system when important aspects of this system are described in terms of state transitions. However, FSMs cannot deal with the data aspects of system behavior [Hie04]. This shortcoming can be alleviated by using the Extended Finite State Machine (EFSM) model. An EFSM is an FSM with some extensions: each logical state has some associated memory and each transition involves the application of an operation (an action) that takes the memory and input of this transition, and generates output and new value for the memory [Hie04]. Before we review test generation techniques that are related to the method proposed in this thesis and that are adapted to the EFSM model, we give a definition of the EFSM model.

2.2.2 Extended Finite State Machine Model

An EFSM (**E**xtended **F**inite **S**tate **M**achine) M is a quadruple (S, s_{en}, s_{ex}, T,) where

- S: finite set of states

- s_{en}: entry state

- s_{ex}: exit state

- T: finite set of transitions

For example, an EFSM model of an Automated Teller Machine (ATM) [Vay02] is graphically described in Figure 2.2. The ATM of Figure 2.2 supports three transactions: withdrawal, deposit and balance inquiry. Before transactions can be performed, user must enter a valid pin that is matched against the PIN stored in user's ATM card. A maximum of three attempts is allowed to enter the valid PIN.

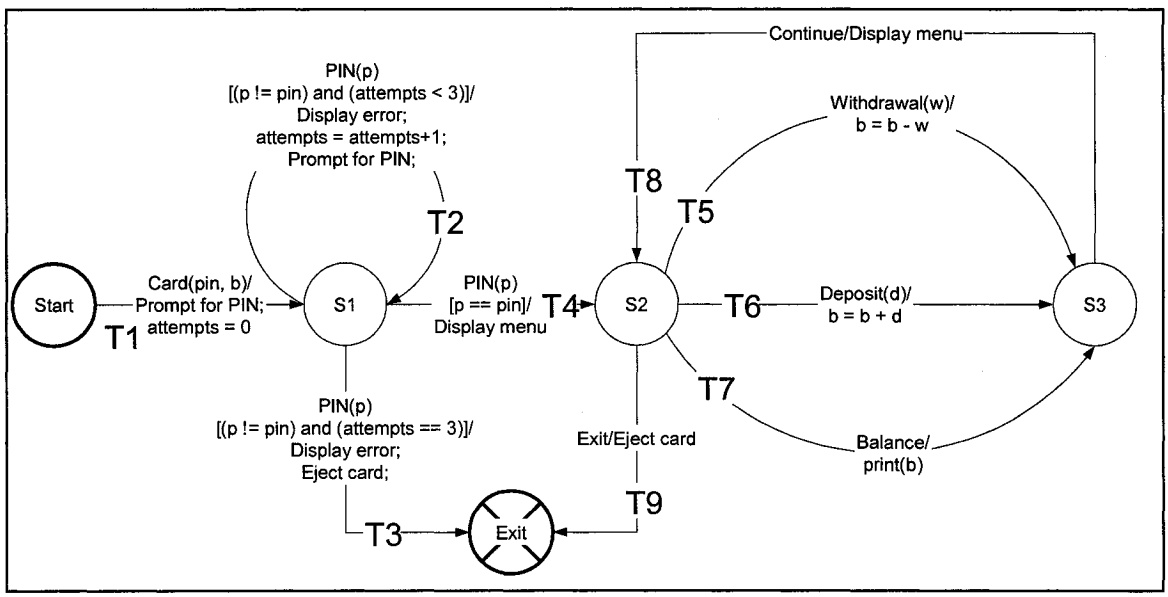


Figure 2.2 ATM EFSM Graph

This ATM system is represented as an EFSM $M = (S, s_{en}, s_{ex}, T)$ where

- S = {Start, S1, S2, S3, Exit}

- s_{en}= Start

- s_{ex} = Exit

- T = {T1, T2, T3, T4, T5, T6, T7, T8, T9}.

Each transition $t \in T$ in an EFSM M is a quintuple (s_s, s_d, i, g, a), where

- s_s: current state
- s_d: next state
- i: input interaction
- g: enabling predicate
- a: sequence of actions

For an EFSM transition t, s_s and s_d are states in S representing the starting state and the terminating state of t, respectively. i is the input interaction that triggers t, g is a Boolean condition that must be evaluated to be true for t to be executable, and a is a sequence of actions that take place when t is executed. An action may be an assignment, output, set, reset or procedure call statement. Elements of an EFSM transition are depicted as illustrated by Figure 2.3 below. Clearly, a transition t in the EFSM may or may not have a predicate.

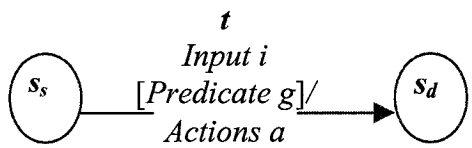


Figure 2.3 Elements of an EFSM transition t

Transitions in the EFSM model of the ATM system shown in Figure 2.2 are:

- T1 = (Start, S1, Card(*pin*, *b*), (), (Prompt_for_PIN(), *attempts* = 0))
- T2 = (S1, S1, PIN(*p*), ((*p* != *pin*) and (*attempts* < 3)), (Display_error(), *attempts* = *attempts* + 1, Prompt_for_PIN()))
- T3 = (S1, Exit, PIN(*p*), ((*p* != *pin*) and (*attempts* = = 3)), (Display_error(), Eject_card()))
- T4 = (S1, S2, PIN(*p*), *p* = = *pin*, Display_menu())

- T5 = (S2, S3, Withdrawal(w), (), $b = b - w$)

- T6 = (S2, S3, Deposit(d), (), $b = b + d$)

- T7 = (S2, S3, Balance(),(), print(b))

- T8 = (S3, S2, Continue(), (), Display_menu())

- T9 = (S3, Exit, Exit(),(), Eject_card())

Note that, in general, an EFSM model may include, in the actions part of a transition, conditional statements such as *if-then-else* or *case* statements as well as repetitive statements such as *for* or *while* statements. In this work, we do not consider EFSMs where transitions have conditional and repetitive statements. Interested reader may refer to [USW00] for handling such statements within the context of the EFSM model.

In this thesis, the EFSM model as defined above, is used to represent system specifications. We assume that a system specification consists of a set of individual requirements where each requirement describes a partial functional behavior of the system. In addition, we assume that requirements can be represented as a single EFSM and each requirement can be adequately represented by a single transition in this EFSM. Therefore "Requirement under test" will also be referred to as "Transition Under Test" or TUT (which is highlighted in bold) in the remaining parts of the thesis.

Figure 2.2 shows an EFSM describing the requirements for a simplified ATM System. Requirements are traditionally non-executable. The set of requirements for the ATM system presented in the EFSM model in Figure 2.2 is given as a set $R = \{r1, r2, r3, r4, r5\}$ where each requirement r is represented by a TUT in the EFSM:

(*x*: *T2 may be inserted 0, 1, 2 or 3 times*)

***r*1**: User fails to enter a correct pin that matches with the PIN stored in the ATM card in less than four attempts. If pins don't match, and less than four attempts were performed, the system displays an error message, increments the number of attempts and prompts for pin. At the fourth attempt, if user still fails to enter a correct pin, the system prints an error message and ejects the user's ATM card.

$r1$ = T2 T2 T2 **T3**

***r*2**: After entering a correct pin in less than four attempts, user selects the withdrawal function. The system adjusts the balance, and displays a menu with withdrawal, deposit, balance inquiry, and exit functions.

$r2 = x$ T4 **T5**

***r*3**: After entering a correct pin in less than four attempts, the user selects the deposit function. The system adjusts the balance, and displays a menu with withdrawal, deposit, balance inquiry, and exit functions.

$r3 = x$ T4 **T6**

***r*4**: After entering a correct pin in less than four attempts, the user selects the balance function. The system displays a menu with withdrawal, deposit, balance inquiry, and exit functions.

$r4 = x$ T4 **T7**

***r*5**: After entering a correct pin in less than four attempts, the user selects the exit function and the system ejects the user's ATM card.

$r5 = x$ T4 **T9**

The EFSM model has been used for system specification and for automatic test case generation in the context of model based testing [Dal98]. Several approaches can be used

to develop tests from the EFSM model. The following section reviews some of these techniques.

2.3 Testing Criteria

White box testing criteria, i.e. criteria which take into account the structure of the source code, are divided into control flow and data flow criteria. A number of control and data flow criteria have been proposed. Control flow testing criteria test the control structure of the implementation. In contrast data flow testing criteria test ways in which data may be transferred. Data and control flow testing criteria have been shown to be applicable to the selection of test sequences from an EFSM by transforming the EFSM into a special flow graph which is a directed graph (digraph) representing the source code or the specification of the system under test [UY91]. A method to transform an EFSM $M = (S, s_{en}, s_{ex}, T)$ into a flow graph F is proposed in [UY91]. Accordingly, $F = (N, en, ex, E)$ where N is the set of nodes in F; en and $ex \in N$ are the entry and exit nodes of F, respectively; E is the set of edges in F. Ritthiruangdech in [Rit04] transformed the EFSM model for the ATM system shown in Figure 2.2 of this thesis to obtain the flow graph represented in Figure 2.4 below by following the approach in [UY91]:

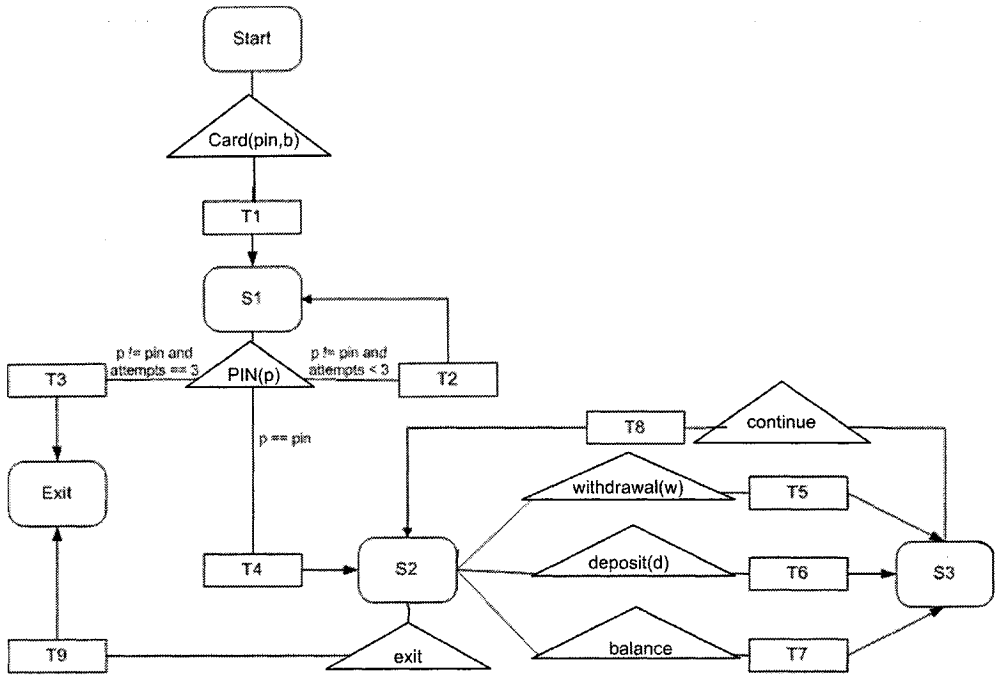


Figure 2.4 Flow graph F generated from the ATM EFSM model in Figure 2.2

Before discussing some commonly used data and control flow testing criteria, the following terminology given in [UY93] is reviewed:

A *path* $(n_1, n_2, \ldots, n_m)$ in a digraph F is a sequence of nodes in F, such that for all i, $1 \leq i \leq m\text{-}1$, $m \geq 2$, (n_i, n_{i+1}) is an edge in F. A *sub-path* of a path $(n_1, ..., n_m)$ is a path $(i_1, ..., i_k)$ if there exists a δ, $0 \leq \delta \leq m\text{-}k$, such that for all j, $1 \leq j \leq k$, $i_j = n_{j+\delta}$. A *loop-free path* is a path in which all nodes are distinct. A *complete path* in a single entry, single exit digraph F is a path whose first node is the entry node and whose last node is the exit node of F. A complete path is *executable* or *feasible* if some input data which causes its execution exists, and *un-executable* (*infeasible*) otherwise.

Let $p = (n_1, ..., n_m)$ be a complete path in a digraph F. We say that a node i is *covered* by p if $i = n_j$ for some j, $1 \leq j \leq m$. Similarly, an edge (i_1, i_2) is covered by p if $i_1 = n_j$ and $i_2 = n_{j+1}$ for some j, $1 \leq j \leq m\text{-}1$. A path $(i_1, ..., i_k)$ is covered by p if path $(i_1, ..., i_k)$ is a sub-

path of p. A node, edge, or path is covered by a set C of complete paths of F if the node, edge, or path, respectively is covered by a complete path in C.

For example, take the ATM EFSM illustrated in Figure 2.2. The EFSM is represented by a digraph $F = (N, E)$. We assume that N, the set of nodes in F, corresponds to the set of states S in the EFSM, and E, the set of edges in F, corresponds to the set of transitions T in the EFSM. Let C be a set of complete paths starting at the start node and terminating at the exit node in F. For instance, a complete path (T1, T4, T5, T8, T6, T8, T7, T8, T9) is in C and covers nodes: *start*, $S1$, $S2$, $S3$ and Exit; covers edges: T1, T4, T5, T6, T7, T8 and T9; and covers a path: (T1, T4, T5, T8, T6).

2.3.1 Control Flow Testing Criteria

Control flow testing criteria examine logical expressions which determine the branch and loop structures of the specification. The commonly used control flow testing criteria are known from the 1960s and 70s. The following are based on the well-known book by Myers [Mye04]. Let C be a set of complete paths of a digraph F.

- **All-nodes criterion** (also called statement coverage)

  For C to satisfy the all-nodes criterion in F, it is required that every node of F is covered by C at least once.

- **All-edges criterion**

  For C to satisfy the all-edges criterion in F, it is required that every edge of F is covered by C at least once.

  For instance, C has been formed to satisfy the all-edges criterion in the ATM EFSM represented by Figure 2.2. Table 2.2 shows that C contains two complete test

paths in the ATM EFSM that cover all the edges in this EFSM. Note that the size of C is minimal.

Table 2.2 All-edges criterion applied to the ATM EFSM

Test Path	Edges covered
T1, T2, T2, T2,T3	T1, T2, T3
T1, T4, T5, T8, T6, T8, T7, T8, T9	T1, T4, T5, T6, T7, T8, T9

- **All-paths criterion**

For C to satisfy the all-paths criterion in F, it is required that every complete path of F is included in C. This is a very stringent criterion which becomes infeasible for complex specifications due to the large number of possible paths. In the presence of loops, a constraint is usually set on the number of occurrences of a loop in each path.

The next control flow testing criteria use concepts of *decision* and *condition*. A decision is a point in the specification at which the control flow is split into various paths. A decision is a Boolean expression consisting of one or several conditions combined by logical connectives. A condition is an elementary Boolean expression (atomic predicate) which cannot be divided into further Boolean expressions.

- **Decision criterion** (also called branch coverage)

For C to satisfy the decision criterion in F, it is required that every possible outcome of each decision (i.e. branch) in F is covered by C at least once.

- **Condition criterion**

For C to satisfy the condition criterion in F, it is required that every condition in each decision in F has evaluated to true at least once and to false at least once.

- **Decision/Condition criterion**

For C to satisfy the decision/condition criterion in F, it is required that every decision in F has taken all possible outcomes at least once, and every condition in each decision in F evaluated to true at least once and to false at least once.

- **Multiple condition criterion**

For C to satisfy the multiple condition criterion in F, it is required that all possible combinations of condition outcomes in each decision in F have been invoked at least once.

For example, consider a decision d and three conditions α, β, and γ in F such that d is *true* if and only if any two of these conditions are true, hence $d = (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \vee (\beta \wedge \gamma)$. Conditions are evaluated to 1 (*true*) or 0 (*false*). Hereafter a condition is represented by its value (instead of using its symbol) i.e., $\alpha = 1$, $\beta = 0$, and $\gamma = 1$ is the same as (1, 0, 1). Eight combinations of the values of the conditions exist: (1, 1, 1), (1, 1, 0), (1, 0, 1), (1, 0, 0), (0, 1, 1), (0, 1, 0), (0, 0, 1), (0, 0, 0). Table 2.3 below gives one example that consists of a subset of combinations that satisfy a criterion for each of the Decision, Condition, Decision/Condition and Multiple Condition criteria. Each column shows the subset of the combinations marked with '×' that satisfy the corresponding criterion:

Table 2.3 Decision and condition based criteria

Combination #	Values				Decision criterion	Condition criterion	Decision/ Condition criterion	Multiple Condition criterion
	α	β	γ	d				
1	1	1	1	1		×		×
2	1	1	0	1			×	×
3	1	0	1	1				×
4	1	0	0	0	×			×
5	0	1	1	1	×			×
6	0	1	0	0				×
7	0	0	1	0			×	×
8	0	0	0	0		×		×

For the decision criterion, two combinations are sufficient as *d* evaluates to *true* with (0, 1, 1) and to *false* with (1, 0, 0). For the conditions criterion, two combinations are sufficient, where each of the three conditions evaluates to *true* in one combination and to *false* in the other: In the pair (1, 1, 1) and (0, 0, 0) each condition evaluates to *true* once and to *false* once, so these two combinations satisfy the condition criterion. Any subset of combinations that satisfy both the decision and condition criterion also satisfy the decision/condition criterion, e.g., (1, 1, 0) and (0, 0, 1). For the multiple condition criterion, the set containing all eight combinations is required.

2.3.2 Data Flow Testing Criteria

Typically data flow testing criteria consider definitions and uses of variables. Then certain associations between definitions and uses of variables required to be covered in a given criterion are established, by applying conventional data flow analysis upon a flow graph *F* representing the system specification.

2.3.2.1 Data Flow Related Concepts

A *definition* (referred to as def) of a variable *v* at node *n* (denoted by d_n^v) is an occurrence of *v* by which a value is assigned to *v* (e.g., an occurrence of *v* in an input statement or on the left hand side of an assignment statement, for instance, the definition of variable "*b*" at node T5 in Figure 2.4). A *use* of a variable *v* is an occurrence of *v* by which the value of *v* is referenced. Each use occurrence is also further classified as a computation use (*c-use*) or a predicate use (*p-use*) [Rap85, Fra88]. A *c*-use of a variable *v* at node *i* (denoted by c_i^v) is a use of *v* that occurs in the right hand side of assignment statements, arguments in a procedure call or output statements, for instance, the use of variable "*w*" at node T5

in Figure 2.4. On the other hand, a *p*-use of a variable *v* on edge (i, j) (denoted by $p^{v}_{(i,j)}$) is a use that occurs in Boolean expressions in a condition statement or in a repetitive statement, for instance, the use of variable "*p*" on the edge (*PIN*(*p*), T4) in Figure 2.4.

A path $(n_1, n_2, \ldots, n_{m-1}, n_m)$ in a flow graph is a *def-clear path with respect to a variable v from node* n_1 *to node* n_m or *to edge* (n_{m-1}, n_m) if either $m = 2$ or $m > 2$ and there is no definition of *v* at nodes n_2 to n_{m-1}. d^{v}_{i} and c^{v}_{j} form a *du-pair* wrt variable *v* (represented by a tuple (d^{v}_{i}, c^{v}_{j})) if there is a def-clear path with respect to *v* from node *i* to node *j*. Similarly, d^{v}_{i} and $p^{v}_{(j,k)}$ form a du-pair wrt variable *v* (represented by a tuple $(d^{v}_{i}, p^{v}_{(j,k)})$) if there is a def-clear path with respect to *v* from node *i* to edge (j, k).

For example, consider the du-pair (d^{b}_{T1}, c^{b}_{T6}) in the ATM EFSM represented by the flow graph *F* in Figure 2.4. The definition of *b* in input node *Card*(*pin*, *b*) in *F* can reach the *c*-use of *b* at node T6 in *F* through the def-clear path "*Card*(*pin*, *b*), T1, S1, *PIN*(*p*), T4, S2, *deposit*(*d*), T6". Next, consider the du-pair $(d^{pin}_{T1}, p^{pin}_{(PIN(p),T2)})$. The definition of *pin* at node *Card*(*pin*, *b*) in *F* can reach the *p*-use of *pin* on the edge (*PIN*(*p*), T2) in *F* through the def-clear path "*Card*(*pin*, *b*), T1, S1, *PIN*(*p*), T2" in *F*.

For ease of reference, u^{v}_{q} denotes henceforth either c^{v}_{q}, if *q* represents a node or p^{v}_{q} if *q* represents an edge. A *du-pair is covered by* a set *C* of complete paths in a flow graph *F* if a def-clear path for the du-pair is a subpath of a complete path contained in *C*.

A path $(n_i, \ldots, n_j, n_k)$ is a *du-path* wrt variable *v* if n_i has a definition of *v* and either

- n_k has a *c*-use of *v* and $(n_i, \ldots, n_j, n_k)$ is a loop-free def-clear path wrt. *v*, or
- (n_j, n_k) has *p*-use of *v* and $(n_i, \ldots, n_j, n_k)$ is a loop-free def-clear path wrt. *v*

Table 2.4 identifies existing defs, *c*-uses in nodes and *p*-uses on edges of the flow graph of the ATM EFSM given in Figure 2.4.

Table 2.4 Identification of defs, *c*-uses and *p*-uses in the ATM EFSM

Node	def	*c*-use	Edge	*p*-use
T1	{*attempts*, *b*, *pin*}	ϕ	S1, T2	{*attempts*, *pin*, *p*}
T2	{*attempts*, *p*}	{*attempts*}	S1, T3	{*attempts*, *pin*, *p*}
T3	{*p*}	ϕ	S1, T4	{*p*, *pin*}
T4	{*p*}	ϕ		
T5	{*w*, *b*}	{*w*, *b*}		
T6	{*b*, *d*}	{*b*, *d*}		
T7	ϕ	{*b*}		

An association between definitions and uses of a variable in the flow graph is tested by executing subpaths from the definition (def) of this variable to a point where the variable is used (use). The following are common data flow testing criteria. Interested reader may refer to [Fra88] for more details.

- **All-Defs criterion**

  For every variable *v*, at least one def-clear path from every definition of *v* to at least one *c*-use or one *p*-use of *v* must be covered.

- **All-Uses criterion** (Also called all du-pair criterion)

  For every variable *v*, at least one def-clear path from every definition of *v* to every use of *v* must be covered.

- **All-*P*-Uses/Some *C*-Uses criterion**

  For every variable *v*, at least one def-clear path from every definition of *v* to every *p*-use of *v* must be covered. In addition, if a definition of *v* has no *p*-use, at least one def-clear path to one *c*-use of *v* must be covered.

- **All-*C*-Uses/Some *P*-Uses criterion**

For every variable v, at least one def-clear path from every definition of v to every c-use of v must be covered. In addition, if a definition of v has no c-use, at least one def-clear path to one p-use of v must be covered.

- **All-du-paths criterion**

For every variable v, every du-path from every definition of v to every c-use and every p-use of v must be covered.

Table 2.5 below shows the set of du-pairs in the ATM EFSM and a corresponding def-clear path (in the ATM EFSM shown in Figure 2.2) for each du-pair. Each def-clear path is associated with a path predicate. A path predicate is a conjunction of Boolean expressions such that this path will be traversed only if the associated path predicate is true.

Table 2.5 Du-Pairs in the ATM EFSM

#	Du-Pair	Def Clear Path	Use	Path Predicate
1	$[d^{attempts}_{T1}, c^{attempts}_{T2}]$	(T1, T2)	c-use	$attempts == 0 \wedge p \neq pin \wedge attempts < 3$
2	$[d^{attempts}_{T2}, c^{attempts}_{T2}]$	(T2, T2)	c-use	$p \neq pin \wedge attempts < 3$
3	$[d^{attempts}_{T1}, p^{attempts}_{(S1,T2)}]$	(T1, T2)	p-use	$p \neq pin \wedge attempts < 3$
4	$[d^{attempts}_{T2}, p^{attempts}_{(S1,T2)}]$	(T2, T2)	p-use	$p \neq pin \wedge attempts < 3$
5	$[d^{attempts}_{T2}, p^{attempts}_{(S1,T3)}]$	(T2, T3)	p-use	$p \neq pin \wedge attempts == 3$
6	$[d^{b}_{T1}, c^{b}_{T5}]$	(T1, T4, T5)	c-use	$attempts == 0 \wedge p == pin$
7	$[d^{b}_{T1}, c^{b}_{T6}]$	(T1, T4, T6)	c-use	$attempts == 0 \wedge p == pin$
8	$[d^{b}_{T1}, c^{b}_{T7}]$	(T1, T4, T7)	c-use	$attempts == 0 \wedge p == pin$
9	$[d^{b}_{T5}, c^{b}_{T5}]$	(T5, T8, T5)	c-use	
10	$[d^{b}_{T5}, c^{b}_{T6}]$	(T5, T8, T6)	c-use	
11	$[d^{b}_{T5}, c^{b}_{T7}]$	(T5, T8, T7)	c-use	
12	$[d^{b}_{T6}, c^{b}_{T5}]$	(T6, T8, T5)	c-use	
13	$[d^{b}_{T6}, c^{b}_{T6}]$	(T6, T8, T6)	c-use	

14	$[d^{b}_{T6}, c^{b}_{T7}]$	(T6, T8, T7)	*c*-use	
15	$[d^{d}_{T6}, c^{d}_{T6}]$	(T6)	*c*-use	
16	$[d^{w}_{T5}, c^{w}_{T5}]$	(T5)	*c*-use	
17	$[d^{p}_{T2}, p^{p}_{(S1,T2)}]$	(T2)	*p*-use	$p \neq attempts \wedge attempts < 3$
18	$[d^{p}_{T3}, p^{p}_{(S1,T3)}]$	(T3)	*p*-use	$p \neq pin \wedge attempts == 3$
19	$[d^{p}_{T4}, p^{p}_{(S1,T4)}]$	(T4)	*p*-use	$p == pin$
20	$[d^{pin}_{T1}, p^{pin}_{(S1,T2)}]$	(T1, T2)	*p*-use	$p \neq pin \wedge attempts < 3$
21	$[d^{pin}_{T1}, p^{pin}_{(S1,T4)}]$	(T1, T4)	*p*-use	$attempts == 0 \wedge p == pin$
22	$[d^{pin}_{T1}, p^{pin}_{(S1,T3)}]$	(T1, T2, T2, T2, T3)	*p*-use	$attempts == 0 \wedge p \neq pin \wedge attempts < 3 \wedge p \neq pin \wedge attempts + 1 < 3 \wedge p \neq pin \wedge attempts + 1 + 1 < 3 \wedge p \neq pin \wedge attempts + 1 + 1 + 1 == 3$

Note that some of the def-clear paths shown in Table 2.5 have empty path predicates due to the absence of a predicate statement in each of the EFSM transitions traversed in these paths, for instance, among du-pairs of the ATM EFSM shown in Table 2.5, du-pair $[d^{b}_{T5}, c^{b}_{T5}]$ is associated with def-clear path (T5, T8, T5). There are no predicate statements in transitions T5 and T8 in the ATM EFSM shown in Figure 2.2, hence the path predicate for path (T5, T8, T5) is empty. Also, note that each def-clear path shown in Table 2.5 is feasible and as a result it has no path predicate or has a path predicate that evaluates to true. Path predicates are usually formed by using the symbolic execution technique discussed in Section 2.1.2.1 of this chapter.

We have also applied the all-uses (or all-du-pairs) criterion to the ATM EFSM and obtained the results shown in Table 2.6. Note that the "du-pairs covered" column refers to the "du-pair number" column in Table 2.5.

Table 2.6 All-Uses criterion applied to the ATM EFSM

Test Path	Du-Pairs covered
(T1, T4, T5, T8, T5, T8, T6, T8, T5, T8, T7, T8, T6, T8, T6, T8, T7, T8, T9)	6, 9, 10, 11, 12, 13, 14, 15, 16, 19, 21
(T1, T4, T6, T8, T9)	7, 15, 19, 21
(T1, T4, T7, T8, T9)	8, 19, 21
(T1, T2, T2, T2, T3)	1, 2, 3, 4, 5, 17, 18, 20, 22

2.3.2.2 Other Data Flow Related Concepts

The data flow testing criteria presented so far focus on tracing data flow between the definition of a variable and uses of the same variable in a flow graph. There are other data flow testing criteria that emphasize the data flow interactions between several and possibly different variables. *Laski and Korel's Ordered Context Coverage* criterion for instance, emphasise that a node in a flow graph may contain uses of several different variables in which each use may be reached by several different definitions through the same path [LK83]. On the other hand, *Ntafos' Required k-Tuples Criteria* emphasize interactions between different variables [Nta84]. These criteria require that a path set cover chains of alternating definitions and uses, called *k-dr interactions*. Each definition in a *k-dr* interaction reaches the next use in the chain, which occurs at the same node as the next definition in the chain. Thus a *k-dr* interaction propagates information along a subpath, which is called an interaction subpath for the *k-dr* interaction [CP89].

The df-chains coverage criterion reviewed next analyses relationships between a use of a variable and the definition of possibly another variable. This criterion is of particular

interest to us as we make use of this criterion in the formulation of the test generation method proposed in this thesis.

The **df-chains criterion** is based on notion of *affect* [UY91]: *a use of a variable y, which may either be a c-use or a p-use, is affected by a def of a variable x* if either

a) x and y are the same variable and the use of y is reached by the def of x through a def-clear path wrt x in the flow graph of an EFSM, or

b) a def of a variable z is given in terms of a use of x at a node which is reached by the def of x through a def-clear path wrt x and the use of y is affected by the def of z.

Accordingly, the data flow is traced from a du-pair (d_i^v, u_j^v) where variable v is defined in node i to a du-pair (d_k^w, u_l^w) where variable w is used in node l, such that d_i^v affects u_l^w. Variables v and w can be either the same or different variables. If du-pairs in an EFSM are linked by the notion of affect, *data-flow-chains* (*df-chains*) can be formed.

[UY91] identifies df-chains starting with an input and terminating with a variable output. Input is a def of a variable v in either an input statement where v takes a value or in an assignment statement where variable v is assigned a constant value. An example of input is d_{T1}^b in Figure 2.2. Variable output is a *c*-use of a variable v in an output statement, for instance, c_{T7}^b in Figure 2.2.

A *df-chain* is an ordered sequence $(d_{n1}^{x1}, u_{n2}^{x1}, d_{n2}^{x2}, u_{n3}^{x2}, \ldots, d_{nm}^{xm}, u_{nm+1}^{xm})$ of du-pairs $(d_{n1}^{x1}, u_{n2}^{x1})$, $(d_{n2}^{x2}, u_{n3}^{x2}), \ldots, (d_{nm}^{xm}, u_{nm+1}^{xm})$, such that $m \geq 1$ and u_{nm+1}^{xm} is affected by d_{n1}^{x1} [UY93]. An *activating path* for a df-chain $(d_{n1}^{x1}, u_{n2}^{x1}, d_{n2}^{x2}, u_{n3}^{x2}, \ldots, d_{nm}^{xm}, u_{nm+1}^{xm})$ is a path $(n_1, \ldots, n_2, \ldots, n_3, \ldots, n_m, \ldots, n_{m+1})$ in which $(n_i, \ldots, n_{i+1})$ is a def-clear path with respect to variable x_i for $1 \leq i \leq m$ [UY93].

After having defined the notion of df-chain, we refer to the df-chains criterion given in [UY91]. It is said that the **df-chains criterion** is satisfied by a set of complete paths C in the flow graph of an EFSM if all activating paths for every df-chain wrt each variable v in the EFSM are covered by C. To ensure that the set of activating paths for a df-chain *df* is finite, each EFSM loop is traversed 0 or 1 times in an activating path of a *df* (or in a def-clear path of du-pairs of *df*).

For example, we have applied the df-chains coverage criterion to the ATM EFSM in Figure 2.2, for variables used in TUT = T5. Table 2.7 below shows the obtained df-chains:

Table 2.7 df-chains criterion applied to the ATM EFSM

ID	Df-chain	An activating path for the df-chain
1	(d_1^b, u_5^b)	(T1, T4, T5)
2	$(d_1^b, u_5^b)(d_5^b, u_5^b)$	(T1, T4, T5, T8, T5)
3	$(d_1^b, u_5^b)(d_5^b, u_6^b)(d_6^b, u_5^b)$	(T1, T4, T5, T8, T6, T8, T5)
4	$(d_1^b, u_6^b)(d_6^b, u_5^b)$	(T1, T4, T6, T8, T5)

Chapter 3

Dependence Analysis in the EFSM Model

Authors in [Vay02] use EFSM dependence analysis in their approach of reduction of the number of test cases in a given test suite for an EFSM model as follows: two types of dependencies, namely *data dependencies* and *control dependencies* are identified between elements of the EFSM model. These dependencies capture potential interactions between elements of the model and are used to determine parts of the model that affect a requirement under test. Next, two types of dependence analysis are used: *static dependence analysis* and *dynamic dependence analysis* to capture a pattern of interaction for each test case, which is employed later to formally identify equivalent test cases. According to [Vay02], the static and dynamic dependence analysis are different, such that static dependence analysis ignores repetitions of the same dependencies between transitions, whereas dynamic dependence analysis takes into account repetition of the same dependencies. Since dynamic dependence analysis requires a given test sequence, and our aim is to generate test sequences, only static dependence analysis is used in this thesis.

3.1 Static Dependence Analysis in the EFSM Model

Two types of static dependencies between transitions (requirements) in the EFSM model are identified: *Data Dependence* and *Control Dependence*. Data and control dependencies are *static* because they are defined only in terms of (the definition of) the EFSM model (no traversal of the EFSM model is involved) [Vay02].

Data dependence defines a relationship between definitions and uses of variables in the EFSM. Control dependence, on the other hand, derives from the concept of post-dominance explained later, and exists when the execution of a transition is conditional upon the execution of another one.

3.1.1 Data Dependence

Data dependence captures the notion that one transition defines a value to a variable and the same or another transition may potentially use this value [Vay02]. Formally, *there exists a data dependence from transition t_i to transition t_k wrt a variable v in the EFSM* if

1) d^{v}_{ti} is the last definition of v in t_i

2) u^{v}_{tk} is a *c*-use or *p*-use of v in t_k before v is redefined in t_k (if any)

3) there exists a def-clear path wrt v from t_i to t_k

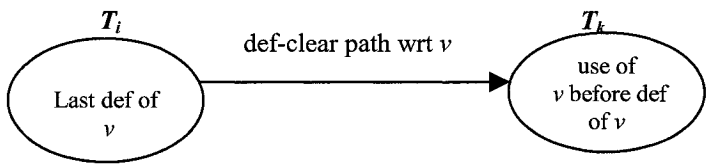


Figure 3.1 Illustration of the concept of data dependence

For example, consider transitions T1 and T2 and variable *attempts* in the ATM EFSM presented in Figure 2.2:

1) variable *attempts* is defined in the action of T1 and *attempts* is not redefined in the action of T1,

2) variable *attempts* is used in the action of T2 before it is defined in T2,

3) there is a path (T1, T2) from T1 to T2 inclusively along which *attempts* is not redefined.

Therefore, there is data dependence wrt variable *attempts* from transition T1 to transition T2 in the ATM EFSM. All data dependencies in the ATM EFSM are listed in Table 3.1

Table 3.1 Data dependencies in the ATM EFSM

Transition	Data dependence	wrt Variable	Path
T1	T2	*attempts*	(T1, T2)
		pin	(T1, T2)
	T3	*pin*	(T1, T2, T2, T2, T3)
	T4	*pin*	(T1, T4)
	T5	*b*	(T1, T4, T5)
	T6	*b*	(T1, T4, T6)
	T7	*b*	(T1, T4, T7)
T2	T2	*p*	(T2, T2)
	T3	*p*	(T2, T3)
T5	T5	*b*	(T5, T8, T5)
	T6	*b*	(T5, T8, T6)
	T7	*b*	(T5, T8, T7)
T6	T5	*b*	(T6, T8, T5)
	T6	*b*	(T6, T8, T6)
	T7	*b*	(T6, T8, T7)

3.1.2 Control Dependence

Control dependence captures the notion that one transition may affect the traversal of another transition. Control dependence between transitions in the EFSM is defined in terms of the concept of *post-dominance.* Let *Y* and *Z* be two states in the EFSM, and *T* be an outgoing transition from *Y* as represented in Figure 3.2:

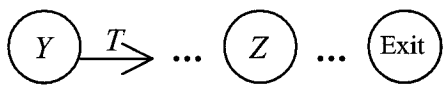


Figure 3.2 Illustration of the concept of post dominance

State *Z post-dominates* state *Y* iff *Z* is on every path from *Y* to the *Exit* state in the EFSM. State *Z post-dominates* transition *T* iff *Z* is on every path from state *Y* to the *Exit*

state through transition T. Let T_i and T_k be two outgoing transitions from states Y and Z, respectively. *T_i has control dependence on T_k* (or *T_k is control dependent on T_i*) iff:

1) Z does not post-dominate Y (Z is not on every path from Y to the *Exit state*)

2) Z post-dominates T_i (Z is on every path from Y to the *Exit* state through T_i)

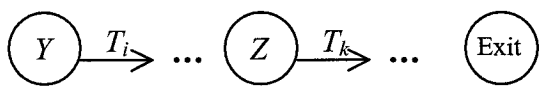


Figure 3.3 Illustration of the concept of control dependence

For example, consider states S1 and S2, and transitions T4 and T5 starting from states S1 and S2 respectively in the ATM EFSM of Figure 2.2.

1) S2 is not on every path from S1 to the *Exit* state (In fact, T3 takes the EFSM from S1 to the *Exit*). Hence, S2 does not post-dominate S1.

2) S2 is on every path from S1 to the *Exit* state through T4 (The only possible path is (T4, T9)), so S2 post-dominates T4.

Therefore, T4 has control dependence on T5 (T5 is control dependent on T4). All control dependencies in the ATM EFSM are shown in Table 3.2.

Table 3.2 Control Dependencies in the ATM EFSM

Transition	Control dependence	Post-dominance
T4	T5	S2 does not post-dominate S1 S2 post-dominates T5
	T6	S2 does not post-dominate S1 S2 post-dominates T6
	T7	S2 does not post-dominate S1 S2 post-dominates T7
	T9	S2 does not post-dominate S1 S2 post-dominates T9
T5	T8	S3 does not post-dominate S2 S3 post-dominates T5
T6	T8	S3 does not post-dominate S2

		S3 post-dominates T6
T7	T8	S3 does not post-dominate S2 S3 post-dominates T7

3.1.3 Static EFSM Dependence Graph (SDG)

Data and control dependencies in an EFSM model are graphically represented in a directed graph where nodes of this graph represent EFSM transitions and plain and dashed directed edges represent EFSM data and control dependencies, respectively. Such graph is called *Static EFSM Dependence Graph* or SDG [Vay02].

For a given EFSM model, the SDG for this EFSM is built by

1) Constructing the internal data structure of the EFSM model
2) Using the EFSM data structure to identify data dependencies in the EFSM
3) Identifying post dominance relationships between EFSM states in one hand and between EFSM states and transitions on the other hand then using this post dominance information to identify control dependencies in the EFSM
4) Constructing the SDG where edges in SDG are representing data and control dependencies identified in 2) and 3) (between EFSM transitions) by plain and dashed directed edges respectively, and nodes in SDG are representing these transitions

An example of an SDG is shown in Figure 3.4. This figure represents the SDG of he ATM EFSM given in Figure 2.2 and constructed using the data and control dependencies between transitions given in Table 3.1 and 3.2; respectively.

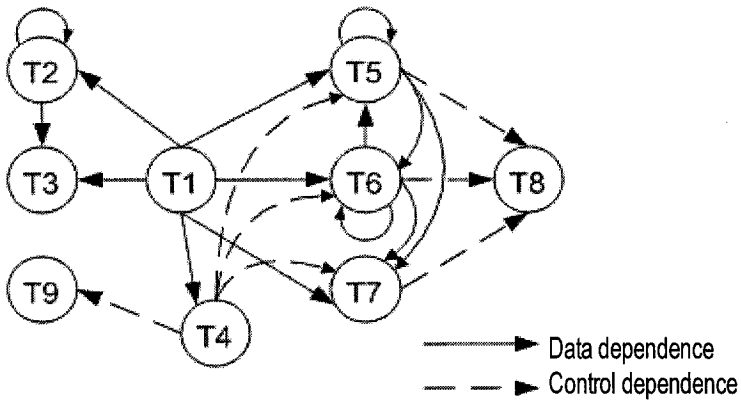


Figure 3.4 Static Dependence Graph of the ATM EFSM

The SDG generation is formalized by the SDG generation algorithm presented next.

3.2 SDG Generation Algorithm

The SDG generation algorithm takes the EFSM internal data structure as input and outputs the SDG data structure. The algorithm consists in two steps, data dependencies identification and control dependencies identification. To present the SDG generation algorithm, this section is organized as follows: sections 3.2.1 and 3.2.2 present the EFSM data structure and the SDG data structure used in the algorithm, respectively, then sections 3.2.3 and 3.2.4 show the data dependence and control dependence identification steps, respectively.

3.2.1 EFSM Internal Data Structure

The EFSM model is input to every algorithm presented in this thesis (SDG generation algorithm, and algorithms related to test suite generation). The EFSM internal data structure presented in this section will be referred to in the rest of the thesis.

The internal data structure for an EFSM model has been defined by Tuong Nguyen, member of the ASERT LAB of SITE of the University of Ottawa. Tuong implemented an

EFSM parser, and a program to classify variable occurrences in the EFSM and to automatically generate the EFSM internal data structure. Below, we review this data structure.

A variable v in the EFSM may occur in different transitions; moreover, a variable v may occur several times within one transition. The order of occurrence of variables in the EFSM is determined by going through the elements of a transition *from top to bottom, from right to left*. Hence, the first variable that occurs in the input of a transition t in the EFSM is assigned order of occurrence 1, the next variable occurring in this input statement has order of occurrence 2, etc... For example, variable b is used in order of occurrence 3 in T5 before it is defined in T5 in the ATM EFSM. A *Type* is associated with each occurrence of a variable v in the EFSM. Types of occurrence of a variable v in the EFSM are *def*, where v is defined in an input statement or in the left hand side of an assignment statement in terms of EFSM variables; *c-use*, where v occurs in the right hand side of an assignment statement or in a procedure call statement or in an output; or *p-use* where v occurs in an enabling predicate.

The internal data structure of an EFSM M is denoted by (*name*, *Start*, *Exit*, {*transition*}) where

name is the name of the EFSM,

Start is the start state of M,

Exit is the exit state of M and

transition is (*Label*, s_b, s_e, *Volist*, *listOfActions*)

Let t, T_i, T_j, T_k denote a transition in the EFSM or a node in the SDG.

For a transition t,

Label(t) denotes its label,

$s_b(t)$ denotes its beginning state by the index of the state,

$s_e(t)$ denotes its ending state by the index of the state,

Volist(t) denotes {(v, *Label*, *OType*, *OOrder*) | variable v occurs in transition with *Label* t as *OType* in order *OOrder*} where *OType* denotes the type of occurrence which is either a definition denoted *def*, or a *c*-use denoted *c-use*, or a *p*-use denoted *p-use*, and *OOrder* denotes the order of occurrence. Elements in *Volist*(t) set are referenced by an index between 0 and size of *Volist*(t) – 1, corresponding to their order in the set,

ListOfActions(t) denotes {(*Index*, *Id*, *AType*, *List*)} where for an action a

Index(a) denotes its index, *Id* denotes its id,

AType(a) denotes the action type, which is either an input denoted *INPUT*, an output denoted *OUTPUT*, an assignment statement denoted *ASSIGN*, a set statements denoted *SET*, a reset statement denoted *RESET*, a predicate denoted *PRED* or a procedure call denoted *PROC*,

List(a) denotes {(v, *Label*, *OType*, *OOrder*) | variable v occurs in transition with *Label* t in action a of t as *OType* in order *OOrder*}.

Yet, additional EFSM data structures need to be constructed prior to the SDG generation algorithm execution; more precisely, the sets defined next are formed from the EFSM internal data structure and later used in the SDG generation algorithm.

Let V denote $\{v \mid v$ is a variable in (EFSM) $M\}$,

$Def(v)$ denote $\{t \mid t$ is a transition in M which defines $v\}$,

$Use(v)$ denote $\{t \mid t$ is a transition in M which uses $v\}$,

$T_{in}(Z)$ denote $\{t \mid t$ is an incoming transition to state Z in $M\}$, for $Z \neq Start$,

$T_{out}(Z)$ denote $\{t \mid t$ is an outgoing transition from state Z in $M\}$, for $Z \neq Exit$,

$S_{in}(Z)$ denote $\{s \mid s = s_b(t)$ and $t \in T_{in}(Z)\}$, for $Z \neq Start$,

$S_{out}(Z)$ denote $\{s \mid s = s_e(t)$ and $t \in T_{out}(Z)\}$, for $Z \neq Exit$.

Sets V, $Def(v)$ and $Use(v)$, $S_{in}(Z)$ and $T_{in}(Z)$, $S_{out}(Z)$ and $T_{out}(Z)$, are generated by using algorithms V_Alg(M), Def-UseAlg(M,V), inAdjAlg(M) and outAdjAlg(M), respectively. These algorithms are given in APPENDIX D.1 of this thesis. Figure 3.5 below illustrates the data structures used in the SDG algorithm:

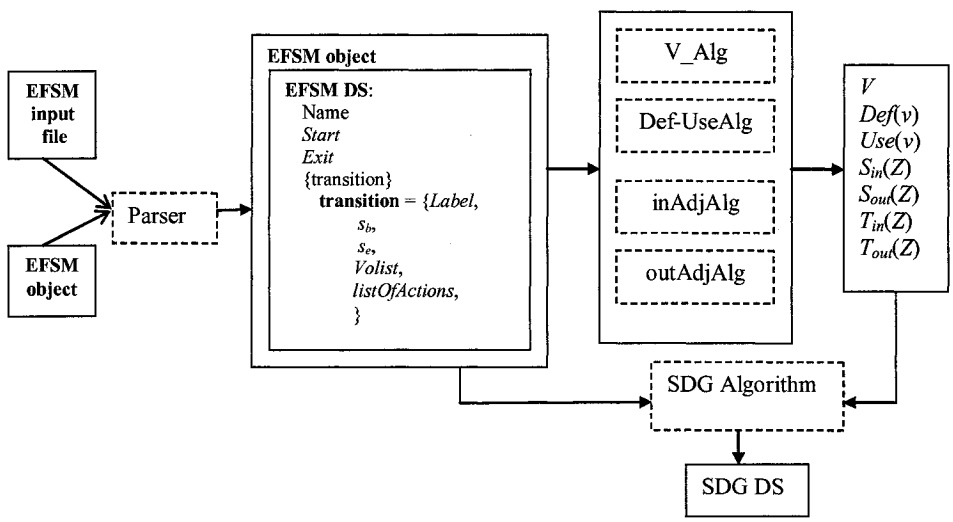


Figure 3.5 Data structures used in the SDG algorithm

3.2.2 SDG Data Structure

The SDG generation algorithm outputs an SDG (for the input EFSM *M*) as an adjacency matrix, $SDG[T_i,T_k]$, where nodes T_i and T_k in SDG *G* are transitions in EFSM *M*, and a directed edge from T_i to T_k in SDG *G* is non-empty element (*v*, *dependenceType*, v_{ik}).

v denotes variable in *V* that occurs in T_i as in $Volist(T_i)$ and in T_k as in $Volist(T_k)$,

dependenceType denotes the type of dependence which is either a control dependence denoted *cd* or a data dependence denoted *dd* or a du-pair based on a *c*-use denoted *dcu-pair* or a du-pair based on a *p*-use denoted *dpu-pair*,

v_{ik} denotes (*oo_def*, *oo_use*) such that variable *v* in *V* occurs in transition T_i in *M* as a definition denoted *def*, in the order of occurrence *oo_def*, and in T_k in *M* as a use denoted *use*, in the order of occurrence *oo_use*.

Let:

$PD(Z)$ denote $\{Y \mid Y$ is a state in $M \cap$ state Z post-dominates state $Y\}$,

$nPD(Z)$ denote $\{Y \mid Y$ is a state in $M \cap$ state Z is reachable from state Y and state Z does not post-dominate state $Y\}$,

$DD(T_k)$ denote $\{(T_i, v) \mid T_i$ and T_k are nodes in *SDG G* and T_k has an incoming data dependence edge, wrt variable *v*, from $T_i\}$,

$CD(T_k)$ denote $\{T_i \mid T_i$ and T_k are nodes in the *SDG* and T_k has an incoming control dependence edge from $T_i\}$.

3.2.3 Data Dependence Identification

Data dependencies are identified wrt to each occurrence of variable v defined in T_i and used in T_k in the given EFSM. First, du-pairs within a transition T_i in M wrt to variable v are identified. Suppose that T_i defines v with order of occurrence *oo_def* and uses v with order of occurrence *oo_use*. There is a du-pair in T_i wrt v iff *oo_def* < *oo_use*, and there is no definition of v in *oo* in T_i such that *oo_def* < *oo* < *oo_use*. Figure 3.6 illustrates such a du-pair.

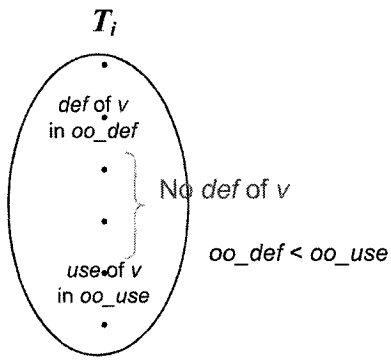


Figure 3.6 Illustration of a du-pair in a transition T_i in M wrt to variable v

If *use* is a *c-use* then $SDG[T_i, T_i] = (v, dcu\text{-}pair, v_{ii})$ where v_{ii} is (*oo_def*, *oo_use*). If *use* is a *p-use* then $SDG[T_i, T_i] = (v, dpu\text{-}pair, v_{ii})$ where v_{ii} is (*oo_def*, *oo_use*).

Next, data dependence from transition T_i to transition T_k in M wrt to v is identified. Let T_i define v in order of occurrence *oo_def* and T_k use v in order of occurrence *oo_use*. There is a data dependence from transition T_i to transition T_k iff:

1) *oo_def* is the last *def* of v in T_i, i.e., the last occurrence of v as a *def* in T_i;

2) *oo_use* is a use of v in T_k before a def of v in T_k;

3) there is a def-clear path wrt v from T_i to T_k.

Such data dependence is illustrated in Figure 3.7 when $T_i \neq T_k$ and in Figure 3.8 when $T_i = T_k$ (case of a self loop data dependence).

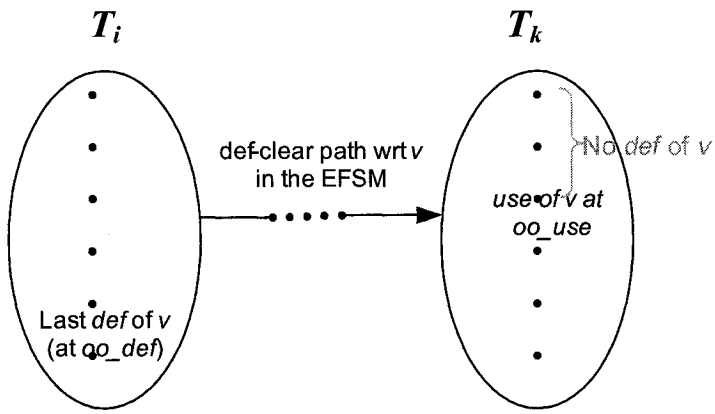


Figure 3.7 Illustration of data dependence from T_i to T_k in M wrt to v, $T_i \neq T_k$

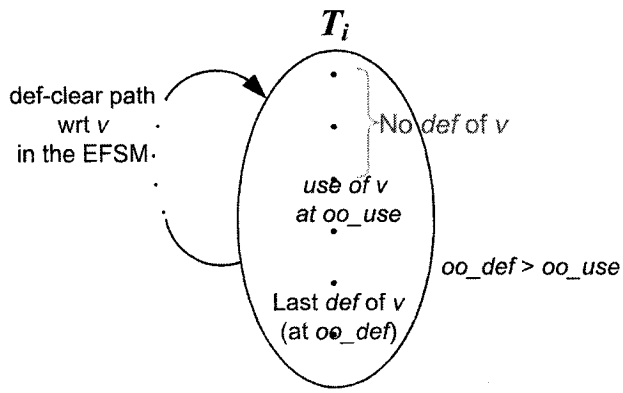


Figure 3.8 Illustration of data dependence from T_i to T_k in M wrt to v, $T_i = T_k$

If *use* is a *c-use* then $SDG[T_i, T_k] = (v, dd, v_{ik})$ where v_{ik} is (*oo_def*, *oo_use*).

If *use* is a *p-use* then $SDG[T_i, T_k] = (v, dd, v_{ik})$ where v_{ik} is (*oo_def*, *oo_use*).

3.2.4 Control Dependence Identification

Control dependence from transition T_i to transition T_k in M, where $s_b(T_i) = Y$ and $s_b(T_k) = Z$, is defined such that Z does not post-dominate Y and Z post-dominates T_i. The following holds from the definition of post-dominance:

- The exit state *Exit* of an EFSM post-dominates all states

- A state post-dominates itself in an EFSM

- If Z post-dominates Y, then Z post-dominates all states in $S_{out}(Y)$

- "Z post-dominates T_i" $\Leftrightarrow$"Z is on every path from Y to the *Exit* state through T_i" $\Leftrightarrow$ "Z is on every path from $s_e(T_i)$ to the *Exit* state" $\Leftrightarrow$ "Z post-dominates $s_e(T_i)$.

Further, from the definition of control dependence, if Z does not post-dominate Y and Z post-dominates T_i, then there is a control dependence from T_i to every outgoing transition from Z.

3.2.5 SDG Generation Algorithm

This algorithm generates the Static Dependence Graph for an EFSM.

Input: M, the data structure for the EFSM,

Output: the adjacency matrix of the SDG for M.

Algorithm Steps

The SDG data structure is initialized as follows:

- $SDG[T_i, T_k] \leftarrow$ empty for all transitions T_i and T_k in M

- $DD(T_k) \leftarrow$ empty and $CD(T_k) \leftarrow$ empty for all transitions T_k in M.

- Unmark all states Y in M.

// identify data dependencies//

For each variable v in V do

for each transition T_i in $Def(v)$ do

for each (v, *Label*, *def*, *oo_def*) in $Volist(T_i)$ do

for each transition T_k in $Use(v)$ do

for each (v, *Label*, *use*, *oo_use*) in $Volist(T_k)$ do

//use is a c-use or a p-use //

//Identify data dependencies from transition T_i to transition T_k//

1) If (($T_i = T_k$ and *oo_def* > *oo_use*) or ($T_i \neq T_k$)) and ((*v*, *Label*, *def*, *oo_def*) is the last *def* of *v* in *Volist*(T_i) and there is no (*v*, *Label*, *def*, *oo*) in *Volist*(T_k) where *oo* < *oo_use*)

then

if T_i reaches T_k

then

search a def-clear path from T_i to T_k wrt *v*

if such a def-clear path exists

then

if *use* = *c-use*

then

$v_{ik} \leftarrow$ (*oo_def*, *oo_use*)

(*dependenceType*, v_{ik}) $\leftarrow$ (*dd*, v_{ik})

else *// use = p-use//*

$v_{ik} \leftarrow$ (*oo_def*, *oo_use*)

(*dependenceType*, v_{ik}) $\leftarrow$ (*dd*, v_{ik})

else (*dependenceType*, v_{ik}) $\leftarrow$ *Nil*

else (*dependenceType*, v_{ik}) $\leftarrow$ *Nil*

//Identify du-pairs within a transition T_i//

2) else if $T_i = T_k$ and $oo\_def < oo\_use$ and there is no

(v,*Label*,*def*, *oo*) in $Volist(T_i)$ such that $oo\_def < oo <$

oo_use

then

if *use* = *c-use*

then

$v_{ii} \leftarrow (oo\_def, oo\_use)$

$(dependenceType, v_{ik}) \leftarrow (dcu\text{-}pair, v_{ii})$

else *//use = p-use//*

$v_{ii} \leftarrow (oo\_def, oo\_use)$

$(dependenceType, v_{ik}) \leftarrow (dpu\text{-}pair, v_{ii})$

3) else $(dependenceType, v_{ik}) \leftarrow Nil$

if $(dependenceType, v_{ik}) \neq Nil$ *//∃ a def-clear path from* T_i *to* T_k *wrt* v*//*

then

$SDG[T_i, T_k] \leftarrow SDG[T_i, T_k] \cup \{(v, dependenceType, v_{ik})\}$

if *dependenceType* = *dd*

then

if $(T_i, v) \notin DD(T_k)$

then

$DD(T_k) \leftarrow DD(T_k) \cup \{(T_i, v)\}$

// T_i *and* T_k *are nodes in the SDG* $DD(T_k)$ *//*

// identify control dependencies//

// Find, for each state in M, the set of states that this state post-dominates//

1) For each state Z, except *Exit*, in M do

   //Find all states that Z post-dominates //

   $PD(Z) \leftarrow \{Z\}$

   $nPD(Z) \leftarrow \{\}$

   for each state Y that reaches Z do

      if state Z post-dominates state Y

         then $PD(Z) \leftarrow PD(Z) \cup \{Y$ and each state $X \in S_{out}(Y), X \neq Z\}$

      else $nPD(Z) \leftarrow nPD(Z) \cup \{Y\}$

2) For each state Z, except *Exit*, in M do

   //Find all transitions that Z post-dominates//

   for each state Y in $nPD(Z)$ do

      for each transition T_i in $T_{out}(Y)$ do

         if $s_e(T_i)$ is in $PD(Z)$

            then *//Z post-dominates* T_i*//*

               for each transition T_k in $T_{out}(Z)$

                  // transition T_i *has control dependence on each outgoing//*

                  // transition T_k *from Z//*

                  $SDG[T_i, T_k] \leftarrow SDG[T_i, T_k] \cup \{(Nil, cd, Nil)\}$

                  $CD(T_k) \leftarrow CD(T_k) \cup \{T_i\}$

Chapter 4

Test Generation Approach Based on Static Dependence Analysis

In this thesis, we propose a novel method to automatically generate an efficient test suite wrt individual requirements of an EFSM model by using dependence analysis[4]. A test suite is generated for a requirement under test by capturing possible interactions affecting the requirement. This problem is solved based on the assumption that *interactions* between EFSM transitions ("active" elements of the EFSM) are represented as control and data dependencies in the Static Dependence Graph (SDG) (of the EFSM) [Vay02]. Data and control dependencies (or patterns of interactions) affecting a transition under test in the EFSM are captured in subgraphs of the SDG, referred to as Static Interaction Patterns.

The motivation behind the proposed test generation approach is presented in Section 1.2: authors in [Vay02] established that *for any EFSM model, the number of static interaction patterns is bounded for a requirement under test*; hence we propose a method to automatically generate the set of static interaction patterns for a requirement under test.

This chapter is organized as follows: Section 4.1 defines a Static Interaction Pattern or SIP, Section 4.2 presents the highlights of the proposed test generation method, and Section 4.3 presents the algorithms we developed to formalize this method.

[4] We assume that tests are generated as sequences of EFSM transitions and input and output parameter values are ignored.

4.1 Static Interaction Patterns

A Static Interaction Pattern or SIP wrt a TUT in the EFSM is a subgraph of the EFSM's SDG such that each data and control dependence edge in SIP affects the traversal of the TUT.

The notion of SIP was first introduced in [Vay02] in the following context: during test execution, different elements of the EFSM model of the system requirements interact with each other and some of them may affect the TUT. Capturing only those interactions that influence the TUT yields a pattern of interactions wrt the TUT exercised by a test sequence. For example, let T5 be the TUT in the ATM EFSM shown in Figure 2.2. The SIP shown in Figure 4.1 below captures a pattern of interactions wrt TUT = T5 exercised by the test sequence *test*_1 = (T1, T4, T6, T8, T6, T8, T5, T8, T9) which corresponds to the scenario where user enters a valid pin, performs two deposit transactions and a withdraw transaction. (This SIP is obtained from *test*_1 by applying the test reduction method steps of [Vay02] reviewed in Section 1.2 of this thesis).

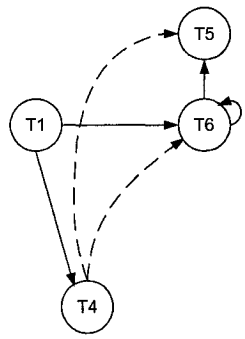


Figure 4.1 Example of a SIP

Two tests in a test suite are considered to be equivalent wrt the TUT if both exhibit the same SIP. There is no need to test the system with tests that exercise the same SIP wrt the TUT hence; one of them can be discarded from the test suite. For example, consider

test sequence *test_2* = (T1, T2, T2, T4, T6, T8, T6, T8, T5, T8, T9) wrt TUT = T5. Like *test_1*, *test_2* exhibits the SIP shown in Figure 4.1, therefore *test_1* and *test_2* are equivalent wrt T5 in the ATM EFSM.

The number of SIPs wrt each TUT in the EFSM model is bounded [Vay02]. To illustrate this observation, consider requirement *r2*[5], represented by transition T5 in the ATM EFSM shown in Figure 2.2. Given 6 test suites wrt T5, authors in [Vay02] applied the test reduction technique they propose on these test suites to obtain reduced size test suites where each test is represented by a SIP. This test reduction technique is summarized in Section 1.2 in this thesis, and formalized by former ASERT LAB member Panitee Ritthiruangdech in [Rit04]. Table 4.1 shows the size of the original test suite, and the number of SIPs obtained after reducing this test suite (*n* corresponds to the maximum number of traversals of an EFSM transition in a test sequence). Table 4.1 reveals that the number of test sequences in the reduced test suite is bounded below by the number of possible SIPs wrt TUT, in fact, there are at most 14 SIPs wrt T5.

Table 4.1 Test Suite Reduction for TUT = T5 in the ATM EFSM

n	**Size of original test suite**	**[6]Size of reduced test suite (# of static interaction patterns)**
2	18	3
3	100	7
4	450	14
5	1,806	14
6	6,7602	14
7	24,300	14

Experiments conducted in [Vay02] and [Rit04] have identified the set of data and control dependencies that affect TUT = T5 in the ATM EFSM presented in Figure 2.2.

[5] Requirement r2 says that after entering a correct pin in less than four attempts, user selects withdrawal function. The system adjusts the balance, and displays a menu with withdrawal, deposit, balance inquiry, and exit functions

[6] Note that the last three numbers in this column are corrected by us and confirmed by the authors of [Vay02]

Figure 4.2 below highlights this set of data and control dependence edges and their corresponding source and destination nodes in the SDG of the ATM ESFM. Note that bold nodes and edges correspond to the overlapping of all possible SIPs wrt TUT = T5. As shown in Table 4.1, there are 14 static interaction patterns wrt T5 in the ATM EFSM. These 14 SIPs are illustrated in APPENDIX A.5.

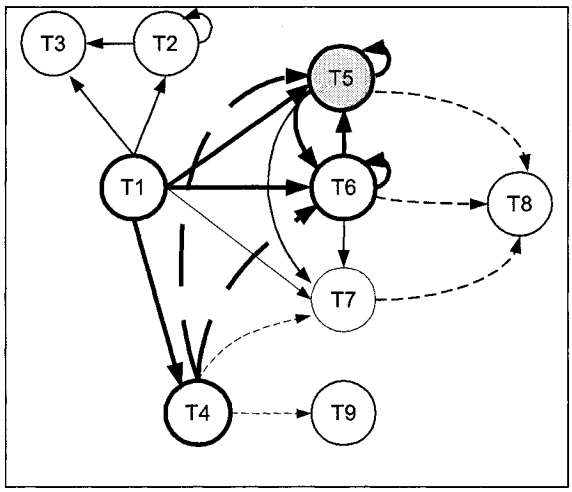


Figure 4.2 Overlapping of all possible SIPs wrt T5 on the SDG of ATM EFSM

4.2 Overview of the Proposed Test Generation Method

The proposed method uses an EFSM model representing a set of requirements R, and the SDG of this EFSM to construct a test suite for each requirement r in R. The approach consists of two phases: S_r, the set of SIPs for a requirement r is first generated, then, each SIP in S_r is mapped to a test sequence in the EFSM to obtain TS_r, a test suite for requirement r.

Constructing S_r for each requirement r in R is achieved by first adapting the df-chains coverage criterion reviewed in Section 2.3.2.2 of this thesis to generate data dependencies affecting a TUT. A set of df-chains along with corresponding activating paths is obtained wrt to each TUT in the EFSM. Activating paths of these df-chains are considered as test

sequences in the EFSM. The method for generating a SIP from a test sequence proposed in [Rit04] is used to derive a SIP from each activating path. Section 4.2.1 explains the use of the df-chains criterion to solve the S_r generation problem, and illustrates the solution with an example.

4.2.1 Construction of S_r, the Set of SIPs for Requirement r

We have established earlier that constructing the set of SIPs for a TUT corresponds to identifying data and control dependencies affecting the TUT. We adapt the df-chains coverage criterion proposed in [UY93] here to construct S_r. In fact, authors in [UY93] defined the notion of data flow chain or df-chain based on the notion of *affect*, where a df-chain is a sequence of linked du-pairs that start by an input and end by the use of a variable v in TUT, such that this input affects this use of v in TUT. Recall that a *df-chain* is an ordered sequence $(d_{n1}^{x1}, u_{n2}^{x1}, d_{n2}^{x2}, u_{n3}^{x2}, \ldots, d_{nm}^{xm}, u_{nm+1}^{xm})$ of du-pairs $(d_{n1}^{x1}, u_{n2}^{x1})$, $(d_{n2}^{x2}, u_{n3}^{x2}), \ldots, (d_{nm}^{xm}, u_{nm+1}^{xm})$, such that $m \geq 1$ and u_{nm+1}^{xm} is affected by d_{n1}^{x1} [UY93]. We call a pair $(u_{nm}^{xm}, d_{nm}^{ym})$ in a df-chain where variable x in node n_m is used to define variable y in node n_m a *link.*

For test generation purposes, we define a *df-chain* in the EFSM wrt a variable v used before it is defined in TUT in the EFSM. In fact, the motivation behind the test generation method proposed in this thesis is to capture data and control flow interactions between different elements of the EFSM that affect the TUT. Capturing only those interactions that influence the TUT yields a pattern of interactions wrt the TUT. In the proposed method, we use df-chains to trace the data flow interactions (between EFSM transitions) that end at a TUT. More precisely, we trace data flow interactions from a du-

pair (d^{v}_{Ti}, u^{v}_{Tk}) where variable v is *input* in transition T_i; to a du-pair (d^{w}_{Tl}, u^{w}_{TUT}) where variable w is used in TUT before it is defined in TUT; and u^{w}_{TUT} is affected by d^{v}_{Ti} . Thus, we define the following properties:

1) A df-chain wrt a variable v used in the TUT starts with an input and ends with a *c*-use or a *p*-use of a variable v in the TUT before v is defined in the TUT, and this input affects this use of v in the TUT

2) Links in a df-chain are distinct.

For example, let V_r be the set of variables used in a TUT before they are defined in the TUT. Figure 4.3 illustrates a df-chain *df* wrt variable v in the TUT such that $df = (d^{x}_{ti}, u^{x}_{ti+1}, d^{y}_{ti+1}, u^{y}_{ti+2}, d^{z}_{ti+2}, u^{z}_{ti+3}, d^{v}_{ti+3}, u^{v}_{TUT})$:

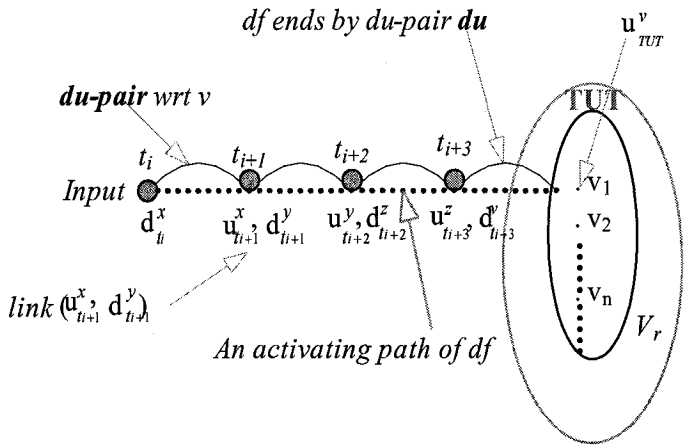


Figure 4.3 Graphical representation of a df-chain in the EFSM

This df-chain starts with du-pair (d^{x}_{ti}, u^{x}_{ti+1}) wrt variable x such that d^{x}_{ti} is an input, and ends with du-pair (d^{v}_{ti+3}, u^{v}_{TUT}) wrt variable v such that u^{v}_{TUT} is a *c*-use or a *p*-use of v in the TUT, and there is no definition of v before this use of v in the TUT. Hence, u^{v}_{TUT} is affected by d^{x}_{ti}. Path (t_i, ...,t_{i+1}, ...,t_{i+2}, ...,t_{+3}, *TUT*) is an activating path of the *df,* where each of (t_i, ...,t_{i+1}), (t_{i+1}...,t_{i+2}), (t_{i+2},...,t_{i+3}) and (t_{i+3},... *TUT*) are def-clear paths wrt variables x,

y, z and v respectively. Since links in a df-chain are unique, the finiteness of an actual activating path of a df-chain is guaranteed.

Accordingly, generating the set of different data dependencies affecting a TUT, for each TUT in the EFSM, is achieved by generating the set of df-chains wrt each TUT, more precisely wrt each variable used before it is defined in this TUT. The EFSM and the SDG internal data structures presented in sections 3.2.1 and 3.2.2 respectively are used to construct the set of df-chains in the EFSM. A df-chain *df* wrt variable v in TUT is constructed such that it starts by an input from the set *inputs*(*M*) and consists of a sequence of linked du-pairs from *SDM*, the SDG matrix with type data dependence, dcu-pairs and dpu-pairs. Du-pairs in *df* are linked by links in the ESFM from the set *links*(*M*).

In addition to the df-chain related assumptions stated above, we assume that self loop data dependencies in the *SDM* are ignored when constructing a df-chain, and combinations of these self loop data dependencies are later added to SIPs. Recall that, in order to ensure the finiteness of the number of df-chains for a requirement under test, we assumed that links are distinct in a df-chain. For this assumption to hold, self loop data dependencies must be ignored when generating a df-chain. For instance, consider df-chain $df = (d_1^b, u_5^b)(d_5^b, u_6^b)(d_6^b, u_5^b)$, with activating path (T1, T4, T5, T8, T6, T8, T5) wrt TUT= T5 in the ATM EFSM shown in Figure 2.2. If T5 is traversed twice in a row in the EFSM, then $df = (d_1^b, u_5^b)(d_5^b, u_6^b)(d_6^b, u_5^b)(d_5^b, u_5^b)$ (a self loop data dependence on T5 is appended to *df*) and link (u_5^b, d_5^b) occurs twice in *df*.

Let M be an EFSM model and R be the set of requirements represented by M. The steps for the generation of S_r, the set of SIPs for a requirement r, are presented next along

with an example for requirement *r2* represented by TUT=T5 in the ATM EFSM shown in Figure 2.2.

In the first step, we generate the set *self-loops*(*SDM*) of self loops in *SDM*, the set *inputs*(*M*) of inputs in the EFSM, and the set *links*(M) of links in the EFSM by using *SDM* and *M*.

In the second step, we generate set V_r, the set of variables used before they are defined in the TUT representing *r* in the EFSM *M*, $\forall\, r \in R$.

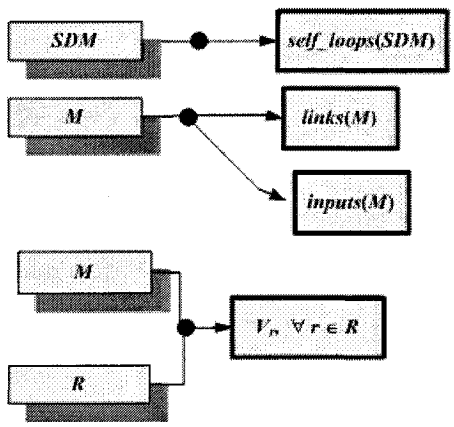


Figure 4.4 Steps 1 and 2 of S_r generation algorithm

In the third step, we generate,

- $\forall\, r \in R$, the set $DF_{AP}(r)$ of all df-chains in the EFSM ending with the uses of variables of V_r, and
- $\forall$ df-chain $df \in DF_{AP}(r)$, the set AP_{df} of activating paths for *df*, as follows:
  - Use the sets, *inputs*(*M*) and *links*(*M*) defined in the internal data structure of the EFSM and du-pairs in *SDM*, the SDG matrix, to generate the set *DF*(*v*) of all df-chains starting with an input and ending with a use of a variable *v*,$\forall$ *v* in V_r
  - For each df-chain *df* in *DF*(*v*),

generate the set AP_{df} of all possible activating paths for *df*. Paths in AP_{df} that do not start by an outgoing transition from the start state of the EFSM are identified, and prefixes[7] are appended to these paths. Pair each *df* with AP_{df}, its set of activating paths, and add this pair to $DF_{AP}(r)$

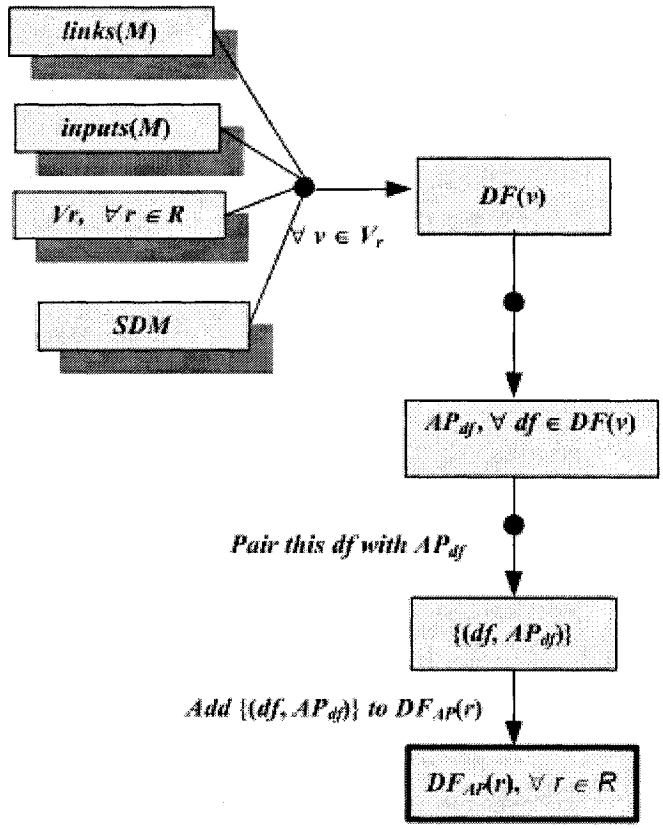


Figure 4.5 Step 3: $DF_{AP}(r)$ generation of S_r generation algorithm

For example, Tables 4.2 and 4.3 show the set of inputs in the ATM EFSM, in the format (*v*, *T*, *oo_def*) where *oo_def* is the order of occurrence of variable *v* in transition *T* and the set of links in the format (u(*v*, *T*, *oo_use*), d(*v*, *T*, *oo_def*)).

[7] Let T_i be an outgoing transition from the start state in the EFSM. A prefix of an activating path that starts with transition T_j is a path that starts at T_i and end at T_j in the EFSM.

Table 4.2 The set *inputs(M)* of inputs in the ATM EFSM

#	***Inputs(M)***
1	(*b*, T1, 1)
2	(*pin*, T1, 2)
3	(*attempts*, T1, 3)
4	(*p*, T2, 1)
5	(*p*, T3, 1)
6	(*p*, T4, 1)
7	(*w*, T5, 1)
8	(*d*, T6, 1)

Table 4.3 The set *links(M)* of links in the ATM EFSM

#	***links(M)***
1	(u(*attempts*, T2, 5), d(*attempts*, T2, 6))
2	(u(*w*, T5, 2), d(*b*, T5, 4))
3	(u(*b*, T5, 3), d(*b*, T5, 4))
4	(u(*d*, T6, 2), d(*d*, T6, 4))
5	(u(*b*, T6, 3), d(*b*, T6, 4))

DF(*v*) is generated wrt *v* = (*b*, 3) in TUT = T5 in the ATM EFSM by using *inputs*(*M*), *links*(*M*) and *SDM*. *DF*(*v*) consists of 6 df-chains that are listed in Appendix A.4. Table 4.4 shows df-chain # 3 among these df-chains along with its activating paths. This *df* consists of two du-pairs. The first du-pair is (d(*b*,T1,1), u(*b*,T6,3)) which has 4 def-clear paths wrt *b*: {(T1, T4, T6), (T1, T4, T7, T8, T6), (T1, T2, T4, T6), (T1, T2, T4, T7, T8, T6)}. The second du-pair of *df* is (d(*b*,T6,4), u(*b*,T5,3)) which has 2 def-clear paths wrt *b*: {(T6, T8, T5), (T6, T8, T7, T8, T5)}. Note that, since activating paths of *df* start by T1 (an outgoing transition from the start state of the ATM EFSM), no prefix is appended to the activating paths of *df*. AP_{df} is obtained by forming combinations of the def-clear paths of the first du-pair followed by those of the second du-pair. Also, note that 2 EFSM loops are encountered when forming the set DCP_{df} of def-clear paths for du-pairs of *df*. These

loops are (T2) and (T8, T7, T8) which are traversed zero and 1 times, as stated in the assumptions related to df-chains.

Table 4.4 df-chain wrt TUT=T5 in the ATM EFSM

df	AP_{df}
{d(*b*,T1,1), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T5,3)}	(T1,T4,T6,T8,T5) (T1,T4,T6,T8,T7,T8,T5) (T1,T4,T7,T8,T6,T8,T5) (T1,T4,T7,T8,T6,T8,T7,T8,T5) (T1,T2,T4,T6,T8,T5) (T1,T2,T4,T6,T8,T7,T8,T5) (T1,T2,T4,T7,T8,T6,T8,T5) (T1,T2,T4,T7,T8,T6,T8,T7,T8,T5)

In the fourth step, we construct $G[df, ap] = (N, E)$, $\forall\, r \in R$, $\forall$ pair $(df, AP_{df}) \in DF_{AP}(r)$ to obtain an initial set of static interaction patterns that we refer to as *elementary static interaction patterns* or SIP_e (S_{r_elem} is the set of these elementary interaction patterns).

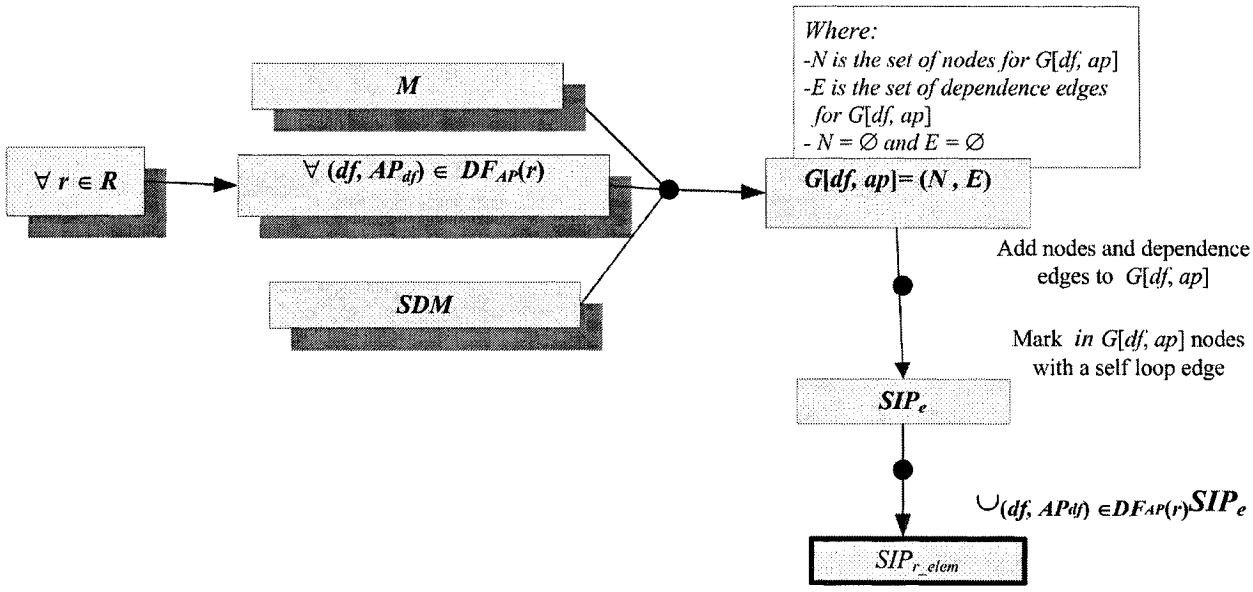


Figure 4.6 Step 4: SIP_{r_elem} generation of S_r generation algorithm

A SIP_e is obtained from *G[df, ap]* by using algorithm *SIP generation for a given test sequence* method formalized in [Rit04] as follows:

1) Control and data dependencies that are encountered during the traversal of *ap* are identified and represented in *G[df, ap]* (a sub-graph of a static dependence graph (SDG))
2) Dependencies affecting the TUT are identified by traversing *G[df, ap]* backwards from the TUT and marking all traversed nodes and edges
3) All unmarked nodes and edges are removed from *G[df, ap]* to obtain SIP_e

For example, take the pair (*df, ap*) = ({d(*b*,T1,1), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T5,3)}, (T1,T4,T6,T8,T7,T8,T5)) in Table 4.4. Figure 4.7 show *G[df, ap]* after steps 2) and 3).

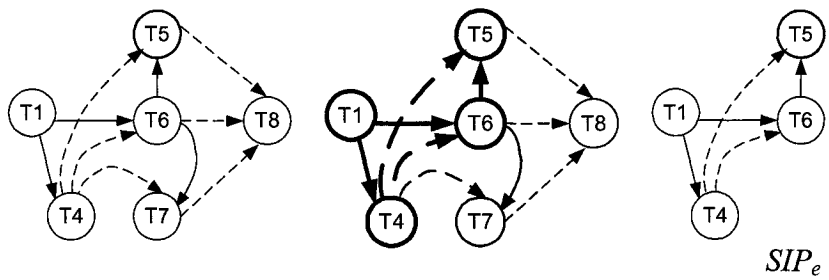


Figure 4.7 Construction of SIP_e from G[df, ap]

In the fifth step, $\forall\ SIP_e \in SIP_{r\_elem}$, $\forall\ r \in R$, to obtain S_r, we add combinations of self loop data dependencies to nodes of SIP_e.

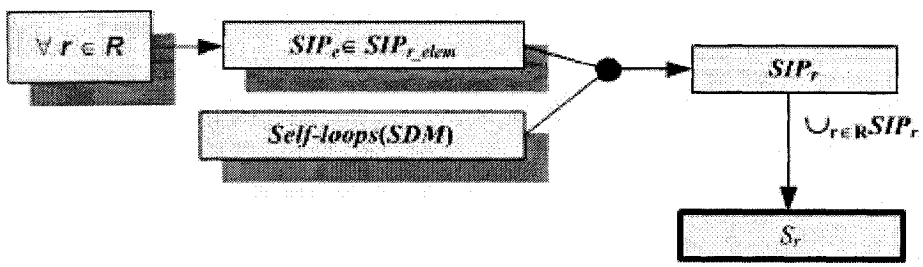
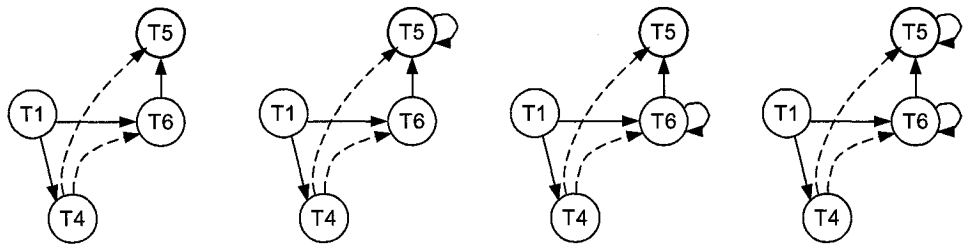


Figure 4.8 Step 5: SIP_r generation of S_r algorithm

For instance, take the elementary SIP SIP_e shown in Figure 4.7. Nodes T5 and T6 of SIP_e have a self loop data dependence edge in the SDG of the ATM ESFM. Combinations of these edges are added to SIP_e to obtain 3 additional SIPs wrt T5. Figure 4.9 shows SIP_e on the left hand side, and the 3 SIPs derived from SIP_e on the right hand side. These 4 SIPs of Figure 4.9 are added to S_r for TUT=T5.



SIP_e

Figure 4.9 Generation of SIPs from a given SIP_e

4.2.2 Construction of TS_r, the Test Suite for Requirement *r*

TS_r is constructed for each requirement r in R by mapping each SIP in S_r to a complete test sequence in the EFSM (A *complete test sequence in the EFSM* is an EFSM path whose first node is *Start*, the start state of the EFSM, and whose last node is *Exit*, the exit state of the EFSM). This test sequence is obtained for a SIP as follows: for each SIP in S_r, take ap_{df}, the activating path of *df*, used to generate the SIP. ap_{df} is a sequence of EFSM transitions in the EFSM that starts at the start state of the EFSM and ends at transition *r*. For ap_{df} to be a complete test sequence, a suffix (sequence of EFSM transitions from *r* to the exit state of the EFSM) is appended to ap_{df} to obtain *ts*, a test sequence that is added to TS_r.

4.3 S_r Generation Algorithm

Input:

- the SDG adjacency matrix SDM,
- the EFSM data structure *M*, and
- the set *R* of requirements for the EFSM

Output: S_r, the set of static interaction patterns SIP_r, $\forall\ r \in R$.

Data Structure[OCS]

Let t_i and t_j denote *SDM* nodes (i.e., *EFSM* transitions),

self-loops(SDM) denote a subset set of *SDM* nodes, each having a self loop data dependence,

V_r denote the set of variables used before they are defined in transition *r*,

(d(v, t_i, *oo_def*), u(v, t_j, *oo_use*)) denote a du-pair wrt v,

(u(v_i, t, oo_{vi}), d(v_j, t, oo_{vj})) denote a link where v_i is used to define v_j in t,

links(M) denote the set of all links of *M*,

inputs(M) denote the set of all inputs in *M*. An input is d(v, t, *oo_v*), such that v is defined in t with *oo_v* in an input statement or in an assignment statement where v is assigned a constant,

df denote a df-chain wrt v where:

- *df* starts with an input in *inputs*(*M*) and ends with a du-pair denoted (d(v, t_i, *oo_def*), u(v, t_j, *oo_use*)) such that $t_j = r$ and $v \in V_r$
- links in *df* are distinct,

ap denote an activating path (a sequence of EFSM transitions) for a df-chain *df*,

AP_{df} denote the set of all activating paths for the df-chain *df* in the EFSM,

DF(v) denote the set of all df-chains ending with the use of variable $v \in V_r$,

$DF_{AP}(r)$ denote the set of all df-chains ending with the use of a variable v, $\forall\ v \in V_r$,

together with their respective sets of AP_{df},

dcp denote a def-clear path (a sequence of EFSM transitions) for a du-pair,

DCP_{du} denote the set of all possible def-clear paths *dcp* for the du-pair *du*,

DCP_{df} denote the set of all possible def-clear paths *dcp* for the df-chain *df*,

AP_{SIP} denote the set of all activating paths that lead to the generation of *SIP*.

Algorithm Steps

1. Generate sets *self-loops*(*SDM*), *inputs*(*M*) and *links*(*M*) by using *SDM* and *M*.
2. Generate set V_r, $r \in R$ by using *M* and *R*.
3. Generate $DF_{AP}(r)$, $\forall\ r \in R$ as follows:

```
for each r in R
    DF_AP(r) ← ∅
    for each variable v in V_r
        //form the set DF(v) of all df-chains wrt v ending with a use of v in V_r//
        form DF(v)
        //form the set AP_df of all possible activating paths ap for df//
        for each df-chain df in DF(v) do
            DCP_df ← ∅
            for each du-pair du in df do
                //Let dcp denote a def-clear-path and let DCP_du denote //
                //the set of all possible def-clear paths for du-pair du//
```

form the set DCP_{du} of all possible def-clear-paths *dcp* for *du*, where loops encountered when forming *dcp* are traversed 0 or 1 times

$DCP_{df} \leftarrow DCP_{df} \cup \{(du, DCP_{du})\}$

endfor

//call the FormAP$_{df}$(df, DCP$_{df}$, AP$_{df}$) procedure to //

//form the set AP$_{df}$ of all activating paths for df //

FormAP$_{df}$(df, DCP$_{df}$, AP$_{df}$)

//Pair df with AP$_{df}$//

$DF_{AP}(r) \leftarrow DF_{AP}(r) \cup \{(df, AP_{df})\}$

//endfor r, v, df here//

//Construct the graph G[df, ap] representing each generated df-chain df and //

// each activating path ap in AP$_{df}$//

4. for each *r* in *R* do

$S_{r\_elem} \leftarrow \varnothing$

for each pair (df, AP_{df}) in $DF_{AP}(r)$ do

for each *ap* in AP_{df} do

//let G[df, ap] = (N, E) where N is the set of nodes of G, and E is //

//the set of dependence edges of G//

//call the ConstructGraph(df, ap, G[df, ap]) to generate G[df, ap]//

ConstructGraph(df, ap, G[df, ap])

// G[df, ap] can have a self loop edge iff the prefix of ap includes the //

//def-clear path of a self loop data dependence//

mark nodes with self loop edges in $G[df, ap]$

$S_{r\_elem} \leftarrow S_{r\_elem} \cup \{G[df, ap]\}$

//add ap to the set AP_{SIP} of activating paths used to generate //

// this elementary SIP//

$AP_{SIP} \leftarrow AP_{SIP} \cup \{ap\}$

//endfor r, (df, AP_{df}), ap here//

5. Add combinations of self loop data dependencies to each SIP_e in S_{r_elem}, as follows:

for each $r \in R$

initialize S_r by S_{r_elem}

for each $SIP_e \in S_{r\_elem}$

use nodes in {{unmarked nodes of SIP_e} $\cap$ {*self-loops*(*SDM*)}} to generate additional *SIPs* SIP_r from SIP_e by adding all combination of self loop edges to these nodes in SIP_e

generate AP_{SIPr} for each newly obtained SIP_r by using AP_{SIPe} such that the EFSM path corresponding to the combination of self loop edges added to SIP_r, is inserted in paths of AP_{SIPe}

add each newly obtained SIP_r to S_r

//end of step 5//

// details of the procedures //

FormAP$_{df}$(df, DCP$_{du}$, AP$_{df}$)

form the set AP_{df} of all activating paths for *df* using combinations of def-clear paths of du-pairs of *df* as follows:

//Let du_0 denote the first du-pair in df (i.e., the def of this //

//du-pair is an input //

//Let DCP_{du0} denote the set of all def-clear paths for du_0//

$AP_{df} \leftarrow DCP_{du0}$

for each du-pair *du* in *df*, except for du_0, do

let *oldSize* denote the size of AP_{df}

for each def-clear path *dcp* in DCP_{du} for this *du* do

remove from this *dcp* its first node

endfor

$j \leftarrow 0$

while $j <$ *oldSize* do

tmp $\leftarrow$ *dcp* with index *j* in AP_{df}

for each def-clear path *dcp* in DCP_{du} for *du* do

if this *dcp* has index 0 in DCP_{du}

then

append the def-clear path with index *j* in AP_{df}, to this *dcp*

else *//Let p be a path //*

append tmp to this *dcp*

$AP_{df} \leftarrow AP_{df} \cup \{dcp\}$

$j \leftarrow j + 1$

endwhile

endfor *//All possible activating paths for df are now generated//*

for each *ap* in AP_{df} do

//Let t_f denote the first transition of this activating path ap, and //

//let T_s denote the set of outgoing transitions from the start state in //

//this EFSM//

if $t_f \notin T_s$

then

remove this *ap* from AP_{df}

//Find all paths from start state in the EFSM to t_f//

for each transition *t* in T_s do

form the set *pre*(*ap*) of all *EFSM* paths from *t* to t_f

for each path *p* in *pre*(*ap*) do

append ap to p to obtain ap_{aug}

$AP_{df} \leftarrow AP_{df} \cup \{ap_{aug}\}$

//end of procedure//

ConstructGraph(df, ap, G[df, ap])

// G[df, ap] = (N, E) //

construct *G*[*df*, *ap*] by adding nodes and dependence edges as follows:

form the set of nodes *N* of *G*[*df*, *ap*] by using *ap* as follows:

capture a set *T* of unique transitions from *ap*

$N \leftarrow \varnothing$

sort T in ascending order of transition labels

for each t in T such that t is represented by a node n in *SDM*

$N \leftarrow N \cup \{n\}$

endfor

form the set of dependence edges E of $G[df, ap]$ as follows:

$E \leftarrow \varnothing$

//Let transitions of ap be indexed from 0 to | ap| -1//

loop from t with index | ap| -1 to t with index 1 in ap:

let n_t denote the node representing this t in $G[df, ap]$

let l be a prefix of ap, not including this t in ap

loop from the last transition in l until the first transition in l is reached:

take the present transition from l and name it pt

let n_{pt} be the node representing pt in $G[df, ap]$

if ((there is a *CD* edge from pt to this t in *SDM*) and (there is no *CD* edge incoming to n_t in E))

then add a dashed edge from n_{pt} to n_t to E

if ((there is *DD* edge from pt to this t wrt a variable v in *SDM*) and (there is no *DD* edge wrt this v incoming to n_t in E))

then add a solid edge from n_{pt} to n_t to E

//end of the inner loop//

//end of the outer loop//

//remove from G[df, ap] dependence edges that don't influence r//

traverse $G[df, ap]$ backward from node r to the source node and during the traversal, mark in $G[df, ap]$ the traversed nodes and edges

remove all unmarked nodes and dependence edges in $G[df, ap]$ to obtain an elementary *SIP* which will be called SIP_e later on.

unmark marked nodes and edges of $G[df, ap]$

//End of step 4//

After generating S_r, the set of SIPs for each requirement r in R, TS_r, the test suite for requirement r is generated.

4.4 TS_r Generation Algorithm

Algorithm: *TSr* Generation

Input: The EFSM data structure, requirement r, S_r, AP_{sip} for each SIP in S_r

Output: TS_r, the test suite for requirement r

Algorithm high level steps:

for each *SIP* in S_r do

select randomly an *ap* from AP_{sip}

form a suffix for *ap* to *Exit* in the EFSM

append this suffix to *ap*

$TS_r \leftarrow TS_r \cup \{ap\}$

Following the generation TS_r, the test suite for requirement r, the issue of the executability of the generated tests needs to be addressed. Determining test executability

is un-decidable [Rap85]. It falls in the context of *feasible path analysis*, and requires both symbolic evaluation and theorem proving [GTZ94]. Symbolic evaluation (discussed in Section 1.2.1.2) requires recording predicate expressions occurring at different transitions of the test sequence to form its path predicate. Theorem proving determines the validity of logical relations between expressions in the path predicate by solving the system of inequalities representing the path predicate. A linear optimization tool such as AMPL[8] can be used for this purpose. To use such a tool will require forming the path predicate for a test sequence and using it as a set of constraints. An artificial objective function is created and the optimizer is then used to attempt to maximize the objective function subject to the constraints. Output from the optimizer will be in one of two forms:

1) a solution, in which case the path is feasible

2) a message indicating that the constraints are contradictory, the path is infeasible.

The path predicate p of a test sequence ts of TS_r is the conjunction of predicates of transitions of ts. p obtained by applying *symbolic execution* along ts where ts is traversed backward, and each variable appearing in a predicate in transitions of ts is replaced by its symbolic value in terms of EFSM input variables. ts is executable in the EFSM iff its path predicate evaluates to true. The path predicate p of a test sequence ts is constructed as follows:

[8] AMPL is a comprehensive and powerful algebraic modeling language for linear and nonlinear optimization problems, in discrete or continuous variables. AMPL is used to describe optimization problems as the minimization or maximization of algebraic expressions in numerical decision variables, subject to constraints expressed as equalities or inequalities between algebraic expressions in the decision variables.

1. $p \leftarrow \{\}$
2. Traverse *ts* backward and find the first transition in *ts* which has a predicate. Let *t* be this transition.
3. Add the predicate of transition *t* to *p*
4. Traverse *ts* backward, starting at the transition preceding *t*. For each traversed transition *t* in *ts* do

    if *t* has a predicate then

        add the predicate of *t* to *p*

    for each assignment statement *assign* in *t* do

        if *v*, the variable defined in *assign,* is used in *p*

            then replace each occurrence of *v* in *p* by the right hand side expression of *assign*

Chapter 5

Implementation of the Proposed Test Suite Generation Method

The test suite generation method proposed in this thesis has been implemented as the TSG (Test Suite Generation) tool which is part of the Test Suite Generation/ Reduction (TSGR) software tool [Ura05]. TSGR has been implemented in the Sun Solaris Sparc 5.8 environment using C++ and the Java 2 Platform as part of a research project on using Dependence Analysis for test suite generation, test suite reduction [Rit04] and regression test suite reduction [Bo05]. TSGR consists of four major applications: an EFSM parser, a test suite generation tool, a test suite reduction tool and a regression test suite reduction tool. TSGR is also provided with a graphical user interface implemented with Java 2. Figure 5.1 illustrates the TSGR components:

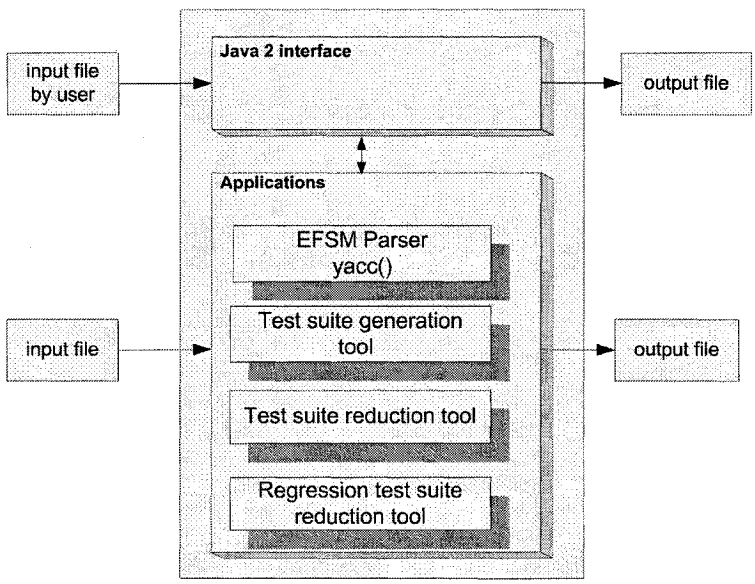


Figure 5.1 TSGR components

5.1 TSG Overview

TSG uses static dependence analysis to generate a reduced test suite for individual requirements of an EFSM model by first constructing the set of static interaction patterns for a requirement under test and then deriving a test sequence for each SIP in this set. Phases of test generation in TSG are:

1) Parsing the EFSM and constructing the EFSM data structure;

2) Constructing the SDG of the EFSM;

3) Generating additional EFSM data structures to construct the set of df-chains wrt to each TUT;

4) Constructing S_r, the set of SIPs for each requirement under test r;

5) Deriving a test suite from S_r for each requirement under test r.

Note that phases 1 and 2 have been implemented by other members of the ASERT lab: Tuong Nguyen implemented the graphical user interface for TSGR, and the EFSM parser. Yan Gao has implemented the SDG algorithm that we proposed in Chapter 3 of this thesis. These five phases are illustrated in Figure 5.2:

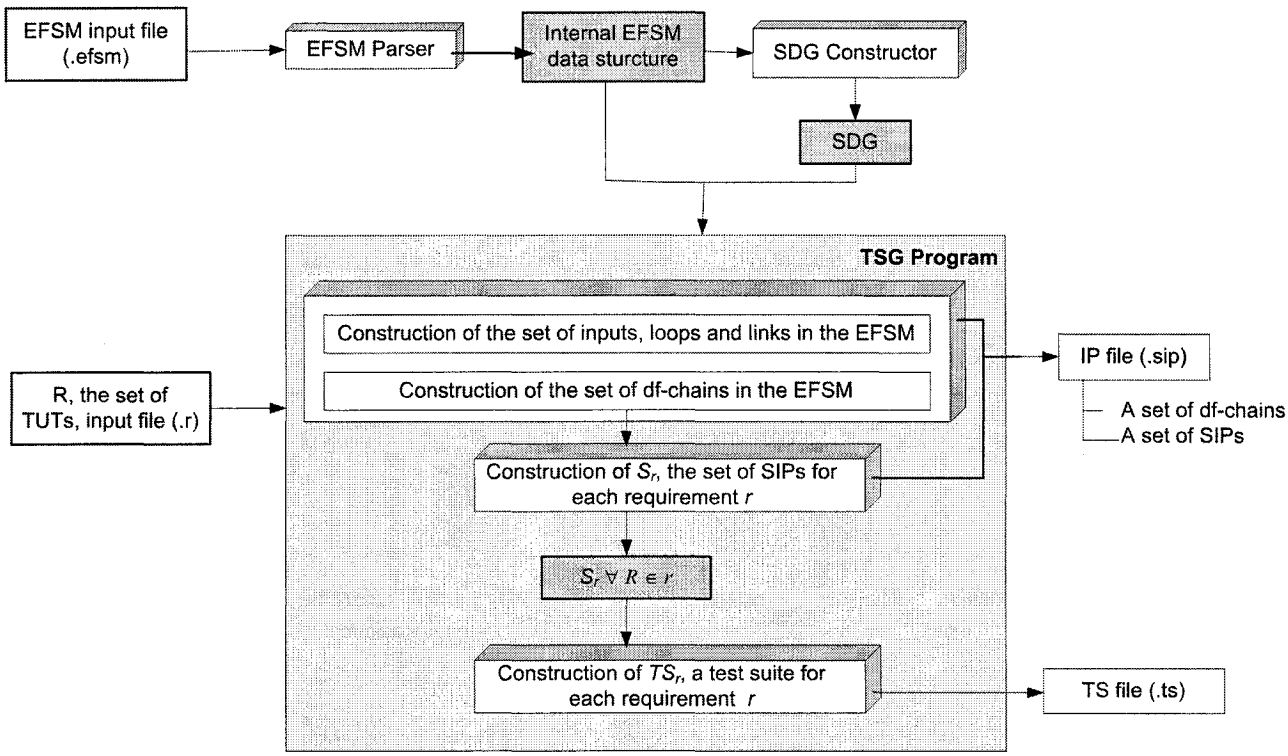


Figure 5.2 Structure of the TSG program

Note that TSG takes two input files, a file that contains the EFSM specifications (`.efsm` file) and the *R* file listing the EFSM requirements to be tested (.r file). TSG generates S_r and TS_r for each requirement in the input *R* file and outputs an *IP* file and a *ts* file with the set of df-chains and a test suite respectively, for each *r* in the *R* file. In this thesis, input and output files are formally specified by the Backus-Naur Form (BNF) [Nau60]. The BNF of the input and output files of the TSG application is presented in sections 5.2 and 5.3, respectively.

5.2 Input File Formats

- EFSM input file format

The EFSM input file is a text file describing of the elements of the EFSM model to be tested. In this thesis, such an EFSM file is given the *.efsm* extension. Table 5.1 shows the BNF for the EFSM input file.

Table 5.1 BNF definition of the EFSM input file

```
<efsm> ::=
            efsmId
            numStates startStateIndex exitStateIndex
            <transitions>
<transitions> ::=
            <transition> | <transitions> <transition>
<transition> ::=
            transition transitionId
            sourceStateIndex destinationStateIndex
            <requirement>
<requirement> ::=
            [<input>]
            [<enablingPredicate>]
            /
            [<actions>]
<actions> ::=
            <action> | <actions> <action>
<action> ::=
            <output> | <assignment> | <set> | <reset> |
<procedureCall>
<input> ::=
            inputId ( [<parameters>] )
<output> ::=
            outputId ( [<parameters>] )
<enablingPredicate> ::=
            <variableIds> [/* booleanExpression */]
<assignment> ::=
```

```
                <variableId> := <expression>
<set> ::=
                set ( constant , timerId )
<reset> ::=
                reset ( timerId )
procedureCall ::=
                procedure ( procedureId ( <variableIds> [; <variableIds>])
) { <pbrDefs> }
<parameters> ::=
                <parameter> {, <parameter>}*
<parameter> ::=
                <variableId> | constant
<variableIds> ::=
                <variableId> {, <variableId>}*
<pbrDefs> ::=
        <pbrDef> | <pbrDefs> <pbrDef>
<pbrDef> ::=
                <variableId> := <expression> ;
<expression> ::=
                function ( <variableIds> ) | constant
<variableId> ::= id
```

An example ".efsm" file for the ATM EFSM given Figure 2.2 is given in Appendix A.2 of this thesis.

- ***R* input file format**

TSG generates a test suite for individual requirements in the EFSM. The *R* (or requirements) file contains the set of EFSM requirements to be tested such that each requirement is represented by an EFSM transition under test, and has been given the ".r" extension. The BNF of the *R* file is shown in Table 5.2.

Table 5.2 BNF definition of the *R* input file

```
<r> ::=
 efsmId <tuts>
<tuts> ::=
  <tut> | <tuts> <tut>
<tut> ::=
 transitionId
```

5.3 Output File Formats

TSG generates two output files: an IP file for the set of SIPs and their corresponding df-chains for each TUT in the *R* input file and a TS file for the generated test suite for each TUT in the *R* input file.

- IP output file format

The IP output file contains, for each TUT in the *R* file, the set of df-chains and the set of SIPs wrt TUT. The IP file is given the ".sip" extension. Table 5.3 below shows the BNF of the IP file.

Table 5.3 BNF definition of the IP output file

```
<sip> ::=
        efsmId <tut> <ips>
<tut> ::=
        transitionId
<ips> ::=
        <ip> | <ips> <ip>
<ip> ::=
        ip ipId [<testIds> | <dfs>] <nodes>
<testIds> ::=
        testId | <testIds> testId
<dfs> ::=
        <df> | <dfs> <df>
<df> ::=
        <dupairs> <path>
<dupairs> ::=
        <dupair> | <dupairs> <dupair>
<dupair> ::=
        <definition> <use> /* 0=def, 1=c-use, 2=p-use */
<definition> ::=
        (0 , variableId , transitionId , occurrenceOrder)
<use> ::=
```

```
        (1 | 2 , variableId , transitionId , occurrenceOrder)
<path> ::=
        transitionId | <path> transitionId
<nodes> ::=
        <node> | <nodes> <node>
<node> ::=
        node nodeIndex <label> [<adjacencySet>]
<label> ::=
        transitionId
<adjacencySet> ::=
        <reverseSet> | <nonreverseSet>
<reverseSet> ::=
        <reverse> | <reverseSet> <reverse>
<reverse> ::=
        inc sourceIndex <dependencyType>
<nonreverseSet> ::=
        <nonreverse> | <nonreverseSet> <nonreverse>
<nonreverse> ::=
        out destinationIndex <dependencyType>
<dependencyTye> ::=
        dat | ctl
```

An example of the *IP* file for the ATM EFSM given in Figure 2.2 is given in Appendix A.7.

- TS output file format

The TS output file is given the ".ts" extension and consists of a set of tests for each TUT in the *R* input file. A test is a complete test sequence of transitions in the EFSM, that starts at the start state and ends at the exit state of the EFSM. The BNF representation of the TS output file is detailed in table 5.4 below:

Table 5.4 BNF definition of the TS output file

```
<ts> ::=
            efsmId <tuts> <tests>
<tuts> ::=
            <tut> | <tuts> <tut>
<tut> ::=
```

```
                transitionId
<tests> ::=
                <test> | <tests> <test>
<test> ::=
                test testId <transitionSeq>
<transitionSeq> ::=
                transitionId | <transitionSeq> transitionId
```

An example of the *TS* output file for the ATM EFSM given in Figure 2.2 is given in Appendix A.6.

5.4 TSG Program

TSG uses dependence analysis to generate a reduced test suite for individual requirements of an EFSM model. We have previously identified five phases for the test suite generation:

1) Parsing the EFSM and constructing the EFSM data structure;

2) Constructing the SDG of the EFSM;

3) Generating additional EFSM data structures to construct the set of df-chains wrt to each TUT;

4) Constructing S_r, the set of SIPs for each requirement under test r;

5) Deriving a test suite from S_r for each requirement under test r.

A detailed description of each of these five phases follows.

Phase 1: Parsing the EFSM and constructing the EFSM data structure

The EFSM parser is generated by *Yacc*[9] based on the EFSM grammar written in BNF notation, as shown in Table 5.1. It is used with a lexical analyzer check that the EFSM specified in the input file is syntactically correct, classifies EFSM variable occurrences and generates the internal data structure for the EFSM as given in Section 3.2.1 of this thesis.

Phase 2: Construction of the SDG

Based on the internal data structure of the EFSM, control and data dependencies are captured to build the SDG. Data dependencies are first identified as follows: for each variable v in the EFSM, TSG checks the existence of a def-clear path wrt v, between each definition of v and each use of v in the EFSM. Checking the existence of a def-clear path in the EFSM is not a trivial task due to existence of loops in the EFSM. To solve this problem in an efficient manner, we use the notion of *transitive closure*. The transitive closure of a directed graph G which provides reachability information about G is defined in [Goo02] as follows:

Given a digraph G, the transitive closure of G is the digraph G^* such that

i) G^* has the same vertices as G,

ii) if G has a directed path from vertex u to vertex v ($u \neq v$) then G^* has a directed edge from u to v.

In other words, if v reaches u (or u is reachable from v) in G then there is an edge from v to u in G^*. In our work, we constructed the reachability information of the EFSM

9 Yacc is a program that serves as the standard parser generator in C/C++ on Unix systems. The name is an acronym for "Yet Another Compiler Compiler." It generates a parser based on a grammar written in BNF notation. Since the parser generated by Yacc requires a lexical analyzer, it is often used in combination with a lexical analyzer generator (Lex is used for the TSGR tool)

by forming sets S_{in}*(Z) and S_{out}*(Z) for each state Z in the EFSM, where S_{in}*(Z) denotes $\{s \mid s$ is a state and state s reaches state Z in $M\}$, and S_{out}*(Z) denote $\{s \mid s$ is a state and state Z reaches state s in $M\}$.

Set S_{in}*(Z) is generated such that states that reach Z with a shortest path to Z such that this path has the least number of edges, are added to S_{in}*(Z) first, and so on. The same applies for S_{out}*(Z). Then, the existence of a def-clear path between two transitions T_i and T_k wrt to a variable v in the EFSM is determined by recursively checking if states reachable from T_i have an outgoing transition that doesn't define v. A detailed version of the SDG algorithm, given in Appendix D.2 of this thesis, outlines the algorithm used to check the existence of a def-clear path.

After identifying data dependencies in the EFSM, control dependencies are identified between EFSM transitions, by forming post dominance relationships between states, and between states and transitions in the EFSM. The EFSM reachability information is also used to solve this problem. Details of this method are given in Appendix D.2 of this thesis.

The complexity of SDG generation is quadratic in S which is a set of states in an EFSM.

Phase 3: Generation of the set of df-chains in the EFSM

During this phase, TSG generates for each requirement r in R, the set $DF_{AP}(r)$ of df-chains ending with the use of a variable v, $\forall\ v \in V_r$, together with their respective sets AP_{df} of activating paths. To generate the set of such df-chains, TSG keeps a stack of du-pairs where a df-chain df is constructed backward from TUT to an input, by using the

recursive DFS logic: a du-pair that ends at *TUT* is first pushed to the stack, then, linked du-pairs are identified by using the set *links*(M) and pushed to the stack. When a du-pair that starts by an input from the set *input*(M) is pushed, a (complete) df-chain is formed in the stack. This df-chain is popped from the stack, added to $DF_{AP}(r)$ and the search for additional df-chains continues.

For each df-chain *df* added to $DF_{AP}(r)$, TSG forms the set DCP_{df} of def-clear paths for each du-pair in *df*. Then the set AP_{df} of activating paths for *df* is constructed by forming all combinations of def-clear paths of du-pairs in *df*. Prefixes are appended to activating paths that do not start by a transition emanating from the start state of the EFSM. As an example, the set of df-chains and their corresponding sets of activating paths generated for each r in the ATM EFSM are given in Appendix A.4.

Phase 4: Construction of TS_r, the test suite for requirement r

TSG generates an elementary SIP, SIP_e, for each pair (*df*, *ap*) generated in Phase 3 as follows: the control and data dependencies that are encountered during the traversal of *ap* are identified and represented in G[*df*, *ap*], a sub-graph of the SDG. Dependencies affecting the TUT are identified by traversing G[*df*, *ap*] backwards from the TUT and marking all traversed nodes and edges. All unmarked nodes and edges are removed from G[*df*, *ap*] to obtain SIP_e. More than one (*df*, *ap*) pair may lead to the generation of the same SIP_e. The set AP_{SIP} of *ap*'s that lead to the generation of the same SIP_e is formed for each SIP_e generated.

Following the generation of the set S_{r_elem} of SIP_e's for r, S_r is generated by adding combinations of applicable self loop data dependencies to nodes of SIP_e to obtain

additional SIPs. These SIPs are associated with the set AP_{SIP} where the EFSM paths of the loops added to these SIPs are inserted to paths of AP_{SIP}.

TS_r, the test suite for requirement r is formed by randomly selecting a path ap from AP_{SIP} for each SIP in S_r. Since paths in AP_{SIP} start by a transition eminating from the EFSM's start state and end at r, a path from r to the exit state in the EFSM is appended to each selected ap to obtain a complete test sequence wrt r.

Chapter 6

Case Studies

Three case studies were conducted to apply the test generation method proposed in this thesis, namely the Automated Teller Machine (ATM) system, Vending Machine System (VMS) and Cruise Control System (CCS) case studies. For the ATM case study, we have used the EFSM system model presented in [Vay02] and shown in Figure 2.2. We have developed an EFSM model for the VMS case study, given in Appendix B.2 and the CCS case study, given in Appendix C.2. Requirements in English for the ATM, VMS and CCS are presented in Section 2.2.2, Appendix B.1 and Appendix C.1, respectively.

For each EFSM model, the following experiments have been carried out:

1. Formal representation of the requirements in an EFSM model;
2. Automatic generation of the SDG related to the EFSM model;
3. Automatic generation of the set of df-chains in the EFSM model,
4. Automatic generation of a test suite related to each requirement in the EFSM,

The output obtained after conducting these experiments are given in the following sections of this chapter.

6.1 ATM Case Study

The requirements of the ATM System (given in Section 2.2.2 of this thesis) are formally represented as an EFSM model (Figure 2.2). Results of the SDG and df-chain generation for the ATM EFSM are shown in Appendix A.3 and Appendix A.4 respectively.

Let M denote the ATM EFSM,

R be the set of requirements for M, $R = \{r \mid r$ is a requirement in $M\}$. R = {T3, T5, T6, T7, T9}

S_{r_elem} be the set of elementary Static Interaction Patterns or SIP_{elem}s for a requirement r in R,

S_r be the set of Static Interaction Patterns or SIPs for a requirement r in R,

TS_r be the test suite generated for r in R by using the test generation approach proposed in this thesis.

S_{r_elem}, S_r and TS_r have been generated for the ATM EFSM. Table 6.1 below shows the test generation results:

Table 6.1 Test generation output for the ATM EFSM

TUT	$\lvert S_{r\_elem}\rvert$	Expected $\lvert S_r\rvert$	Obtained $\lvert S_r\rvert$	$\lvert TS_r\rvert$
T3	2	3	3	3
T5	4	14	14	14
T6	4	14	14	14
T7	17	57	57	57
T9	0	0	0	0

The column titled "Expected $|S_r|$" shows the expected size of S_r for a requirement r. These numbers were obtained by the test suite reduction experiments conducted in [Vay02] and in [Rit04], in addition we have manually generated S_r for each r in R by generating all possible SIPs for r from the SDG, to confirm these results obtained by [Vay02][10] and [Rit04]. The "Obtained $|S_r|$" column shows the size of S_r we obtained after applying the test generation method we proposed in this thesis. S_r for r = T5 is shown in

[10] Authors in [1] have first mistakenly obtained $|S_r| = 16$ for both r = T5 and T6. They later confirmed the values for $|S_r|$ for r = T5 and T6 as identified by us and as shown in Table 6.1 after our personal communication with them

Appendix A.5 of this thesis. Note that the obtained S_r set matches the expected S_r for each r in R.

Test suite TS_r for each requirement r in R is shown in Appendix A.6 of this thesis. It is important to note that TS_r contains both feasible and infeasible test sequences for a requirement r. Unfeasible test sequences are marked by * in Appendix A.6 and B.6 and C.6 of the thesis. TS_r for r = T3 for instance is shown in Table 6.2 below. Figure 6.1 illustrates S_r for r = T3. Test_1, Test_2 and Test_3 have been generated for each SIP in S_r. None of these three tests is feasible because self loop transition T2 in the ATM EFSM needs to be traversed exactly three times for transition T3 to be taken in a test sequence. This is due to the fact that, according to our test generation method, EFSM loops are traversed at most once. Feasible test sequences wrt r = T3 are obtained if T2 is traversed a third time in Test_3. On the other hand, when r = T5 or T6 or T7, all paths leading to r are feasible, hence all generated tests are feasible for this r.

Table 6.2 TS_r for TUT = T3 in the ATM EFSM

ATM_System
T3
Test Test_1 T1 T3
Test Test_2 T1 T2 T3
Test Test_3 T1 T2 T2 T3

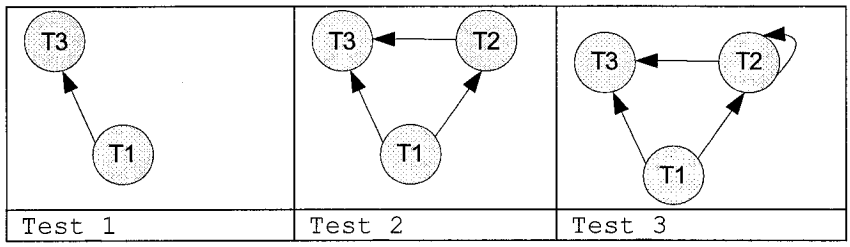


Figure 6.1 SIP_r for TUT = T3 in the ATM EFSM

For TUT = T9, $|S_r| = 0$. This can be explained by the fact that no df-chains wrt T9 are generated by TSG, because no variable is used in T9.

6.2 Vending Machine Case Study

The requirements of the Vending Machine System (given in Appendix B.1) are formally represented as an EFSM model (Appendix B.2). The SDG generated for the VMS EFSM is given in Appendix B.3. The df-chains generated wrt requirements in r= T2 and r= T5 are given in Appendix B.4 and TS_r, an example of S_r is given in Appendix B.5 for TUT = T2, the test suite generated for a requirement r is given for each requirement r in R in Appendix B.6 of this thesis. Table 6.3 below shows the sizes of the generated test suites.

Table 6.3 Test generation output for the VMS EFSM

TUT	$\|S_{r\_elem}\|$	$\|S_r\|$	$\|TS_r\|$
T2	5	7	7
T5	20	40	40
T8	16	28	28
T11	14	24	24
T12	14	24	24

According to the S_r Generation algorithm, self loop data dependencies are not considered when generating the set of df-chains in the EFSM. This implies that self loop data dependence edges that represent such data dependencies shall not occur in an elementary SIP. However, we noticed that some of the elementary SIPs generated for the VMS EFSM have a self loop data dependence edges. We have explained this by making the following statement: if, for an elementary SIP generated for a df-chain *df*, a data dependence edge corresponding to a du-pair *du* such that *du* is not in *df*, occurs in this

SIP, then the def-clear path of *du* is a subpath of the activating path of *df*. The following example illustrates this statement.

Take $SIP_{e\_}12$, and elementary SIP in S_{r_elem} with index 12 for r = T8. $SIP_{e\_}12$ is generated from df-chain df = {($d_{T3}^{totalDeposit}$, $u_{T8}^{totalDeposit}$), (d_{T8}^{change}, u_{T12}^{change}), (d_{T12}^{credit}, u_{T4}^{credit}), (d_{T4}^{price}, u_{T8}^{price})}. $SIP_{e\_}12$ is shown in Figure 6.2.

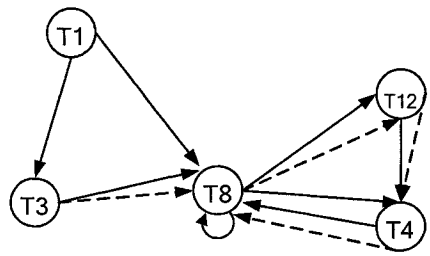


Figure 6.2 SIP_{e_12} for r = T8 in the VMS EFSM

Let ap_{df} = (T1, T3, T8, T12, T4, T8) be an activating path of df ap_{df} (T1 is a prefix for ap_{df}, def-clear path for ($d_{T3}^{totalDeposit}$, $u_{T8}^{totalDeposit}$) = (T3, T8), def-clear path for (d_{T8}^{change}, u_{T12}^{change}) = (T8, T12), def-clear path for (d_{T12}^{credit}, u_{T4}^{credit}) = (T12, T4) and def-clear path for (d_{T4}^{price}, u_{T8}^{price}) = (T4, T8))

Self loop data dependence edge on node T8 in the SDG of the VMS EFSM represents a du-pair from transition T8 to transition T8 in the VMS EFSM, wrt variables {*Q*, *D*, *N*, *B*, *G*, *M*}. For instance take du-pair du = (d_{T8}^{Q}, u_{T8}^{Q}) wrt variable Q in the VMS EFSM with def-clear path dfc_{du} = (T8, T12, T4, T8). du is represented by self loop data dependence edge on node T8 in $SIP_{e\_}12$. This edge occurs in $SIP_{e\_}12$ because dfc_{du} = (T8, T12, T4, T8) is a subpath of ap_{df} = (T1, T3, T8, T12, T4, T8).

6.3 Cruise Control Case Study

A Cruise Control System (CCS) maintains the speed of a car at a pre-selected value. The requirements of the CCS (given in Appendix C.1) are formally represented as an EFSM model (Appendix C.2). The SDG generated for the CCS EFSM is given in Appendix C.3. The df-chains generated wrt requirements T3, T4, T8, T11, T12, are given in Appendix C.4. Table 6.4 below shows the sizes of the generated test suites for each requirement r in R. Note that all test sequences in TS_r for each r in R are feasible in the CCS, due to the fact that, for each transition t in the CCS EFSM that has a predicate expression, the constraints of this predicate apply to the variables that are defined in the input of t. This is in turn due to the nature of the CCS. It has a monitor that reads the current speed of the vehicle every second or so. Therefore, variables used in the predicate of a transition do not conflict with variables in other transitions in the EFSM.

Table 6.4 Test generation output for the CCS EFSM

TUT	$\lvert S_{r\_elem}\rvert$	$\lvert S_r\rvert$	$\lvert TS_r\rvert$
T2	0	0	0
T3	429	1465	1465
T4	429	1465	1465
T5	428	1468	1468
T6	106	408	408
T7	114	440	440
T8	0	0	0
T9	0	0	0
T10	0	0	0
T11	129	435	435
T13	0	0	0

The size of the S_r generated for each r in the CCS EFSM is considerably large compared to the size of S_r's obtained in the ATM and the VMS case studies, due to the

generation of large AP_{df}'s for df-chains wrt a requirement. For example, Table 6.5 below shows the set of df-chains obtained for TUT= T3 in the CCS EFSM:

Table 6.5 Illustration of $|AP_{df}|$ for df-chain df

	df	$\|AP_{df}\|$
1	{d(*current_speed*,T02,3), u(*current_speed*,T02,7), d(*cruise_speed*,T02,8), u(*cruise_speed*,T03,2)}	32
2	{d(current_speed,T02,3), u(current_speed,T02,7), d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3) d(*cruise_speed*,T06,4), u(*cruise_speed*,T03,2)}	448
3	{d(*current_speed*,T02,3), u(*current_speed*,T02,7), d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3) d(*cruise_speed*,T07,4), u(*cruise_speed*,T03,2)}	2048
4	{d(*current_speed*,T02,3), u(*current_speed*,T02,7), d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3), d(*cruise_speed*,T07,4), u(*cruise_speed*,T03,2)}	448
5	{d(*current_speed*,T02,3), u(*current_speed*,T02,7), d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3), d(*cruise_speed*,T06,4), u(*cruise_speed*,T07,3), d(*cruise_speed*,T07,4), u(*cruise_speed*,T03,2)}	12992

The size of the set of activating paths obtained for df-chains for the CCS EFSM (for TUT = T3 and all TUTs) is considerably large, due to the existence of multiple self loop transitions on one state in the CCS EFSM. (In fact, state S2 has 8 loops, namely, T3, T4, T5, T6, T7, (T8, T11), (T9, T11), (T10, T11)). The number of possible def-clear paths generated for a du-pair that traverses state S2 is large because all combinations of applicable loops are considered, and therefore, the number of activating paths for a df-chain containing more that one of such du-pairs is very large. For example, consider *df* #2 for TUT = T3 shown in Table 6.6 below, where 2048 = 1*64*32

Table 6.6 Details of $|AP_{df}|$ for df-chain #2 for TUT=T3

df #	***df* for TUT = T3**	**# def-clear paths for du-pairs of *df***	$\|AP_{df}\|$
2	(0,current_speed,T02,3)(1,current_speed,T02,7)	1	2048
	(0,cruise_speed,T02,8)(1,cruise_speed,T06,3)	64	
	(0,cruise_speed,T06,4)(2,cruise_speed,T03,2)	32	

When generating a def-clear path that traverses state S2, all possibilities of traversal of loops (with def-clear transitions) on S2 are considered. There is no need to take order of traversal of these loops into account. We used the notion of *combination* to determine a pattern of traversal of such loops. A *combination* is an unordered grouping of a set. Combinations are represented as ${}_nC_r$, where unordered subgroups of size r are selected from a set of size n. ${}_nC_r$ is calculated as follows:

$$ {}_nC_r = \frac{n!}{(n-r)!r!} $$

For instance, let TUT = T3, *df* be the df-chain # 2 for TUT, *du* be the second du-pair of *df*. *du* = (0,*cruise_speed*,T02,8)(1,*cruise_speed*,T06,3) and $DFSet_{du}$ denote the set of dfc_{du} of def-clear paths of *du* denote a def-clear path for du-pair *du*. The set of EFSM transitions traversed in df-clear paths of *df* are {T2, T3, T4, T5, T8, T9, T10, T11, T12}. 6 EFSM self loops can be traversed by def-clear paths of *du*:

1. T3
2. T4
3. T5
4. (T8, T11)
5. (T9, T11)
6. (T10, T11)

Let $Comb(dfc_{du})$ denote the set of combinations of loops traversed in a row in def-clear paths of *du*.

$$ |Comb(dfc_{du})| = {}_6C_0 + {}_6C_1 + {}_6C_2 + {}_6C_3 + {}_6C_4 + {}_6C_5 + {}_6C_6 $$

$$ = \frac{6!}{(6-0)!*0!} + \frac{6!}{(6-1)!*1!} + \frac{6!}{(6-2)!*2!} + \frac{6!}{(6-3)!*3!} + \frac{6!}{(6-4)!*4!} + $$

$$\frac{6!}{(6-5)!*5!} + \frac{6!}{(6-6)!*6!}$$

$$= 1+ 6+ 15+ 20+ 15+ 6+ 1 = 64$$

After performing the three experiments, it is observed that the TSG program provides ease of use and substantial degree of automation for the generation of efficient requirement-based test suites. The size of the generated test suite varies considerably with the number of loops in the EFSM, that we force to be traversed 0 and 1 times in a test sequence. Table 6.7 below illustrates the complexity of the three EFSMs in regards if the number of loops, states and transitions in a given EFSM.

Table 6.7 Complexity of the ATM, VMS and CCS EFSM's

	ATM EFSM	**VMS EFSM**	**CCS EFSM**
# of loops	4	4	8
# of states	5	5	5
#of transitions	9	12	14
Average size of $\|TS_r\|$	18	25	228
loops	T2 T5, T8 T6, T8 T7, T8	T3, T8, T12 T4, T8, T12 T5 T6	T3 T4 T5 T6 T7 T8,T11 T9,T11 T10,T11

Chapter 7

Conclusions

7.1 Final Remarks

The problem addressed by this thesis is that of generating efficient requirement based test suites by using dependence analysis. In the thesis, we assume that requirements can be represented as a single EFSM and each requirement can be adequately represented by a single transition. The generated test suites cover all possible patterns of interactions between data and control dependencies in an EFSM model.

The problem was approached in this thesis by formalizing the generation of the SDG for an EFSM and by considering an interaction pattern between data and control dependencies of the EFSM as df-chain and its activating path. The df-chain captures data dependencies in this interaction pattern, whereas control dependencies in this interaction pattern are captured by the activating path associated with this df-chain. An existing notion of df-chains was adapted and several assumptions were made to solve this problem.

It is interesting to note that the EFSM model has not been widely applied for verification, and particularly for automatic test pattern generation. The reason of this fall down depends on the difficulty of traversing an EFSM. The existence of loops and the existence of infeasible paths in an EFSM graph due to conflicts among actions and conditions in the EFSM are major problems when generating test sequences from an EFSM model. In this thesis we have addressed the existence of loops by restricting the number of times they are traversed in a test sequence. Traversing loops 0 and 1 time in the ATM EFSM for example, didn't impact on the interaction coverage capability of the

generated test suite for this EFSM. The minimal test suites identified by experiments conducted in [Vay02] and [Rit04] for the ATM EFSM are automatically generated by the test generation method we propose in this thesis.

7.2 Summary of Contributions

Below we list the major contributions of the thesis:

We have formalized the generation of the SDG in the SDG generation algorithm. Then, we analyzed the reduction approach based on dependence analysis proposed in [Rit04] to define a novel test generation method for requirements in an EFSM model, based on a SDG. The test generation method proposed in this thesis solved the complex problem of automatically generated all patterns of interactions between data and control dependencies of the EFSM, wrt a requirement under test. We have adapted the notion of df-chains proposed in [UY93] to the requirements to construct such interaction patterns.

We have proposed and developed algorithms to generate the set of df-chains wrt a requirement under test in an EFSM. In another contribution, we proposed and implemented algorithms to generate the set of static interaction patterns and the test suite for a requirement under test.

We have developed a test suite generation program which automates the generation of efficient test suite for requirements represented in an EFSM models.

The VMS and CCS EFSMs were defined, and three case studies conducted to test the proposed test generation approach.

7.3 Directions for Future Research

The part of the test generation method that was not fully addressed in this thesis and should be addressed in future work is the identification of infeasible tests in the generated test suites. A program needs to be developed to automatically generate the path predicate of each generated test sequence. A batch session of a numerical optimization tool such as AMPL can be executed from the path predicate generation program to evaluate the path predicate. A test sequence is feasible if its path predicate evaluates to true.

On another hand, we assumed that each requirement can uniquely be represented by a single transition in the given EFSM. It would also be interesting to expand the approach presented in this thesis to those EFSMs where some requirements may uniquely be represented by a sequence of transitions.

Analysis of the effectiveness of the proposed test generation method also needs to be performed. One way to evaluate the effectiveness of this method is to compare it against other commonly used test generation methods for EFSMs. Such methods involve criteria such as 1) Transition coverage, where each transition in the EFSM is covered at least once, 2) Constrained N Path coverage, where each path in the EFSM is covered at least once, subject to the constraint that a transition may not appear on such a path more than N times, and 3) Predicate coverage, where each transition is covered at least once while applying well known techniques from predicate testing to the transition guard conditions. There are several dimensions along which the various methods could be compared such as the expressive power of the notations used, the amount of information required to be modeled to generate an effective set of test cases, the extent of automated support for the technique, number of generated test sequences, test feasibility identification capability,

fault detection effectiveness, of the generated test cases. Before performing performance evaluation, the test objective(s) needs to be identified. For the method proposed in the thesis, our test objectives would be 1) A small number of generated test sequences, 2) Support for complex state based systems, 3) Test feasibility identification. Nonetheless, it would be interesting to investigate on its fault detection effectiveness. This can be done by using the ATM and VMS case studies presented in Chapter 6, as follows:

1) Test suites are generated by a) proposed method (Requirement based test generation using static dependence analysis), b) transition coverage, c) constrained *N* path coverage, and d) predicate coverage. Test suites for the last three methods are generated automatically or manually depending on the availability of tool support. Test suites for the method we proposed are generated for each requirement in ATM and VMS EFSMs by using the TSG tool
2) Implementations for the ATM and VMS are produced. Java can be used for these implementations where the ATM system for example, can be implemented by using 2 classes with 20 lines of code, 10 statements and 4 branches
3) Several faulty versions are of these implementations are generated
4) The generated test suites are applied by using each method on both the correct and the faulty implementations
5) Branch and statement coverage is measured for the four methods in both the correct and the faulty implementations. Methods will be evaluated and compared based on the percentage of coverage performed and based on the number of faults caught.

References

[Bei90] Beizer, B., "Software testing techniques" (2nd ed.), Van Nostrand Reinhold, New York, 1990

[Bir83] Bird, D.; and Munoz, C. "Automatic generation of random self-checking test cases" *IBM System Journal,* 22(3):229-245, 1983

[Bo05] Xie, B., "Using Dependence Analysis in Regression Test Suite Reduction" Masters Thesis in Computer Science, University of Ottawa, 2005

[Bou97] Bourhfir, C.; Dssouli, R.; Aboulhamid, E., "Automatic Executable Test Case Generation for EFSM Specified Protocols", Proceeding of IWTCS, pp. 75-90, 1997

[Cla96] Clarke, E.; Wing, J., "Formal methods: state of the art and future directions" ACM Computing Surveys (CSUR) archive Volume 28, Issue 4 (December 1996),Pages:626-643

[Cl89] Clarke, L.A.; Podgurski, A.; Richardson, D.J.; Zeil, S.J., "A formal evaluation of data flow path selection criteria" Software Engineering, IEEE Transactions on Volume 15, Issue 11, Nov. 1989 Page(s):1318 - 1332

[Dal98] Dalal, S.; Jain, A.; Karunanithi, N.; Leaton, J.; Lott. C., "Model-based Testing of a Highly Programmable System," issre, vol. 00, no. , p. 174, 1998

[Fer96] Ferguson, R.; Korel, B., "The chaining approach for software test data generation" January 1996 ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 5 Issue 1

[Fra88] Frankl, P. G.; and Weyuker, E. J., "An Applicable Family of Data Flow Testing Criteria", IEEE Trans. Software Eng., 14, 10, pp. 1483-1498, 1988

[Gil62] Gill, A., "Introduction to the Theory of Finite-state Machines". McGraw-Hill, 1962

[Goo02] Goodrich, M. T.; and Tamassia, R., "Data Structures and Algorithms in Java", 2nd edition, year of Publication: 2000

[GTZ94] Goldberg, A.; Wang, T. C.; Zimmerman, D., "Applications of feasible path analysis to program" Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis Seattle. Pages: 80 - 94

[Hie04] Hierons, R. M., "Testing from a Nondeterministic Finite State Machine Using Adaptive State Counting", IEEE October 2004 (Vol. 53, No.10) pp.1330-1342

[ISO89] ISO: DIS8807: Information Processing Systems. Open Systems Interconnection ~ LOTOS ~ A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour'. July 1989

[Kin96] King, J., "Symbolic execution and testing" Comm. of the ACM, 19(7):385–394, 1976

[Kor90] Korel, B., "Automated software test data generation" *IEEE Transactions on Software Engineering,* 16(8):870-897, Aug. 1990

[LK83] Laski, J.W.; and Korel, B., "A Data Flow Oriented Program Testing Strategy," IEEE Transactions on Software Engineering, 9(5):347-354, May 1983

[Mye04] Myers, G. J., "The Art of Software Testing" John Wiley & Sons; 2 edition June 2004

[Nau60] Naur, P., "Revised Report on the Algorithmic Language ALGOL 60.", *Communications of the ACM*, Vol. 3 No.5, pp. 299-314, May 1960

[Nta84] Ntafos, S.C., "On Required Element Testing" 1EEE Transactions on Software Engineering, 10(11):795-803, Nov. 1984

[Off96] Offutt, J.; and Hayes, J., "A semantic model of program faults" In International Symposium on Software Testing and Analysis (ISSTA 96), pages 195{200. ACM Press, 1996

[Pro82] Probert, R., "Grey-Box (Design-Based) Testing Techniques" Proc. 15th Hawaii Int'l. Conf. Sys. Sci., 1982,pp. 94–102

[Rap85] Rapps, S.; and Weyuker, E. J., "Selecting Software Test Data Using Data Flow Information", IEEE Trans. Software Eng., 11, 4, pp. 367-375, 1985

[Rit04] Ritthiruangdech, P., "Using Dependence Analysis in Requirement Based Test Suite Reduction" Masters Thesis in Computer Science, University of Ottawa, 2004

[Sar87] Saracco, R.; and J. Tilanus, P. A., "CCITT SDL: Overview of language and its application," Comput. Networks and ISDN Sysr., vol. 13, no. 2, pp. 65-74, 1987

[Ura05] Dr. Hasan Ural http://www.site.uottawa.ca/~ural/TSR

[USW00] Ural, H.; Saleh, K.; and Williams, A., "Test generation based on control and data dependencies within system specifications in SDL", Computer Communications, Vol.23, No.7, 2000, pp.609-627

[UY91] Ural, H.; and Yang, B., "A test sequence selection method for protocol testing", IEEE Transactions on Communications, Vol.39, No.4, 1991, pp.514-523

[UY93] Ural, H.; and Yang, B., "Modeling software for accurate data flow representation", Proc. of 15th Int. Conference on Software Engineering, ICSE'93, Baltimore, Maryland, May 1993, pp.277-286

[Vay02] Vaysburg, B.; Tahat, L. H.; Korel, B., "Dependence Analysis in Reduction of Requirement Based Test Suites", Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02, Roma, Italy), pp. 107-111, ACM Press, ISBN: 1-58113-562-9, 2002

APPENDIX A: ATM System

A.1 Requirements of the ATM

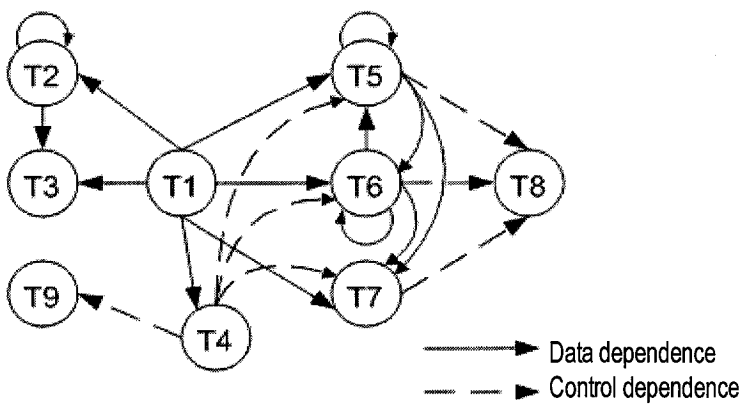
The requirements and the EFSM for the ATM system are given in Section 2.2.2 of this thesis

A.2 ".efsm" file for the ATM EFSM

```
ATM_System
5 0 4

transition T1
0 1
Card(pin, b) /
Prompt_for_PIN()
attempts := constant

transition T2
1 1
PIN(p)
p, pin, attempts /* [(p!=pin) and (attempts<3)] */ /
Display_error()
attempts := function(attempts)
Prompt_for_PIN()

transition T3
1 4
PIN(p)
p, pin, attempts /* [(p!=pin) and (attempts==3)] */ /
Display_error()
Eject_card()

transition T4
1 2
PIN(p)
p, pin /* [p==pin] */ /
Display_menu()

transition T5
2 3
Withdrawal(w) /
b := function(b, w)

transition T6
2 3
Deposit(d) /
b := function(b, d)

transition T7
2 3
Balance() /
```

```
print(b)

transition T8
3 2
Continue() /
Display_menu()

transition T9
2 4
Exit() /
Eject_card()
```

A.3 SDG of the ATM EFSM



A.4 df-chains of the ATM EFSM

Note that the bold part of an *ap* is the prefix generated for this *ap* when the corresponding df-chain doesn't start by a transition which start state is not the start state of the EFSM.

The set of df-chains generated for TUT = T3

	Df	AP_{df}
1	{d(*attempts*,T1,3),u(*attempts*,T3,2)}	(T1,T3)
2	{d(*attempts*,T1,3),u(*attempts*,T2,5) d(*attempts*,T2,6),u(*attempts*,T3,2)}	(T1,T2, T3)
3	{d(*pin*,T1,2),u(*pin*,T3,3)}	(T1,T3) (T1,T2, T3)

The set of df-chains generated for TUT = T5

	Df	AP_{df}
1	{d(*b*,T1,1), u(*b*,T5,3)}	(T1,T2,T4,T5) (T1,T2,T4,T7,T8,T5) (T1,T4,T5) (T1,T4,T7,T8,T5)
2	{d(*d*,T6,1), u(*d*,T6,2), d(*b*,T6,4), u(*b*,T5,3)}	(**T1,T2,T4,**T6,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T5) (**T1,T4,**T6,T8,T5) (**T1,T4,T7,T8,**T6,T8,T5) (**T1,T2,T4,**T6,T8,T7,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T7,T8,T5) (**T1,T4,**T6,T8,T7,T8,T5) (**T1,T4,T7,T8,**T6,T8,T7,T8,T5)
3	{d(*b*,T1,1), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T5,3)}	(T1,T2,T4,T6,T8,T5) (T1,T2,T4,T7,T8,T6,T8,T5) (T1,T4,T6,T8,T5) (T1,T4,T7,T8,T6,T8,T5) (T1,T2,T4,T6,T8,T7,T8,T5) (T1,T2,T4,T7,T8,T6,T8,T7,T8,T5) (T1,T4,T6,T8,T7,T8,T5) (T1,T4,T7,T8,T6,T8,T7,T8,T5)
4	{d(*w*,T5,1), u(*w*,T5,2), d(*b*,T5,4), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T5,3)}	(**T1,T2,T4,**T5,T8,T6,T8,T5) (**T1,T2,T4,T7,T8,**T5,T8,T6,T8,T5) (**T1,T4,**T5,T8,T6,T8,T5) (**T1,T4,T7,T8,**T5,T8,T6,T8,T5) (**T1,T2,T4,**T5,T8,T7,T8,T6,T8,T5) (**T1,T2,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T5) (**T1,T4,**T5,T8,T7,T8,T6,T8,T5) (**T1,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T5) (**T1,T2,T4,**T5,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,T7,T8,**T5,T8,T6,T8,T7,T8,T5) (**T1,T4,**T5,T8,T6,T8,T7,T8,T5) (**T1,T4,T7,T8,**T5,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,**T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T4,**T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T7,T8,T5)
5	{d(*b*,T1,1), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T5,3)}	(T1,T2,T4,T5,T8,T6,T8,T5) (T1,T2,T4,T7,T8,T5,T8,T6,T8,T5) (T1,T4,T5,T8,T6,T8,T5) (T1,T4,T7,T8,T5,T8,T6,T8,T5) (T1,T2,T4,T5,T8,T7,T8,T6,T8,T5) (T1,T2,T4,T7,T8,T5,T8,T7,T8,T6,T8,T5) (T1,T4,T5,T8,T7,T8,T6,T8,T5) (T1,T4,T7,T8,T5,T8,T7,T8,T6,T8,T5) (T1,T2,T4,T5,T8,T6,T8,T7,T8,T5)

		(T1,T2,T4,T7,T8,T5,T8,T6,T8,T7,T8,T5) (T1,T4,T5,T8,T6,T8,T7,T8,T5) (T1,T4,T7,T8,T5,T8,T6,T8,T7,T8,T5) (T1,T2,T4,T5,T8,T7,T8,T6,T8,T7,T8,T5) (T1,T2,T4,T7,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5) (T1,T4,T5,T8,T7,T8,T6,T8,T7,T8,T5) (T1,T4,T7,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5)
6	{d(*d*,T6,1), u(*d*,T6,2), d(*b*,T6,4), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T5,3)}	(**T1,T2,T4,**T6,T8,T5,T8,T6,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T5,T8,T6,T8,T5) (**T1,T4,**T6,T8,T5,T8,T6,T8,T5) (**T1,T4,T7,T8,**T6,T8,T5,T8,T6,T8,T5) (**T1,T2,T4,**T6,T8,T7,T8,T5,T8,T6,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T7,T8,T5,T8,T6,T8,T5) (**T1,T4,**T6,T8,T7,T8,T5,T8,T6,T8,T5) (**T1,T4,T7,T8,**T6,T8,T7,T8,T5,T8,T6,T8,T5) (**T1,T2,T4,**T6,T8,T5,T8,T7,T8,T6,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T5,T8,T7,T8,T6,T8,T5) (**T1,T4,**T6,T8,T5,T8,T7,T8,T6,T8,T5) (**T1,T4,T7,T8,**T6,T8,T5,T8,T7,T8,T6,T8,T5) (**T1,T2,T4,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T5) (**T1,T4,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T5) (**T1,T4,T7,T8,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T5) (**T1,T2,T4,**T6,T8,T5,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T5,T8,T6,T8,T7,T8,T5) (**T1,T4,**T6,T8,T5,T8,T6,T8,T7,T8,T5) (**T1,T4,T7,T8,**T6,T8,T5,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,**T6,T8,T7,T8,T5,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T7,T8,T5,T8,T6,T8,T7,T8,T5) (**T1,T4,**T6,T8,T7,T8,T5,T8,T6,T8,T7,T8,T5) (**T1,T4,T7,T8,**T6,T8,T7,T8,T5,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,**T6,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T4,**T6,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T4,T7,T8,**T6,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T2,T4,T7,T8,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T4,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5) (**T1,T4,T7,T8,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T7,T8,T5)

The generated set of df-chains for TUT = T6

	Df	AP_{df}
1	{d(*b*,T1,1), u(*b*,T6,3)}	(T1,T2,T4,T6) (T1,T2,T4,T7,T8,T6) (T1,T4,T6) (T1,T4,T7,T8,T6)
2	{d(*w*,T5,1), u(*w*,T5,2),	(**T1,T2,T4,**T5,T8,T6)

	d(*b*,T5,4), u(*b*,T6,3)}	(**T1,T2,T4,T7,T8,**T5,T8,T6) (**T1,T4,**T5,T8,T6) (**T1,T4,T7,T8,**T5,T8,T6) (**T1,T2,T4,**T5,T8,T7,T8,T6) (**T1,T2,T4,T7,T8,**T5,T8,T7,T8,T6) (**T1,T4,**T5,T8,T7,T8,T5) (**T1,T4,T7,T8,**T5,T8,T7,T8,T6)
3	{d(*b*,T1,1), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T6,3)}	(T1,T2,T4,T5,T8,T6) (T1,T2,T4,T7,T8,T5,T8,T6) (T1,T4,T5,T8,T6) (T1,T4,T7,T8,T5,T8,T6) (T1,T2,T4,T5,T8,T7,T8,T6) (T1,T2,T4,T7,T8,T5,T8,T7,T8,T6) (T1,T4,T5,T8,T7,T8,T6) (T1,T4,T7,T8,T5,T8,T7,T8,T6)
4	{d(*d*,T6,1), u(*d*,T6,2), d(*b*,T6,4), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T6,3)}	(**T1,T2,T4,**T6,T8,T5,T8,T6) (**T1,T2,T4,T7,T8,**T6,T8,T5,T8,T6) (**T1,T4,T6,T8,**T5,T8,T6) (**T1,T4,T7,T8,T6,T8,**T5,T8,T6) (**T1,T2,T4,T6,T8,T7,T8,**T5,T8,T6) (**T1,T2,T4,T7,T8,T6,T8,T7,T8,**T5,T8,T6) (**T1,T4,T6,T8,T7,T8,**T5,T8,T6) (**T1,T4,T7,T8,T6,T8,T7,T8,**T5,T8,T6) (**T1,T2,T4,T6,T8,**T5,T8,T7,T8,T6) (**T1,T2,T4,T7,T8,T6,T8,**T5,T8,T7,T8,T6) (**T1,T4,T6,T8,**T5,T8,T7,T8,T6) (**T1,T4,T7,T8,T6,T8,**T5,T8,T7,T8,T6) (**T1,T2,T4,T6,T8,T7,T8,**T5,T8,T7,T8,T6) (**T1,T2,T4,T7,T8,T6,T8,T7,T8,**T5,T8,T7,T8,T6) (**T1,T4,T6,T8,T7,T8,**T5T8,T7,T8,T6) (**T1,T4,T7,T8,T6,T8,T7,T8,**T5,T8,T7,T8,T6)
5	{d(*b*,T1,1), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T6,3)}	(T1,T2,T4,T6,T8,T5,T8,T6) (T1,T2,T4,T7,T8,T6,T8,T5,T8,T6) (T1,T4,T6,T8,T5,T8,T6) (T1,T4,T7,T8,T6,T8,T5,T8,T6) (T1,T2,T4,T6,T8,T7,T8,T5,T8,T6) (T1,T2,T4,T7,T8,T6,T8,T7,T8,T5,T8,T6) (T1,T4,T6,T8,T7,T8,T5,T8,T6) (T1,T4,T7,T8,T6,T8,T7,T8,T5,T8,T6) (T1,T2,T4,T6,T8,T5,T8,T7,T8,T6) (T1,T2,T4,T7,T8,T6,T8,T5,T8,T7,T8,T6) (T1,T4,T6,T8,T5,T8,T7,T8,T6) (T1,T4,T7,T8,T6,T8,T5,T8,T7,T8,T6) (T1,T2,T4,T6,T8,T7,T8,T5,T8,T7,T8,T6) (T1,T2,T4,T7,T8,T6,T8,T7,T8,T5,T8,T7,T8,T6) (T1,T4,T6,T8,T7,T8,T5,T8,T7,T8,T6) (T1,T4,T7,T8,T6,T8,T7,T8,T5,T8,T7,T8,T6)
6	{d(*w*,T5,1), u(*w*,T5,2), d(*b*,T5,4), u(*b*,T6,3),	(**T1,T2,T4,**T5,T8,T6,T8,T5,T8,T6) (**T1,T2,T4,T7,T8,**T5,T8,T6,T8,T5,T8,T6)

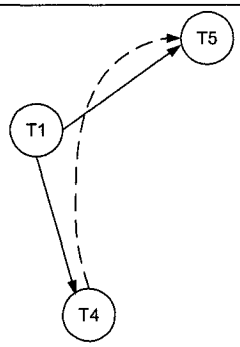
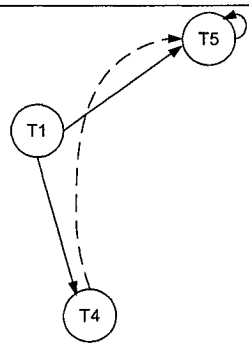
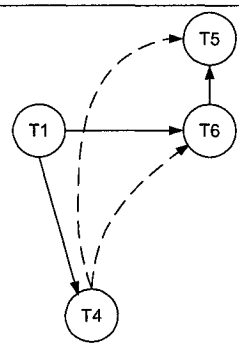
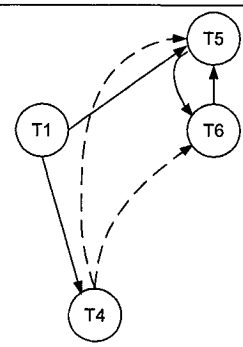
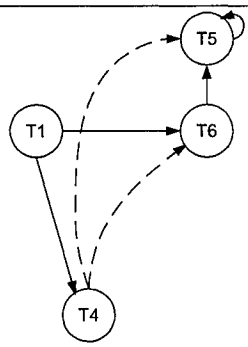
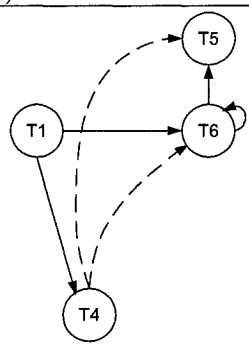
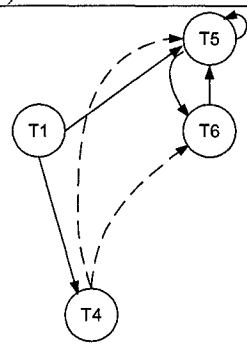
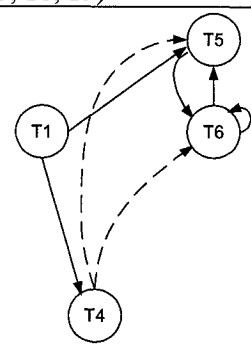
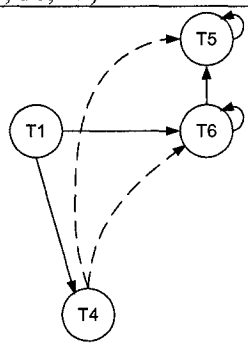
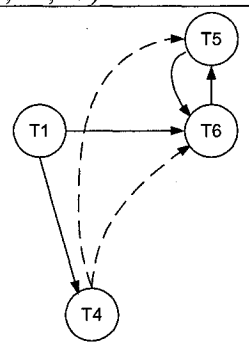
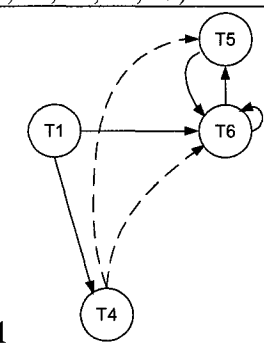
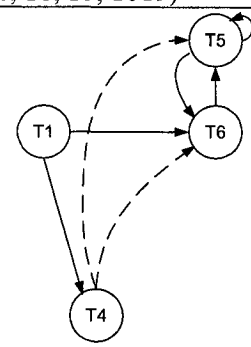
	d(*b*,T6,4), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T6,3)}	(**T1,T4,**T5,T8,T6,T8,T5,T8,T6) (**T1,T4,T7,T8,**T5,T8,T6,T8,T5,T8,T6) (**T1,T2,T4,**T5,T8,T7,T8,T6,T8,T5,T8,T6) (**T1,T2,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T5,T8,T6) (**T1,T4,**T5,T8,T7,T8,T6T8,T5,T8,T6) (**T1,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T5,T8,T6) (**T1,T2,T4,**T5,T8,T6,T8,T7,T8,T5,T8,T6) (**T1,T2,T4,T7,T8,**T5,T8,T6,T8,T7,T8,T5,T8,T6) (**T1,T4,**T5,T8,T6,T8,T7,T8,T5,T8,T6) (**T1,T4,T7,T8,**T5,T8,T6,T8,T7,T8,T5,T8,T6) (**T1,T2,T4,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T6) (**T1,T2,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T6) (**T1,T4,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T6) (**T1,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T6) (**T1,T2,T4,**T5,T8,T5,T8,T6,T8,T5,T8,T6) (**T1,T2,T4,T7,T8,**T5,T8,T6,T8,T5,T8,T7,T8,T6) (**T1,T4,T5,T8,T6,T8,**T5,T8,T7,T8,T6) (**T1,T4,T7,T8,**T5,T8,T6,T8,T5,T8,T7,T8,T6) (**T1,T2,T4,**T5,T8,T7,T8,T6,T8,T5,T8,T7,T8,T6) (**T1,T2,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T5,T8,T7,T8,T6) (**T1,T4,T5,T8,T7,T8,T6,T8,**T5,T8,T7,T8,T6) (**T1,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T5,T8,T7,T8,T6) (**T1,T2,T4,**T5,T8,T6,T8,T7,T8,T5,T8,T7,T8,T6) (**T1,T2,T4,T7,T8,**T5,T8,T6,T8,T7,T8,T5,T8,T7,T8,T6) (**T1,T4,T5,T8,T6,T8,T7,T8,**T5,T8,T7,T8,T6) (**T1,T4,T7,T8,**T5,T8,T6,T8,T7,T8,T5,T8,T7,T8,T6) (**T1,T2,T4,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T7,T8,T6) (**T1,T2,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T7,T8,T6) (**T1,T4,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T7,T8,T6) (**T1,T4,T7,T8,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T7,T8,T6)

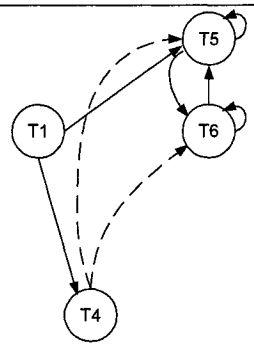
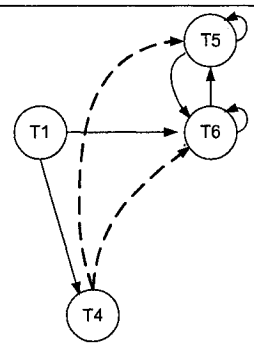
The generated set of df-chains for TUT = T7

	Df	AP_{df}
1	{d(*b*,T1,1), u(*b*,T7,1)}	(T1,T2,T4,T7) (T1,T4,T7)
2	{d(*w*,T5,1), u(*w*,T5,2), d(*b*,T5,4), u(*b*,T7,1)}	(**T1,T2,T4,**T5,T8,T7) (**T1,T2,T4,T6,T8,**T5,T8,T7) (**T1,T4,**T5,T8,T7) (**T1,T4,T6,T8,**T5,T8,T7)
3	{d(*b*,T1,1), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T7,1)}	(T1,T2,T4,T5,T8,T7) (T1,T2,T4,T7,T8,T5,T8,T7) (T1,T4,T5,T8,T7) (T1,T4,T7,T8,T5,T8,T7)
4	{d(*d*,T6,1), u(*d*,T6,2), d(*b*,T6,4), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T7,1)}	(**T1,T2,T4,**T6,T8,T5,T8,T7) (**T1,T4,**T6,T8,T5,T8,T7) (**T1,T2,T4,**T6,T8,T7,T8,T5,T8,T7)

		(**T1,T4,**T6,T8,T7,T8,T5,T8,T7)
5	{d(*b*,T1,1), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T7,1)}	(T1,T2,T4,T6,T8,T5,T8,T7) (T1,T2,T4,T7,T8,T6,T8,T5,T8,T7) (T1,T4,T6,T8,T5,T8,T7) (T1,T4,T7,T8,T6,T8,T5,T8,T7) (T1,T2,T4,T6,T8,T7,T8,T5,T8,T7) (T1,T2,T4,T7,T8,T6,T8,T7,T8,T5,T8,T7) (T1,T4,T6,T8,T7,T8,T5,T8,T7) (T1,T4,T7,T8,T6,T8,T7,T8,T5,T8,T7)
6	{d(*w*,T5,1), u(*w*,T5,2), d(*b*,T5,4), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T7,1)}	(**T1,T2,T4,**T5,T8,T6,T8,T5,T8,T7) (**T1,T4,**T5,T8,T6,T8,T5,T8,T7) (**T1,T2,T4,**T5,T8,T7,T8,T6,T8,T5,T8,T7) (**T1,T4,**T5,T8,T7,T8,T6,T8,T5,T8,T7) (**T1,T2,T4,**T5,T8,T6,T8,T7,T8,T5,T8,T7) (**T1,T4,**T5,T8,T6,T8,T7,T8,T5,T8,T7) (**T1,T2,T4,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T7) (**T1,T4,**T5,T8,T7,T8,T6,T8,T7,T8,T5,T8,T7)
7	{d(d,T6,1), u(d,T6,2), d(b,T6,4), u(b,T7,1)}	(**T1,T2,T4,**T6,T8,T7) (**T1,T2,T4,T5,T8,**T6,T8,T7) (**T1,T4,**T6,T8,T7) (**T1,T4,T5,T8,**T6,T8,T7)
8	{d(*b*,T1,1), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T7,1)}	(T1,T2,T4,T6,T8,T7) (T1,T2,T4,T7,T8,T6,T8,T7) (T1,T4,T6,T8,T7) (T1,T4,T7,T8,T6,T8,T7)
9	{d(*w*,T5,1), u(*w*,T5,2), d(*b*,T5,4), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T7,1)}	(**T1,T2,T4,**T5,T8,T6,T8,T7) (**T1,T4,**T5,T8,T6,T8,T7) (**T1,T2,T4,**T5,T8,T7,T8,T6,T8,T7) (**T1,T4,**T5,T8,T7,T8,T6,T8,T7)
10	{d(*b*,T1,1), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T7,1)}	(T1,T2,T4,T5,T8,T6,T8,T7) (T1,T2,T4,T7,T8,T5,T8,T6,T8,T7) (T1,T4,T5,T8,T6,T8,T7) (T1,T4,T7,T8,T5,T8,T6,T8,T7) (T1,T2,T4,T5,T8,T7,T8,T6,T8,T7) (T1,T2,T4,T7,T8,T5,T8,T7,T8,T6,T8,T7) (T1,T4,T5,T8,T7,T8,T6,T8,T7) (T1,T4,T7,T8,T5,T8,T7,T8,T6,T8,T7)
11	{d(*d*,T6,1), u(*d*,T6,2), d(*b*,T6,4), u(*b*,T5,3), d(*b*,T5,4), u(*b*,T6,3), d(*b*,T6,4), u(*b*,T7,1)}	(**T1,T2,T4,**T6,T8,T5,T8,T6,T8,T7) (**T1,T4,**T6,T8,T5,T8,T6,T8,T7) (**T1,T2,T4,**T6,T8,T7,T8,T5,T8,T6,T8,T7) (**T1,T4,**T6,T8,T7,T8,T5,T8,T6,T8,T7) (**T1,T2,T4,**T6,T8,T5,T8,T7,T8,T6,T8,T7) (**T1,T4,**T6,T8,T5,T8,T7,T8,T6,T8,T7) (**T1,T2,T4,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T7) (**T1,T4,**T6,T8,T7,T8,T5,T8,T7,T8,T6,T8,T7)

A.5 S_r generation for the ATM EFSM, Example of TUT =T5

 **1**	 **2**	 **3**	 **4**
(T1, T4, T5, T8, T9)	(T1, T4, T5, T8, T5, T8, T9)	(T1, T4, T6, T8, T5, T8, T9)	(T1, T4, T5, T8, T6, T8, T5, T8, T9)
 **5**	 **6**	 **7**	 **8**
(T1, T4, T6, T8, T5, T8, T5, T8, T9)	(T1, T4, T6, T8, T6, T8, T5, T8, T9)	(T1, T4, T5, T8, T5, T8, T6, T8, T5, T8, T9)	(T1, T4, T5, T8, T6, T8, T6, T8, T5, T8T9)
 **9**	 **10**	 **11**	 **12**
(T1, T4, T6, T8, T6, T8, T5, T8, T5, T8, T9)	(T1, T4, T6, T8, T5, T8, T6, T8, T5, T8, T9)	(T1, T4, T6, T8, T6, T8, T5, T8, T6, T8, T5, T8, T9)	(T1, T4, T6, T8, T5, T8, T5, T8, T6, T8, T5, T8, T9)

 **13**	 **14**
(T1, T4, T5, T8, T5, T8, T6, T8, T6, T8, T5, T8T9)	(T1, T4, T6, T8, T6, T8, T5, T8, T6, T8, T5, T8, T5, T8, T9)

A.6 TS_r generation for the ATM EFSM

* Infeasible test sequence

An example of the *TS* file: TS_r for TUT = T3

```
ATM_System
T3
Test Test_1 T1 T3*
Test Test_2 T1 T2 T3*
Test Test_3 T1 T2 T2 T3*
```

TS_r for TUT = T5

```
ATM_System
T5
Test Test_1 T1 T4 T5 T8 T9
Test Test_2 T1 T4 T5 T8 T5 T8 T9
Test Test_3 T1 T4 T6 T8 T5 T8 T9
Test Test_4 T1 T4 T5 T8 T6 T8 T5 T8 T9
Test Test_5 T1 T4 T6 T8 T5 T8 T5 T8 T9
Test Test_6 T1 T4 T6 T8 T6 T8 T5 T8 T9
Test Test_7 T1 T4 T5 T8 T6 T8 T5 T8 T5 T8 T9
Test Test_8 T1 T4 T5 T8 T6 T8 T6 T8 T5 T8 T9
Test Test_9 T1 T4 T6 T8 T6 T8 T5 T8 T5 T8 T9
Test Test_10 T1 T4 T6 T8 T5 T8 T6 T8 T5 T8 T9
Test Test_11 T1 T4 T6 T8 T6 T8 T5 T8 T6 T8 T5 T8 T9
Test Test_12 T1 T4 T6 T8 T5 T8 T5 T8 T6 T8 T5 T9
Test Test_13 T1 T4 T5 T8 T5 T8 T6 T8 T6 T8 T5 T8 T9
Test Test_14 T1 T4 T6 T8 T6 T8 T5 T8 T5 T8 T6 T8 T5 T8 T9
```

TS_r for TUT = T6

```
ATM_System
T6
Test Test_1 T1 T4 T6 T8 T9
Test Test_2 T1 T4 T6 T8 T6 T8 T9
Test Test_3 T1 T4 T5 T8 T6 T8 T9
```

```
Test Test_4 T1 T4 T6 T8 T5 T8 T6 T8 T9
Test Test_5 T1 T4 T5 T8 T6 T8 T6 T8 T9
Test Test_6 T1 T4 T5 T8 T5 T8 T6 T8 T9
Test Test_7 T1 T4 T6 T8 T5 T8 T6 T8 T6 T8 T9
Test Test_8 T1 T4 T6 T8 T5 T8 T5 T8 T6 T8 T9
Test Test_9 T1 T4 T5 T8 T5 T8 T6 T8 T6 T8 T9
Test Test_10 T1 T4 T5 T8 T6 T8 T5 T8 T6 T8 T9
Test Test_11 T1 T4 T5 T8 T5 T8 T6 T8 T5 T8 T6 T8 T9
Test Test_12 T1 T4 T5 T8 T6 T8 T6 T8 T5 T8 T6 T9
Test Test_13 T1 T4 T6 T8 T6 T8 T5 T8 T5 T8 T6 T8 T9
Test Test_14 T1 T4 T5 T8 T5 T8 T6 T8 T6 T8 T5 T8 T6 T8 T9
```

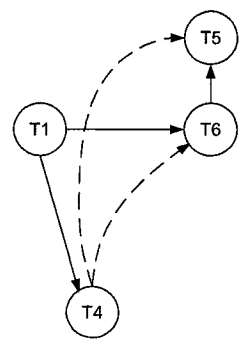
TS_r for TUT = T7

```
ATM_System
T7
Test Test_1 T1 T4 T7 T8 T9
Test Test_2 T1 T4 T5 T8 T7 T8 T9
Test Test_3 T1 T4 T6 T8 T5 T8 T7 T8 T9
Test Test_4 T1 T4 T7 T8 T5 T8 T7 T8 T9
Test Test_5 T1 T4 T6 T8 T7 T5 T8 T7 T8 T9
Test Test_6 T1 T4 T8 T6 T8 T5 T8 T7 T8 T9
Test Test_7 T1 T4 T7 T8 T6 T8 T7 T8 T5 T8 T7 T8 T9
Test Test_8 T1 T4 T5 T8 T6 T8 T5 T8 T7 T8 T9
Test Test_9 T1 T4 T5 T8 T7 T8 T6 T8 T7 T8 T5 T8 T7 T8 T9
Test Test_10 T1 T4 T6 T8 T7 T8 T9
Test Test_11 T1 T4 T5 T8 T6 T8 T7 T8 T9
Test Test_12 T1 T4 T7 T8 T6 T8 T7 T8 T9
Test Test_13 T1 T4 T5 T8 T7 T8 T6 T8 T7 T8 T9
Test Test_14 T1 T4 T7 T8 T5 T8 T6 T8 T7 T8 T9
Test Test_15 T1 T4 T7 T8 T5 T8 T7 T8 T6 T8 T7 T8 T9
Test Test_16 T1 T4 T6 T8 T5 T8 T6 T8 T7 T8
Test Test_17 T1 T4 T6 T8 T7 T8 T5 T8 T7 T8 T6 T8 T7 T8 T9
Test Test_18 T1 T4 T5 T8 T5 T8 T7 T8 T9
Test Test_19 T1 T4 T6 T8 T6 T8 T5 T8 T7 T8 T9
Test Test_20 T1 T4 T6 T8 T5 T8 T5 T8 T7 T8 T9
Test Test_21 T1 T4 T6 T8 T6 T8 T5 T8 T5 T8 T7 T8 T9
Test Test_22 T1 T4 T7 T8 T5 T8 T5 T8 T7 T8 T9
Test Test_23 T1 T4 T6 T8 T6 T8 T7 T5 T8 T7 T8 T9
Test Test_24 T1 T4 T6 T8 T7 T5 T8 T5 T8 T7 T8 T9
Test Test_25 T1 T4 T6 T8 T6 T8 T7 T5 T8 T5 T8 T7 T8 T9
Test Test_26 T1 T4 T8 T6 T8 T6 T8 T5 T8 T7 T8 T9
Test Test_27 T1 T4 T8 T6 T8 T5 T8 T5 T8 T7 T8 T9
Test Test_28 T1 T4 T8 T6 T8 T6 T8 T5 T8 T5 T8 T7 T8 T9
Test Test_29 T1 T4 T7 T8 T6 T8 T6 T8 T7 T8 T5 T8 T7 T8 T9
Test Test_30 T1 T4 T7 T8 T6 T8 T7 T8 T5 T8 T5 T8 T7 T8 T9
Test Test_31 T1 T4 T7 T8 T6 T8 T6 T8 T7 T8 T5 T8 T5 T8 T7 T8 T9
Test Test_32 T1 T4 T5 T8 T5 T8 T6 T8 T5 T8 T7 T8 T9
Test Test_33 T1 T4 T5 T8 T6 T8 T6 T8 T5 T8 T7 T8 T9
Test Test_34 T1 T4 T5 T8 T5 T8 T6 T8 T6 T8 T5 T8 T7 T8 T9
Test Test_35 T1 T4 T5 T8 T7 T8 T6 T8 T6 T8 T7 T8 T5 T8 T7 T8 T9
Test Test_36 T1 T4 T5 T8 T5 T8 T7 T8 T6 T8 T7 T8 T5 T8 T7 T8 T9
Test Test_37 T1 T4 T5 T8 T5 T8 T7 T8 T6 T8 T6 T8 T7 T8 T5 T8 T7 T8 T9
Test Test_38 T1 T4 T6 T8 T6 T8 T7 T8 T9
Test Test_39 T1 T4 T5 T8 T5 T8 T6 T8 T7 T8 T9
Test Test_40 T1 T4 T5 T8 T6 T8 T6 T8 T7 T8 T9
```

```
Test Test_41 T1 T4 T5 T8 T5 T8 T6 T8 T6 T8 T7 T8 T9
Test Test_42 T1 T4 T7 T8 T6 T8 T6 T8 T7 T8 T9
Test Test_43 T1 T4 T5 T8 T5 T8 T7 T8 T6 T8 T7 T8 T9
Test Test_44 T1 T4 T5 T8 T7 T8 T6 T8 T6 T8 T7 T8 T9
Test Test_45 T1 T4 T5 T8 T5 T8T7 T8 T6 T8 T6 T8 T7 T8 T9
Test Test_46 T1 T4 T7 T8 T5 T8 T5 T8 T6 T8 T7 T8 T9
Test Test_47 T1 T4 T7 T8 T5 T8 T6 T8 T6 T8 T7 T8 T9
Test Test_48 T1 T4 T7 T8 T5 T8 T5 T8 T6 T8 T6 T8 T7 T8 T9
Test Test_49 T1 T4 T7 T8 T5 T8 T5 T8 T7 T8 T6 T8 T7 T8 T9
Test Test_50 T1 T4 T7 T8 T5 T8 T7 T8 T6 T8 T6 T8 T7 T8 T9
Test Test_51 T1 T4 T7 T8 T5 T8 T5 T8 T7 T8 T6 T8 T6 T8 T7 T8 T9
Test Test_52 T1 T4 T6 T8 T6 T8 T5 T8 T6 T8 T7 T8
Test Test_53 T1 T4 T6 T8 T5 T8 T5 T8 T6 T8 T7 T8
Test Test_54 T1 T4 T6 T8 T6 T8 T5 T8 T5 T8 T6 T8 T7 T8
Test Test_55 T1 T4 T6 T8 T6 T8 T7 T8 T5 T8 T7 T8 T6 T8 T7 T8 T9
Test Test_56 T1 T4 T6 T8 T7 T8 T5 T8 T5 T8 T7 T8 T6 T8 T7 T8 T9
Test Test_57 T1 T4 T6 T8 T6 T8 T7 T8 T5 T8 T5 T8 T7 T8 T6 T8 T7 T8 T9
```

A.7 An example of the IP file for SIP # 3 for TUT=T5

```
ATM_System T5
 IP_T5_3
 df-chain
 (0,b,T1,1)(1,b,T6,3)
 (0,b,T6,4)(1,b,T5,3)
 Number of paths: 8
  T1,T2,T4,T6,T8,T5
  T1,T2,T4,T7,T8,T6,T8,T5
  T1,T4,T6,T8,T5
  T1,T4,T7,T8,T6,T8,T5
  T1,T2,T4,T6,T8,T7,T8,T5
  T1,T2,T4,T7,T8,T6,T8,T7,T8,T5
  T1,T4,T6,T8,T7,T8,T5
  T1,T4,T7,T8,T6,T8,T7,T8,T5
df-chain
 (0,d,T6,1)(1,d,T6,2)
 (0,b,T6,4)(1,b,T5,3)
 Number of paths: 8
      T1,T2,T4,T6,T8,T5
      T1,T2,T4,T7,T8,T6,T8,T5
      T1,T4,T6,T8,T5
      T1,T4,T7,T8,T6,T8,T5
      T1,T2,T4,T6,T8,T7,T8,T5
      T1,T2,T4,T7,T8,T6,T8,T7,T8,T5
      T1,T4,T6,T8,T7,T8,T5
      T1,T4,T7,T8,T6,T8,T7,T8,T5


  Node index: 0, Label: T1
  Node index: 1, Label: T4
    Number of incoming edges: 1
    List of edges: DType(Data=0,Control=1)
      (Source node index, DType)
```



```
    (0,0)
Node index: 2, Label: T5
  Number of incoming edges: 2
  List of edges: DType(Data=0,Control=1)
    (Source node index, DType)
    (3,0)
    (1,1)
Node index: 3, Label: T6
  Number of incoming edges: 2
  List of edges: DType(Data=0,Control=1)
    (Source node index, DType)
    (1,1)
    (0,0)
```

APPENDIX B: Vending Machine System (VMS)

B.1 Requirements of the VMS

The system sells three products: gum, M&Ms and beer which cost $.35, $.50 and $.80, respectively.

R = {r | r is a requirement which is a **transition**} or

R = {r | r is a requirement which is a **sequence of transitions**}.

R = {$r1$, $r2$, $r3$, $r4$, $r5$} for the Vending Machine EFSM where:

r1: The customer selects an item, deposits coins, and the machine dispenses the item only if it is available and if the payment is sufficient

$r1$ = (T3 or T4) x **T8**

r2: If the item selected is sold out, the machine displays an error message and prompts for item selection.

$r2$ = **T2**

r3: When the customer deposits coins, the machine validates them then displays the total amount inserted.

$r3$ = **T5**

r4: The machine returns the correct change when the customer deposits too much money and when coins for change are available. The machine then displays the change amount and the welcome message.

$r4$ = **T11**

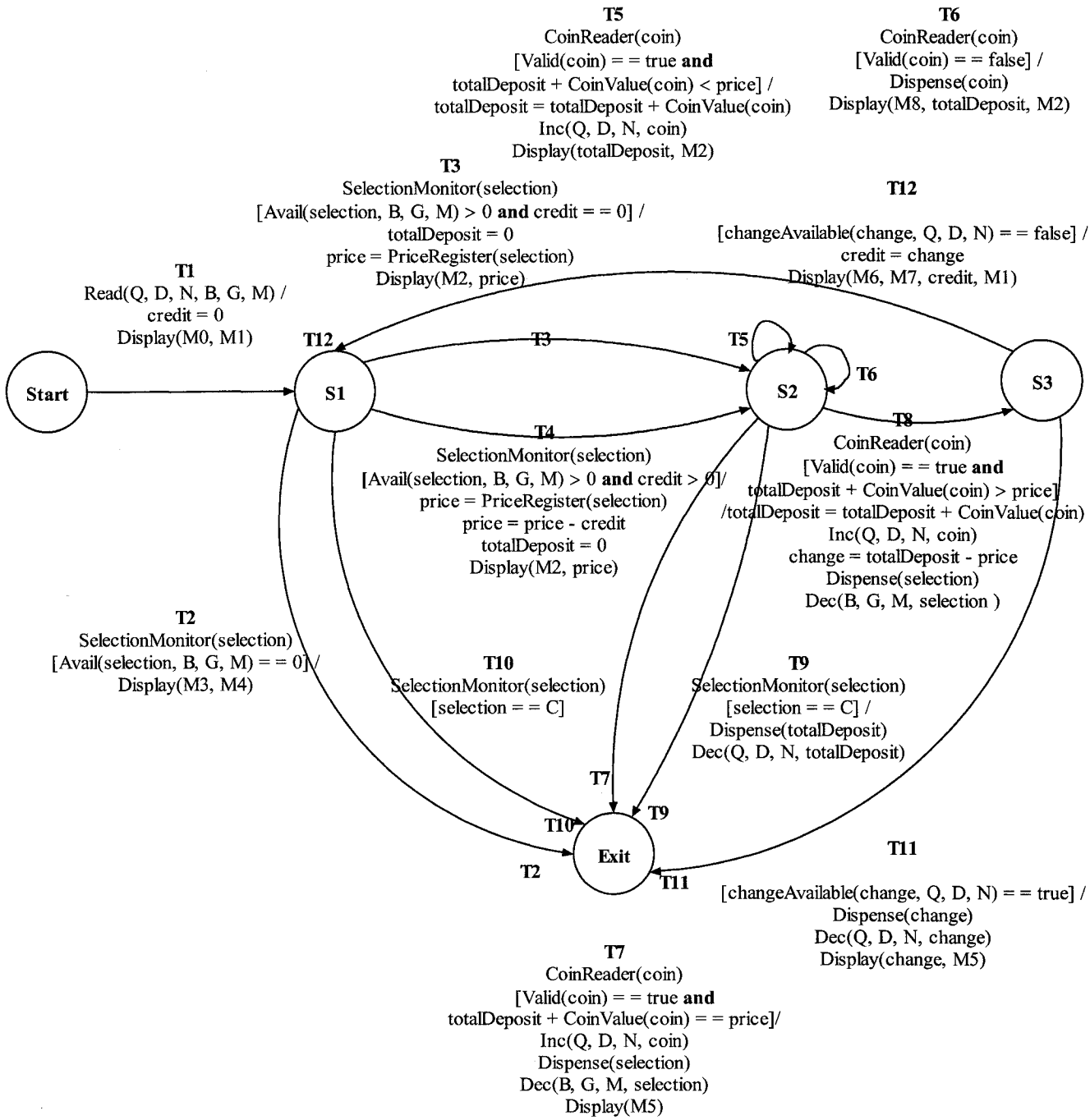
r5: If there are not enough coins for change in the machine, it will show a credit towards the next purchase and prompt for item selection.

$r5$ = **T12**

x : T5 may be inserted1 to 15 times

T# *is the transition under test (TUT)*

B.2 The EFSM model of the VMS



Symbol Definitions

- *M0*: message "Welcome! Buy something... Warning: If I do not have enough change I will dispense the product and give a credit for the next purchase".
- *M1*: message "Enter your selection: G, M, B, C to Cancel:".
- *M2*: message "Insert coins: Quarter, Dime, Niquel".
- *M3*: message "Sorry, item is sold out".
- *M4*: message "Make another selection".
- *M5*: message "Thank you!".
- *M6*: message "Sorry! I am out of change".
- *M7*: message "Credit: $".
- *M8*: message "Invalid Coin".
- *Q*: total number of quarters in the machine.
- *N*: total number of nickels in the machine.
- *D*: total number of dimes in the machine.
- *B*: total number of beers in the machine.
- *G*: total number of gums in the machine.
- *M*: total number of M&Ms in the machine.
- *price*: price of the selected item. $price \in \{35, 50, 80\}$
- *selection*: selection of the customer. $selection \in \{B, G, M, C\}$
- *totalDeposit*: total amount of money a customer deposits.
- *coin*: coin inserted by the customer. $coin \in \{Quarter, Dime, Nickel\}$
- *change*: amount to be returned when the total deposit is greater than the price.
- *credit*: amount of credit available for the customer's next purchase.

Inputs, Outputs, Functions, and Procedures

- *Read(Q, D, N, B, G, M)*: reads initial amounts of *Q, D, N, B, G, M.*
- *SelectionMonitor(selection)*: reads the selection made by the customer.
- *CoinReader(coin)*: reads the inserted *coin.*
- *Display(M)*: displays *M* on the machine's display panel.
- *Dispense(X)*: dispenses *X* to the customer. $X \in$ {*selection*, *totalDeposit*, *change*, *coin*}
- *Avail*(*selection, B, G, M*): returns the number of the selected item, *selection*, available in the machine.
- *PriceRegister*(*selection*): returns the price of the selected item, *selection.*
- *Valid(coin)*: returns *true* if *coin* $\in$ {*Quarter*, *Dime*, *Nickel*}.
- *CoinValue(coin)*: returns 25 or 10 or 5, the value of *coin.*
- *changeAvailable*(*change*, *Q*, *D*, *N*): returns *true* if the machine has enough coins to provide the correct change.
- *Inc*(*Q, D, N, coin*) : According to the type of *coin,* increments *Q*, *D*, or *N* by 1.
- *Dec*(*Q, D, N, X*) : According to the value of *X,* decrements *Q*, *D,* or *N* by the corresponding number of quarters, dimes and nickels in *X*. $X \in$ { *totalDeposit*, *change*}
- *Dec*(*B, G, M, selection*) : According to the type of *selection,* decrements *B*, *G*, or *M* by 1.

States

Start: Idle state.

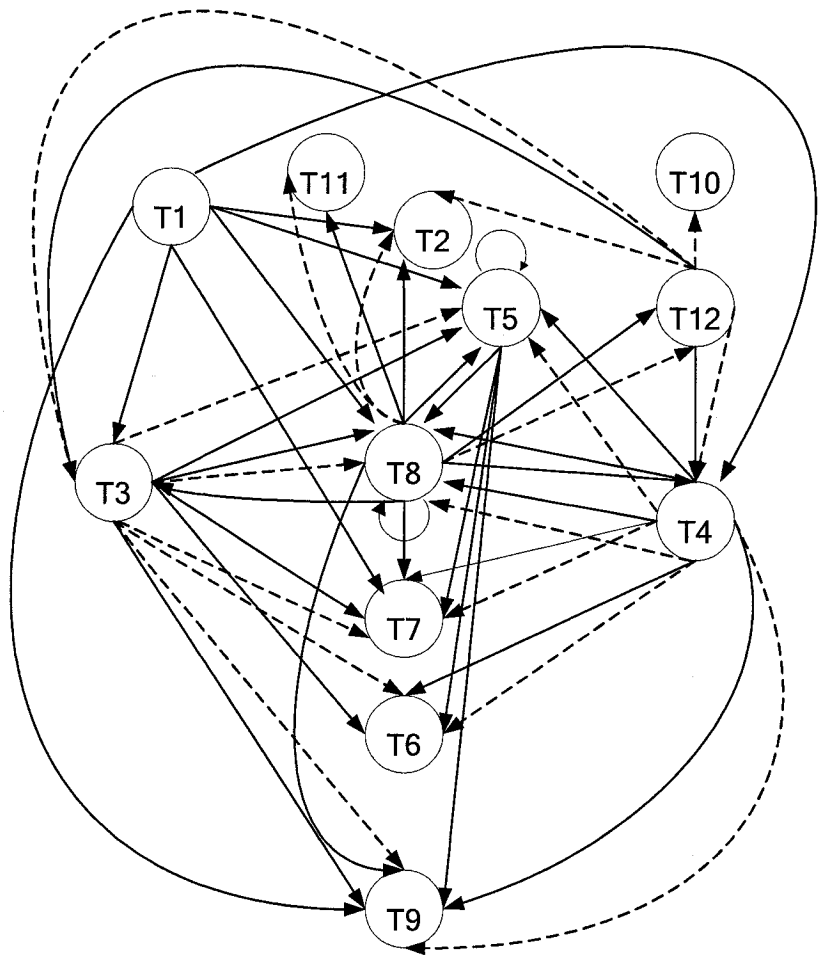
S1: Waiting for selection.

S2: Accepting coins.

S3: Making change.

Exit.

B.3 SDG of the VMS EFS



SDG of Vending Machine EFSM

data dependence edge

control dependence edge

Data dependence edges in the SDG of the VMS

	Transition	has a data dependence on	Identified by variables
1	T1	T2	*B, G, M*
2	T1	T3	*B, G, M, credit*
3	T1	T4	*B, G, M, credit*
4	T1	T5	*D, N, Q*
5	T1	T7	*B, D, G, M, N, Q*
6	T1	T8	*B, D, G, M, N, Q*
7	T1	T9	*D, N, Q*
8	T3	T5	*price, totalDeposit*
9	T3	T6	*totalDeposit*
10	T3	T7	*price, selection, totalDeposit*
11	T3	T8	*price, selection, totalDeposit*
12	T3	T9	*totalDeposit*
13	T4	T5	*price, totalDeposit*
14	T4	T6	*totalDeposit*
15	T4	T7	*price, selection, totalDeposit*
16	T4	T8	*price, selection, totalDeposit*
17	T4	T9	*totalDeposit*
18	T5	T5	*coin, D, N, Q, totalDeposit*
19	T5	T6	*totalDeposit*
20	T5	T7	*D, N, Q, totalDeposit*
21	T5	T8	*D, N, Q, totalDeposit*
22	T5	T9	*D, N, Q, totalDeposit*
23	T8	T2	*B, G, M*
24	T8	T3	*B, G, M*
25	T8	T4	*B, G, M*
26	T8	T5	*D, N, Q*
27	T8	T7	*B, D, G, M, N, Q*
28	T8	T8	*B, D, G, M, N, Q*
29	T8	T9	*D, N, Q*
30	T8	T11	*change, D, N, Q*
31	T8	T12	*change, D, N, Q*
32	T12	T3	*credit*
33	T12	T4	*credit*

Control dependence edges in the SDG of the VMS EFSM

	Transition	has a control dependence on
1	T3	T5
2	T3	T6
3	T3	T7
4	T3	T8
5	T3	T9
6	T4	T5
7	T4	T6
8	T4	T7
9	T4	T8
10	T4	T9
11	T8	T11
12	T8	T12
13	T12	T2
14	T12	T3
15	T12	T4
16	T12	T10

B.4 df-chains of the VMS

The set of df-chains generated for TUT = T2

	df	AP_{df}
1	{d(M,T01,1), u(*M*,T02,2)}	(T01,T02)
2	{d(*selection*,T03,1), u(*selection*,T08,24), d(*M*,T08,26), u(*M*,T02,2)}	(**T01,**T03,T08,T12,T02) (**T01,**T03,T05,T08,T12,T02) (**T01,**T03,T05,T06,T08,T12,T02) (**T01,**T03,T06,T08,T12,T02)
3	{d(*selection*,T04,1), u(*selection*,T08,24), d(*M*,T08,26), u(*M*,T02,2)}	(**T01,**T04,T08,T12,T02) (**T01,**T04,T05,T08,T12,T02) (**T01,**T04,T05,T06,T08,T12,T02) (**T01,**T04,T06,T08,T12,T02)
4	{d(*M*,T01,1), u(*M*,T08,25), d(*M*,T08,26), u(*M*,T02,2)}	(T01,T03,T08,T12,T02) (T01,T03,T05,T08,T12,T02) (T01,T03,T05,T06,T08,T12,T02) (T01,T03,T06,T08,T12,T02) (T01,T04,T08,T12,T02) (T01,T04,T05,T08,T12,T02) (T01,T04,T05,T06,T08,T12,T02) (T01,T04,T06,T08,T12,T02)
5	{d(*G*,T01,2), u(*G*,T02,3)}	(T01,T02)
6	{d(*selection*,T03,1), u(*s1election*,T08,21),	(**T01,**T03,T08,T12,T02)

	d(*G*,T08,23), u(*G*,T02,3)}	(**T01,**T03,T05,T08,T12,T02) (**T01,**T03,T05,T06,T08,T12,T02) (**T01,**T03,T06,T08,T12,T02)
7	{d(*selection*,T04,1), u(*selection*,T08,21) d(*G*,T08,23), u(*G*,T02,3)}	(**T01,**T04,T08,T12,T02) (**T01,**T04,T05,T08,T12,T02) (**T01,**T04,T05,T06,T08,T12,T02) (**T01,**T04,T06,T08,T12,T02)
8	{d(*G*,T01,2), u(*G*,T08,22), d(*G*,T08,23), u(*G*,T02,3)}	(T01,T03,T08,T12,T02) (T01,T03,T05,T08,T12,T02) (T01,T03,T05,T06,T08,T12,T02) (T01,T03,T06,T08,T12,T02) (T01,T04,T08,T12,T02) (T01,T04,T05,T08,T12,T02) (T01,T04,T05,T06,T08,T12,T02) (T01,T04,T06,T08,T12,T02)

The set of df-chains generated for TUT = T5

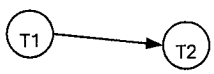
	Df	$\|AP_{df}\|$
1	{d(*selection*,T03,1), u(*selection*,T03,8), d(*price*,T03,9), u(*price*,T05,2)}	5
2	{d(*credit*,T01,7), u(*credit*,T04,9), d(*price*,T04,11), u(*price*,T05,2)}	5
3	{d(*selection*,T03,1), u(*selection*,T03,8), d(*price*,T03,9), u(*price*,T08,14), d(*change*,T08,16), u(*change*,T12,5), d(*credit*,T12,6), u(*credit*,T04,9), d(*price*,T04,11), u(*price*,T05,2)}	4
4	{d(*selection*,T04,1), u(*selection*,T04,7), u(*price*,T04,8)(*price*,T04,10), d(*price*,T04,11), u(*price*,T08,14), d(*change*,T08,16), u(*change*,T12,5), d(*credit*,T12,6), u(*credit*,T04,9), d(*price*,T04,11), u(*price*,T05,2)}	4
5	{d(*totalDeposit*,T04,12), u(*totalDeposit*,T08,15), d(*change*,T08,16), u(*change*,T12,5), d(*credit*,T12,6), u(*credit*,T04,9), d(*price*,T04,11), u(*price*,T05,2)}	2
6	{d(*coin*,T05,1)(*coin*,T05,5), u(*totalDeposit*,T05,7), d(*totalDeposit*,T08,15), u(*change*,T08,16), d(*change*,T12,5), u(*credit*,T12,6), d(*credit*,T04,9), u(*price*,T04,11), d(*price*,T05,2)}	4
7	{d(*totalDeposit*,T03,7), u(*totalDeposit*,T05,6), d(*totalDeposit*,T05,7), u(*totalDeposit*,T08,15), d(*change*,T08,16), u(*change*,T12,5), d(*credit*,T12,6), u(*credit*,T04,9), d(*price*,T04,11), u(*price*,T05,2)}	2
8	{d(*totalDeposit*,T04,12), u(*totalDeposit*,T05,6), d(*totalDeposit*,T05,7), u(*totalDeposit*,T08,15), d(*change*,T08,16), u(*change*,T12,5), d(*credit*,T12,6), u(*credit*,T04,9), d(*price*,T04,11), u(*price*,T05,2)}	2
9	{d(*selection*,T04,1), u(*selection*,T04,7), d(*price*,T04,8), u(*price*,T04,10), d(*price*,T04,11), u(*price*,T05,2)}	5
10	{d(*totalDeposit*,T03,7), u(*totalDeposit*,T05,3)}	5
11	{d(*totalDeposit*,T04,12), u(*totalDeposit*,T05,3)}	5
12	{d(*totalDeposit*,T04,12), u(*totalDeposit*,T05,6)}	5
13	{d(*Q*,T01,6), u(*Q*,T05,9)}	2
14	{d(*coin*,T08,1), u(*coin*,T08,5), d(*Q*,T08,7), u(*Q*,T05,9)}	8
15	{d(*Q*,T01,6), u(*Q*,T08,6), d(*Q*,T08,7), u(*Q*,T05,9)}	8

16	{d(*coin*,T05,1), u(1,*coin*,T05,8), d(0,*Q*,T05,10), u(1,*Q*,T08,6), d(0,*Q*,T08,7), u(1,*Q*,T05,9)}	16
17	{d(*Q*,T01,6), u(*Q*,T05,9), d(*Q*,T05,10), u(*Q*,T08,6), d(*Q*,T08,7), u(*Q*,T05,9)}	8
18	{d(*coin*,T08,1), u(*coin*,T08,5), d(*Q*,T08,7), u(*Q*,T05,9), d(*Q*,T05,10), u(*Q*,T08,6), d(*Q*,T08,7), u(*Q*,T05,9)}	32
19	{d(*coin*,T08,1), u(*coin*,T08,8), d(*D*,T08,10), u(*D*,T05,12)}	8
20	{d(*D*,T01,5), u(*D*,T08,9), d(*D*,T08,10), u(*D*,T05,12)}	8
21	{d(*coin*,T05,1), u(*coin*,T05,11), d(*D*,T05,13), u(*D*,T08,9), d(*D*,T08,10), u(*D*,T05,12)}	16
22	{d(*D*,T01,5), u(*D*,T05,12), d(*D*,T05,13), u(*D*,T08,9), d(*D*,T08,10), u(*D*,T05,12)}	8
23	{d(*coin*,T08,1), u(*coin*,T08,8), d(*D*,T08,10), u(*D*,T05,12), d(*D*,T05,13), u(*D*,T08,9), d(*D*,T08,10), u(*D*,T05,12)}	32
24	{d(*N*,T01,4), u(*N*,T05,15)}	2
25	{d(*coin*,T08,1), u(*coin*,T08,11), d(*N*,T08,13), u(*N*,T05,15)}	8
26	{d(*N*,T01,4), u(*N*,T08,12), d(*N*,T08,13), u(*N*,T05,15)}	8
27	{d(*N*,T01,4), u(*N*,T05,15), d(*N*,T05,16), d(*N*,T08,12), u(*N*,T08,13), u(*N*,T05,15)}	8
28	{d(*coin*,T05,1), u(*coin*,T05,14), d(*N*,T05,16), u(*N*,T08,12), d(*N*,T08,13), u(*N*,T05,15)}	16
29	{d(*coin*,T08,1), u(*coin*,T08,11), d(*N*,T08,13), u(*N*,T05,15), d(*N*,T05,16), u(*N*,T08,12), d(*N*,T08,13), u(*N*,T05,15)}	32

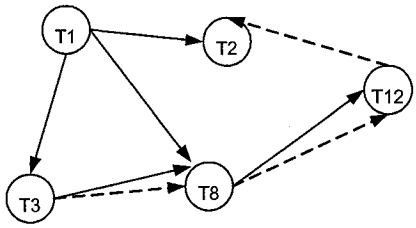
B.5 S_r generation for $r \in R$, R = {T2, T5, T8, T11, T12}

TUT	$\lvert S_{r\_elem} \rvert$	$\lvert S_r \rvert$	$\lvert TS_r \rvert$
T2	5	7	7
T5	20	40	40
T8	16	28	28
T11	14	24	24
T12	14	24	24

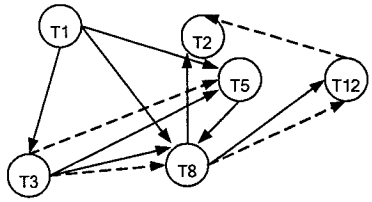
S_r for r = T2



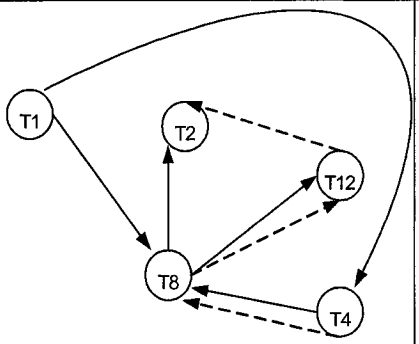
SIP_0



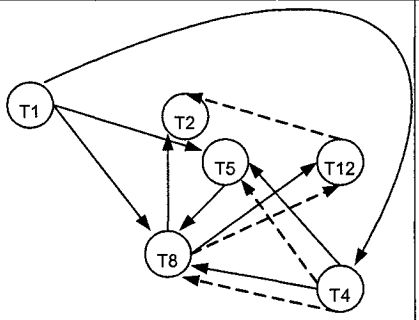
SIP_1



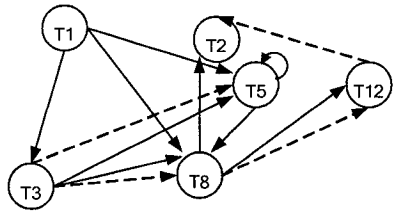
SIP_2



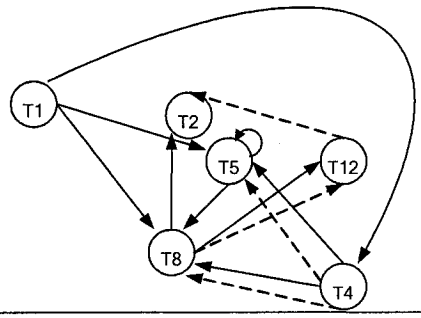
SIP_3



SIP_4

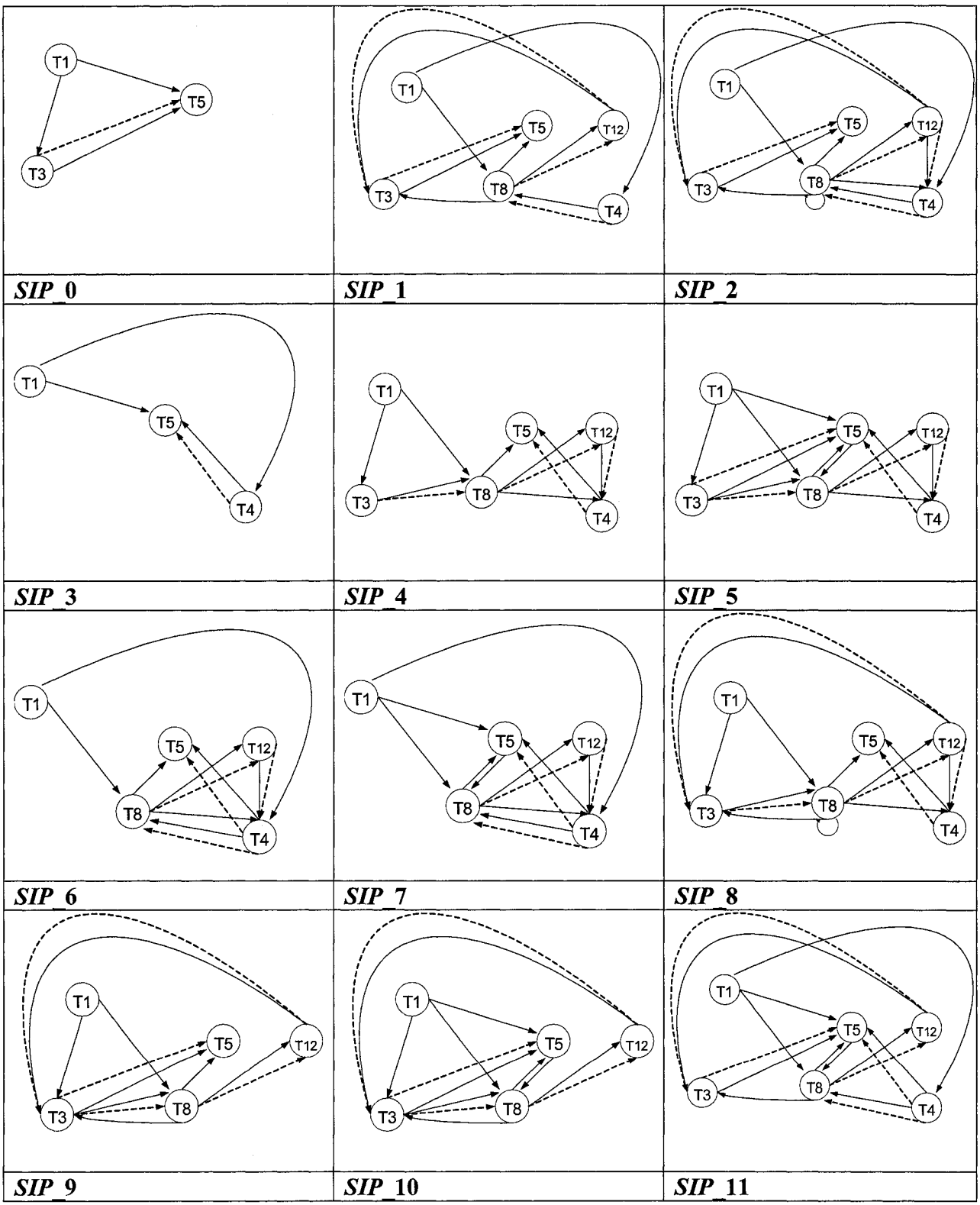


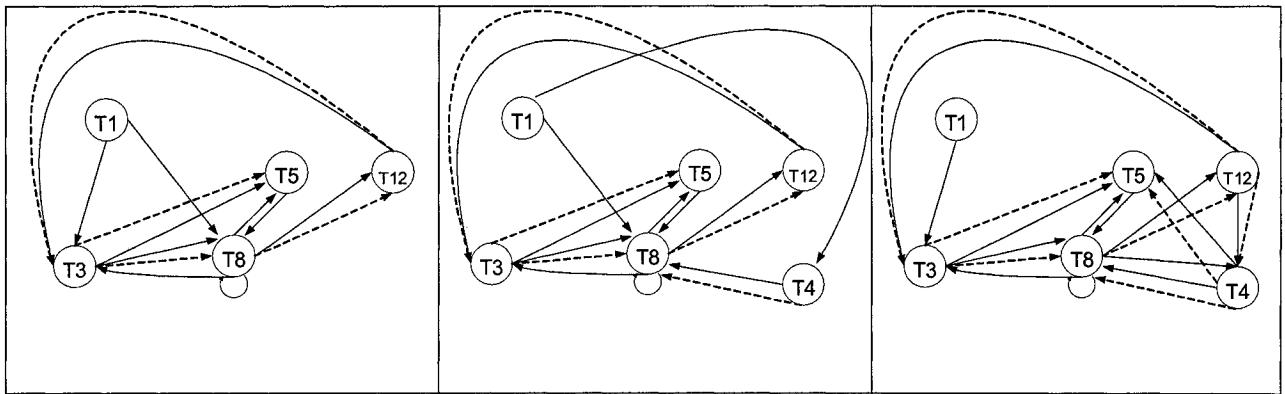
SIP_5



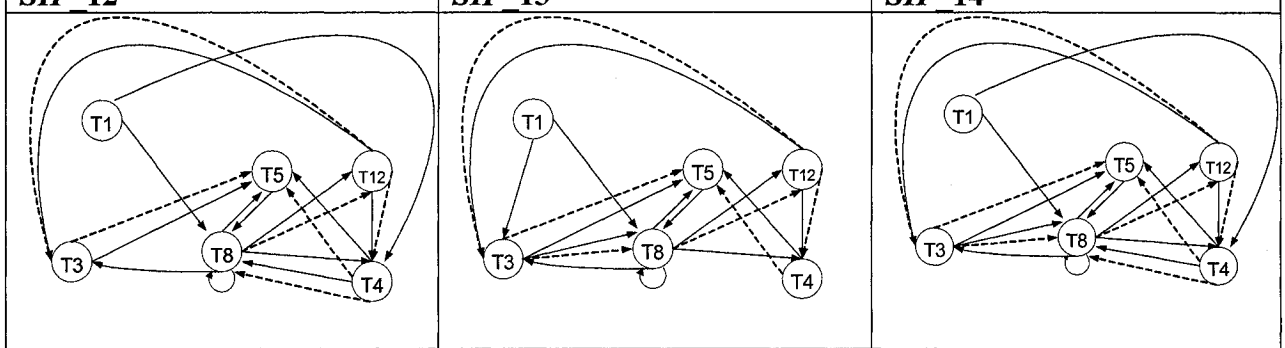
SIP_6

S_r for r = T5

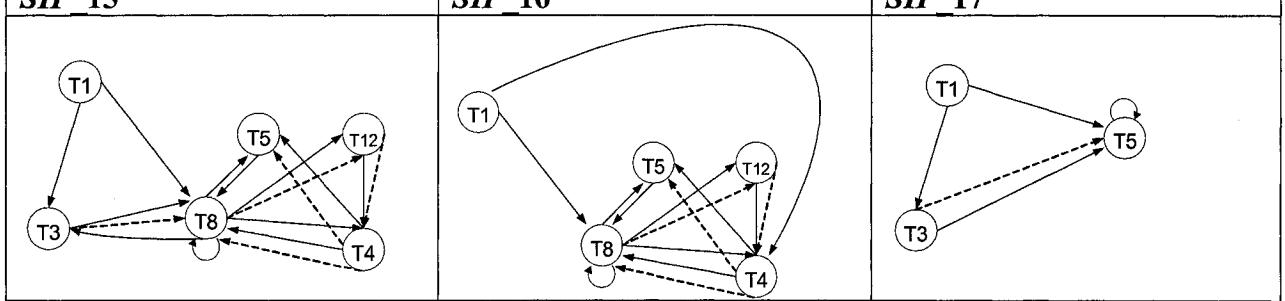

T1
T5
T3
SIP_0
T1
T5
T12
T3
T8
T4
SIP_1
T1
T5
T12
T3
T8
T4
SIP_2
T1
T5
T3
T4
SIP_3
T1
T5
T12
T3
T8
T4
SIP_4
T1
T5
T12
T3
T8
T4
SIP_5
T1
T5
T12
T8
T4
SIP_6
T1
T5
T12
T8
T4
SIP_7
T1
T5
T12
T3
T8
T4
SIP_8
T1
T5
T12
T3
T8
SIP_9
T1
T5
T12
T3
T8
SIP_10
T1
T5
T12
T3
T8
T4
SIP_11



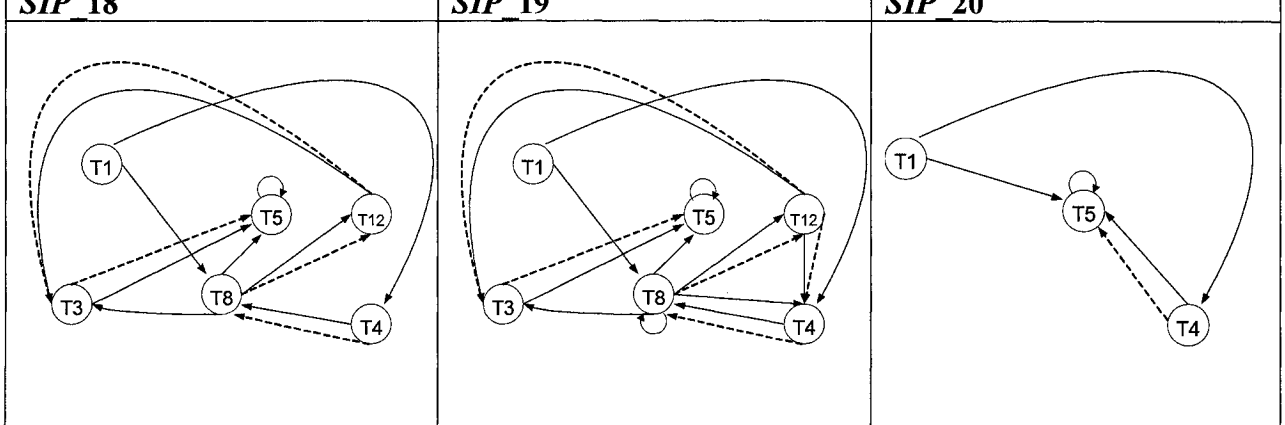
***SIP*_12** | ***SIP*_13** | ***SIP*_14**



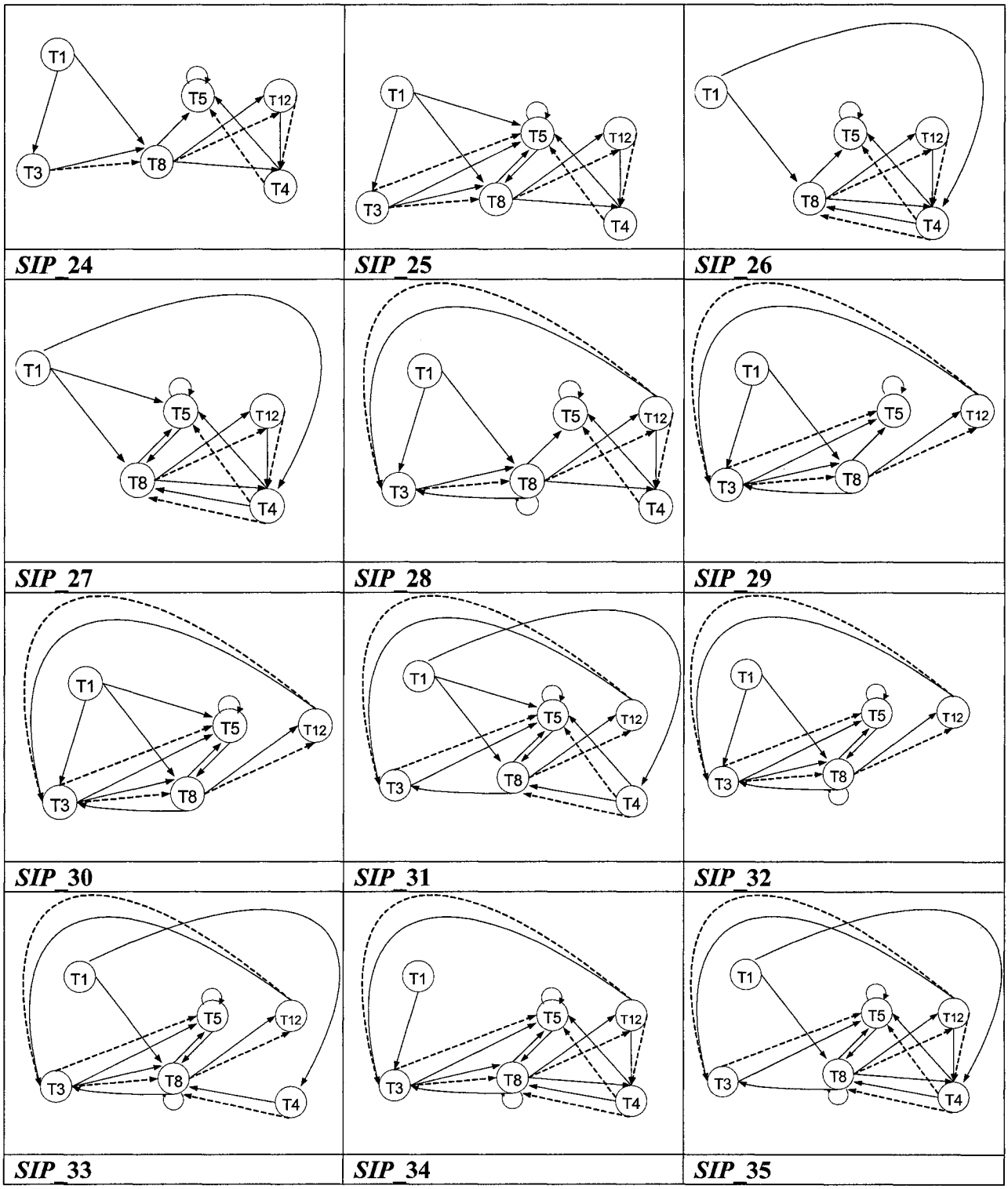
***SIP*_15** | ***SIP*_16** | ***SIP*_17**

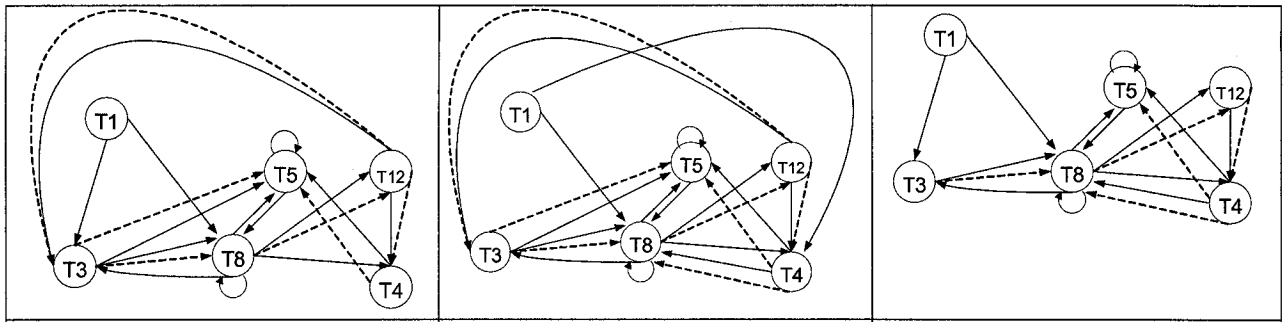


***SIP*_18** | ***SIP*_19** | ***SIP*_20**

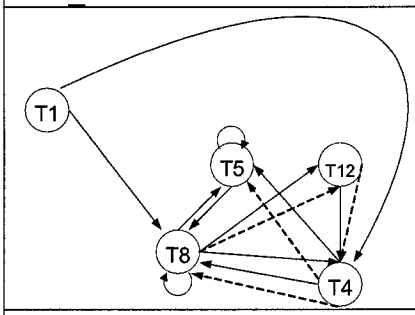


***SIP*_21** | ***SIP*_22** | ***SIP*_23**


T1
T5
T12
T3
T8
T4
SIP_24
SIP_25
SIP_26
SIP_27
SIP_28
SIP_29
SIP_30
SIP_31
SIP_32
SIP_33
SIP_34
SIP_35

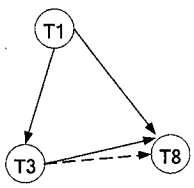


***SIP*_36** | ***SIP*_37** | ***SIP*_38**

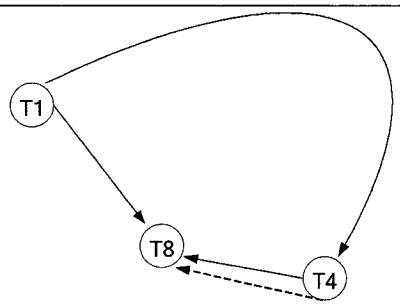


***SIP*_39**

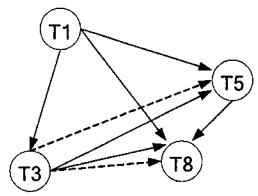
S_r for r = T8



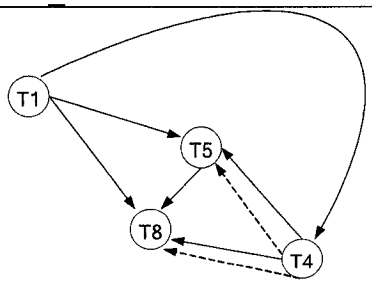
***SIP*_0**



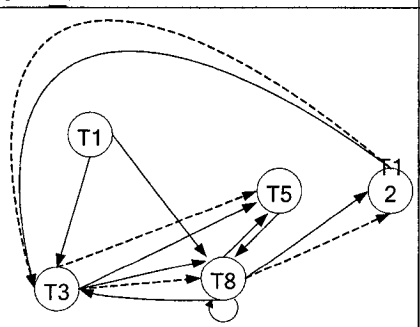
***SIP*_1**



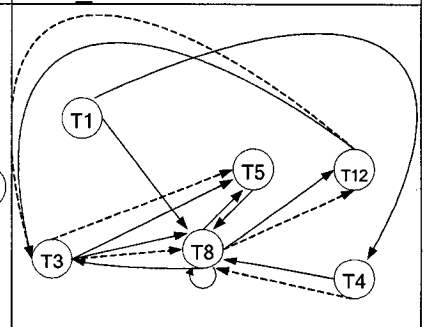
***SIP*_2**



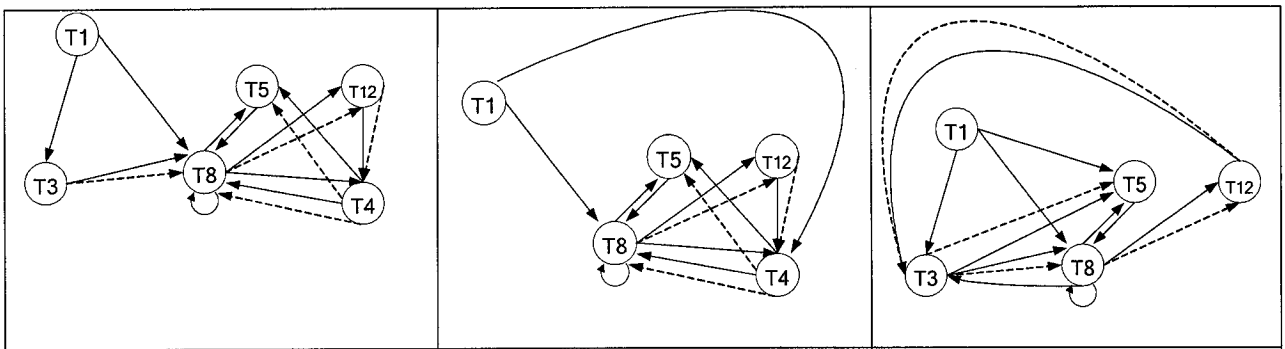
***SIP*_3**



***SIP*_4**



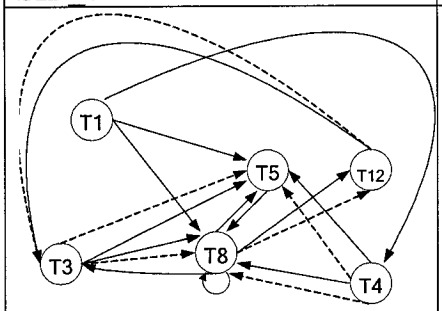
***SIP*_5**



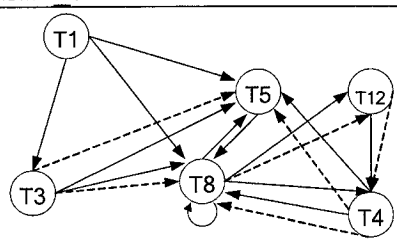
SIP_6

SIP_7

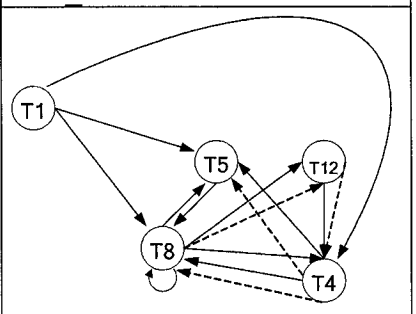
SIP_8



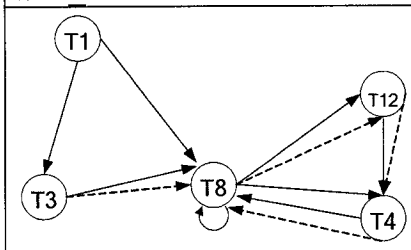
SIP_9



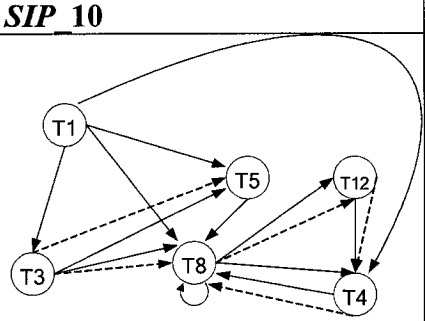
SIP_10



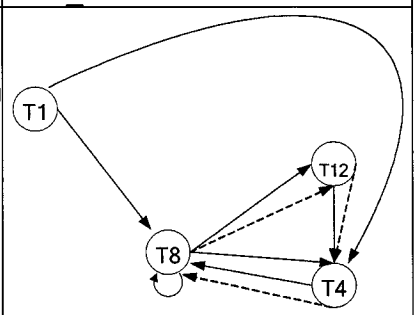
SIP_11



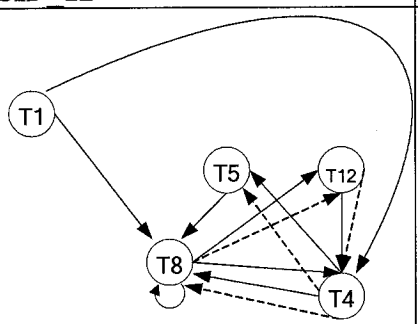
SIP_12



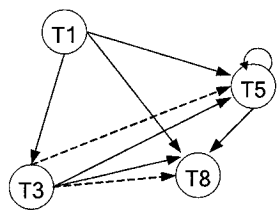
SIP_13



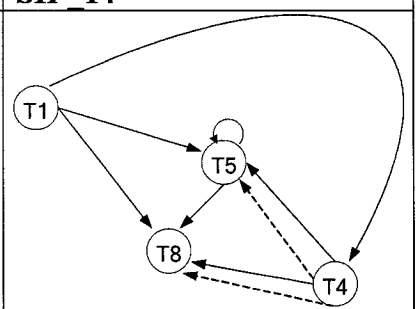
SIP_14



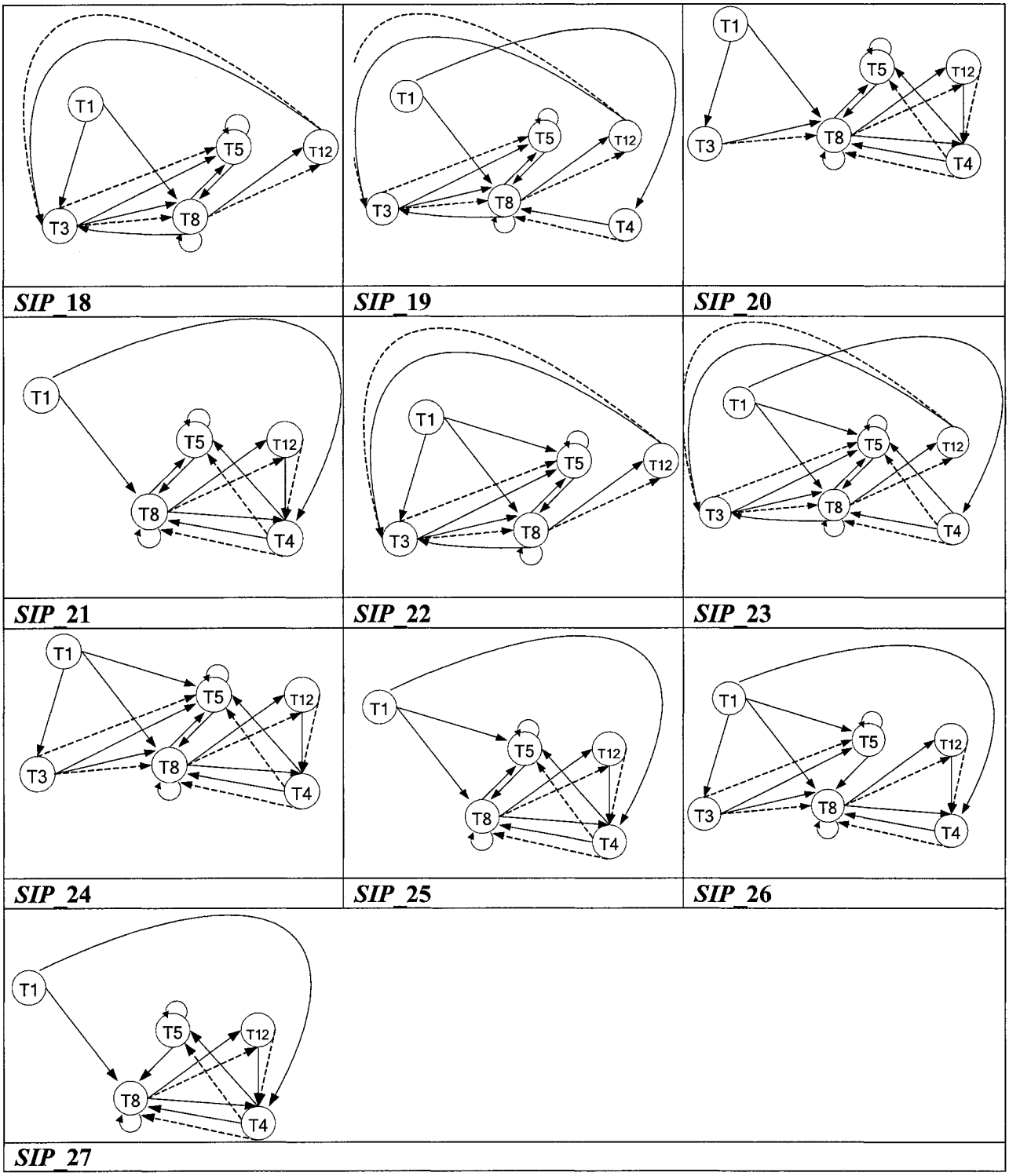
SIP_15



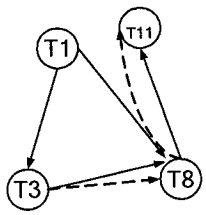
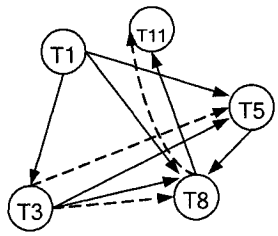
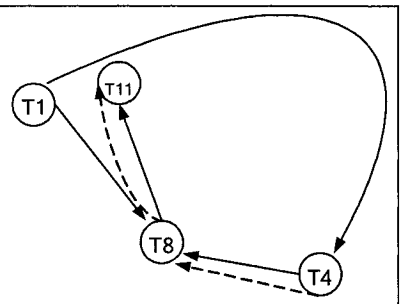
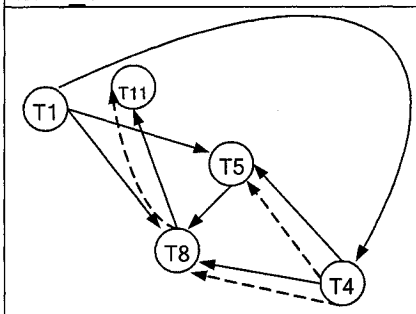
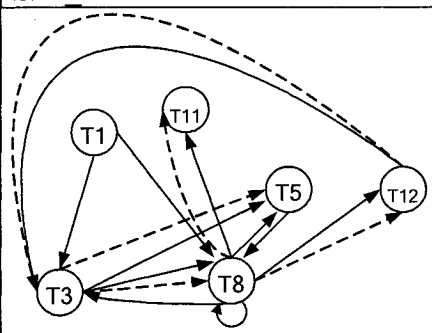
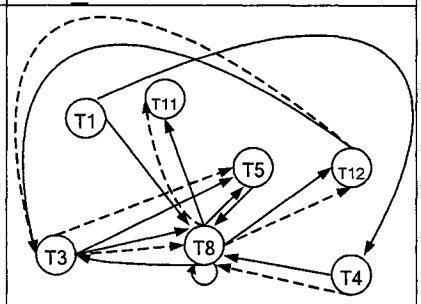
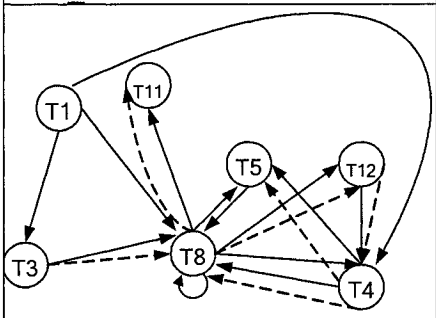
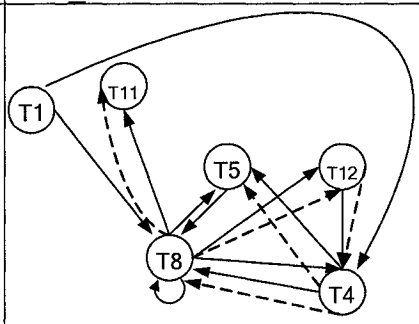
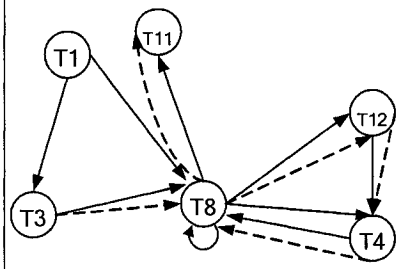
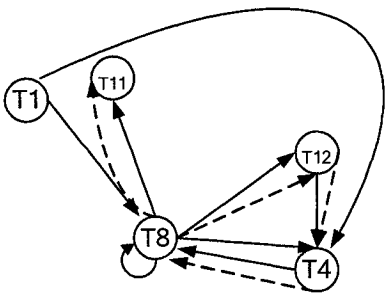
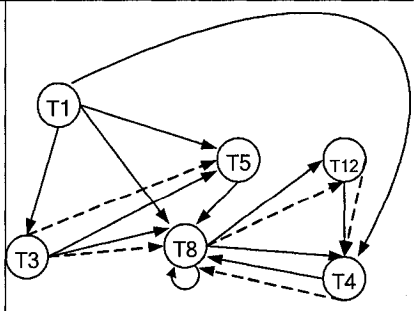
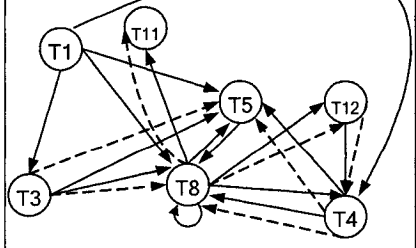
SIP_16

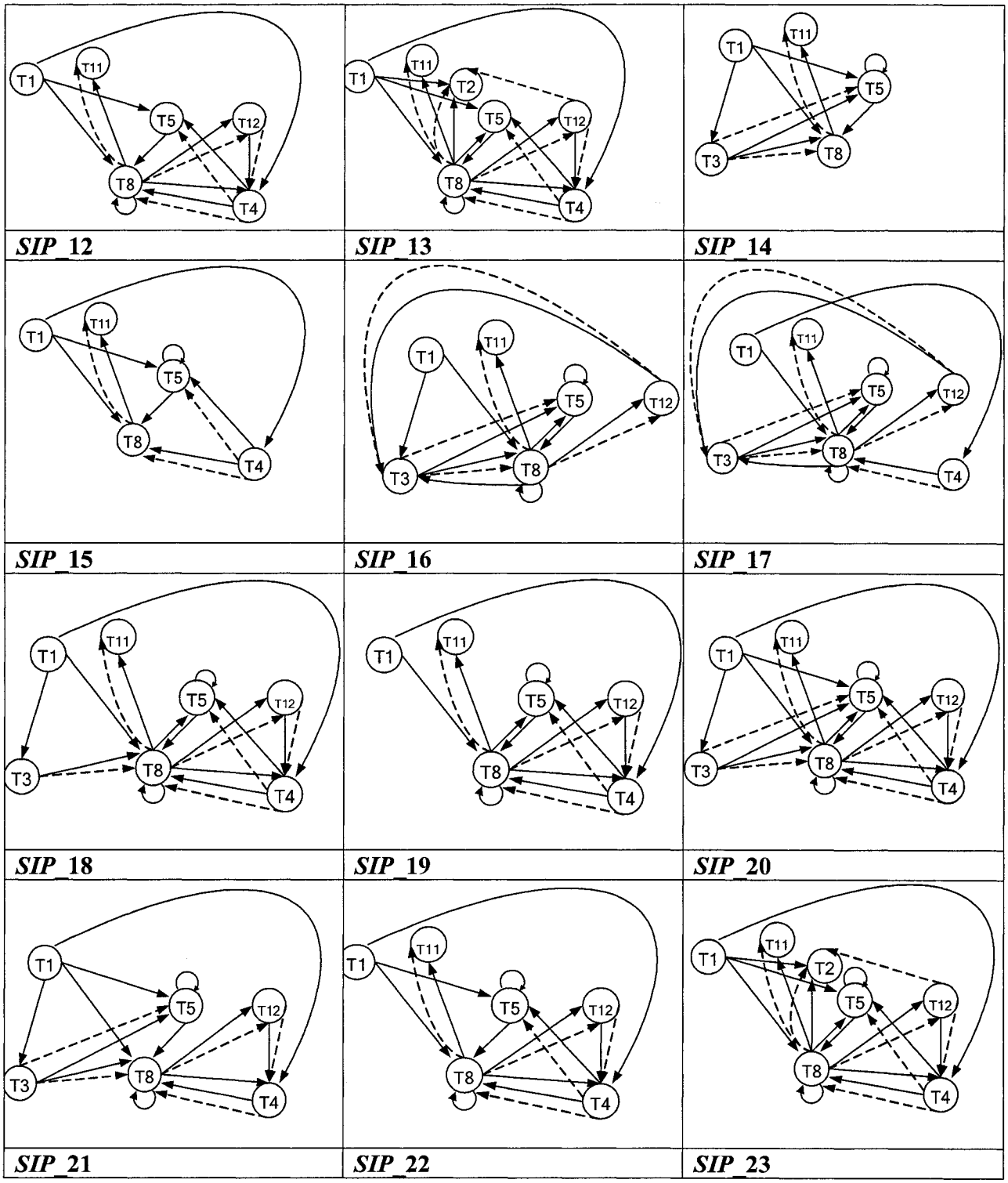


SIP_17

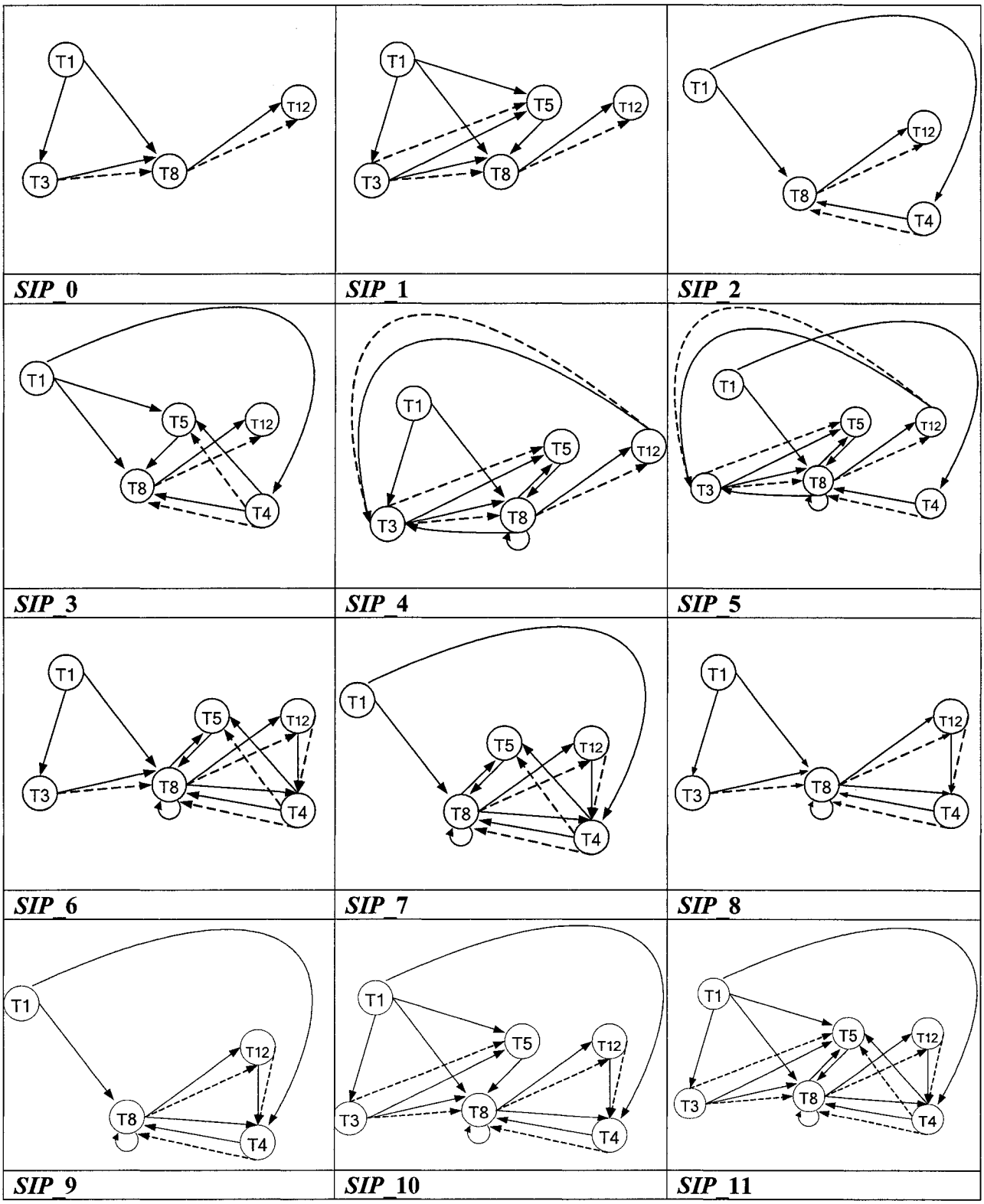

T1
T5
T12
T3
T8
T4
SIP_18
SIP_19
SIP_20
SIP_21
SIP_22
SIP_23
SIP_24
SIP_25
SIP_26
SIP_27

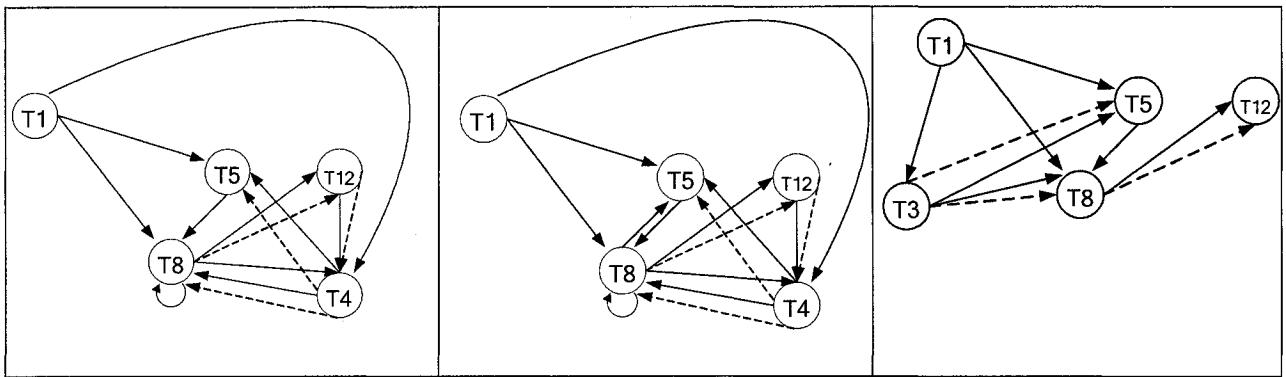
S_r for r = T11

		
SIP_0	***SIP_1***	***SIP_2***
		
SIP_3	***SIP_4***	***SIP_5***
		
SIP_6	***SIP_7***	***SIP_8***
		
SIP_9	***SIP_10***	***SIP_11***

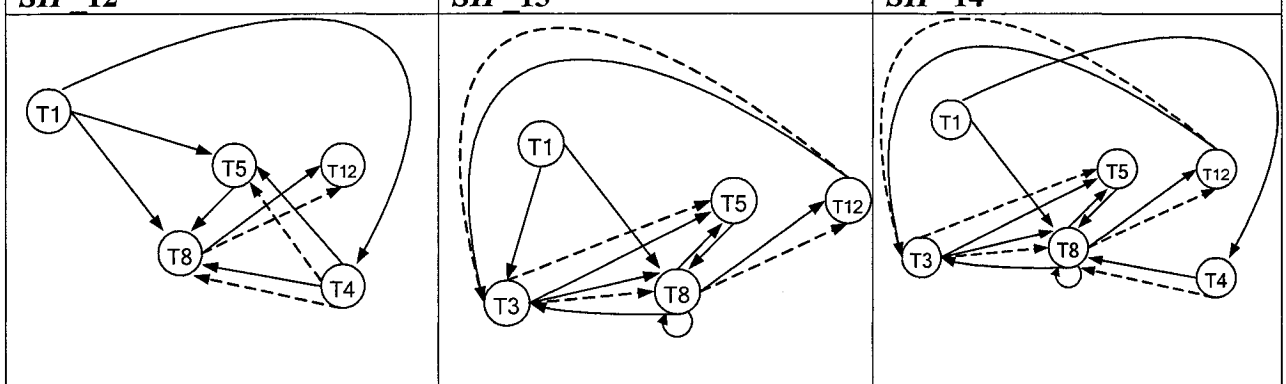

T1
T11
T5
T12
T8
T4
SIP_12
T1
T11
T2
T5
T12
T8
T4
SIP_13
T1
T11
T5
T3
T8
SIP_14
T1
T11
T5
T8
T4
SIP_15
T1
T11
T5
T12
T3
T8
T4
SIP_16
T1
T11
T5
T12
T3
T8
T4
SIP_17
T1
T11
T5
T12
T3
T8
T4
SIP_18
T1
T11
T5
T12
T8
T4
SIP_19
T1
T11
T5
T12
T3
T8
T4
SIP_20
T1
T11
T5
T12
T3
T8
T4
SIP_21
T1
T11
T2
T5
T12
T8
T4
SIP_22
T1
T11
T2
T5
T12
T8
T4
SIP_23

S_r for r = T12

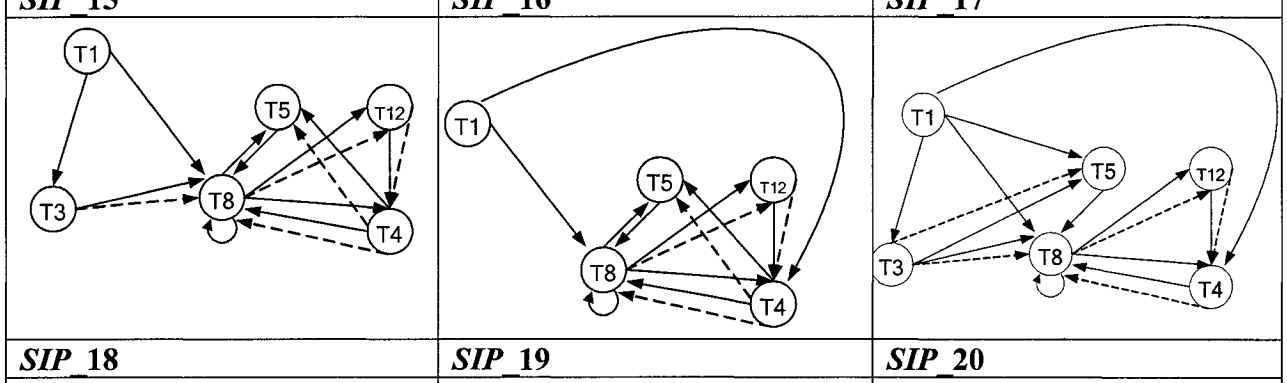

T1
T12
T3
T8
T1
T5
T12
T3
T8
T1
T12
T8
T4
SIP_0
SIP_1
SIP_2
T1
T5
T12
T8
T4
T1
T5
T12
T3
T8
T1
T5
T12
T3
T8
T4
SIP_3
SIP_4
SIP_5
T1
T5
T12
T3
T8
T4
T1
T5
T12
T8
T4
T1
T12
T3
T8
T4
SIP_6
SIP_7
SIP_8
T1
T12
T8
T4
T1
T5
T12
T3
T8
T4
T1
T5
T12
T3
T8
T4
SIP_9
SIP_10
SIP_11



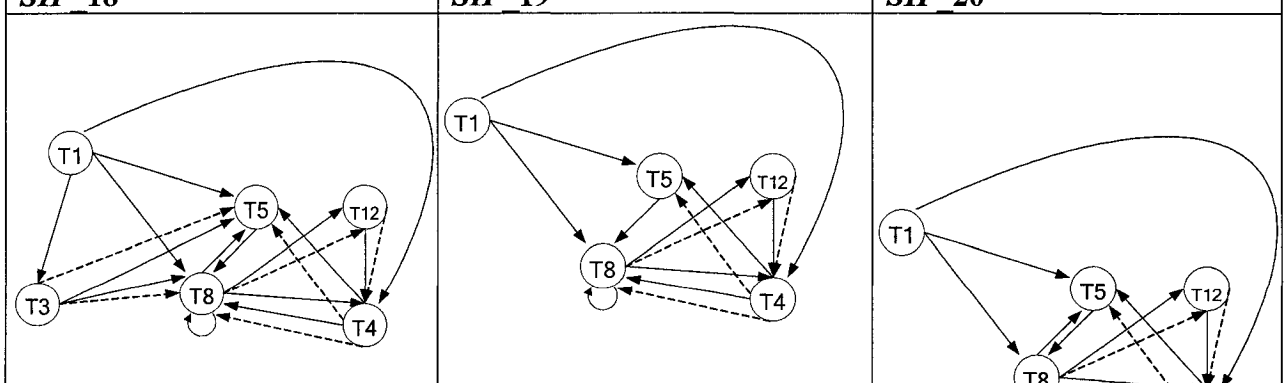
SIP_12 ***SIP_13*** ***SIP_14***



SIP_15 ***SIP_16*** ***SIP_17***



SIP_18 ***SIP_19*** ***SIP_20***



SIP_21 ***SIP_22*** ***SIP_23***

B.6 TS_r generation for $r \in R$, R = {T2, T5, T8, T11, T12}

* Test infeasible

TS_r for TUT = T2

```
Vending Machine
T02
Test Test_1 T01 T02
Test Test_2 T01 T03 T08 T12 T02
Test Test_3 T01 T04 T08 T12 T02*
Test Test_4 T01 T04 T05 T08 T12 T02*
Test Test_5 T01 T03 T05 T08 T12 T02
Test Test_6 T01 T04 T05 T05 T08 T12 T02*
Test Test_7 T01 T03 T05 T05 T08 T12 T02
```

TS_r for TUT = T5

```
Vending _System
T05
Test Test_1 T01 T03 T05
Test Test_2 T01 T04  T08 T12 T03 T05*
Test Test_3 T01 T04 T08 T12 T04 T05 T08 T12 T03 T05*
Test Test_4 T01 T04 T05*
Test Test_5 T01 T03 T08 T12 T04 T05
Test Test_6 T01 T03 T05 T08 T12 T04 T05
Test Test_7 T01 T04 T08 T12 T04 T05*
Test Test_8 T01 T04 T05 T08 T12 T04 T05*
Test Test_9 T01 T03 T08 T12 T03 T08 T12 T04 T05*
Test Test_10 T01 T03 T08 T12 T03 T05
Test Test_11 T01 T03 T05 T08 T12 T03 T05
Test Test_12 T01 T04 T05 T08 T12 T03 T05*
Test Test_13 T01 T04 T08 T12 T04 T05 T08 T12 T03 T12*
Test Test_14 T01 T03 T08 T12 T03 T05  T08 T12 T04 T05*
Test Test_15 T01 T04 T08 T12 T03 T05 T08 T12 T04 T05*
Test Test_16 T01 T03 T08 T12 T04 T05 T08 T12 T04 T05
Test Test_17 T01 T04 T08 T12 T03 T05  T08 T12 T03 T05*
Test Test_18 T01 T04 T08 T12 T04 T05 T08 T12 T04 T05*
Test Test_19 T01 T04 T08 T12 T03 T05 T08 T12 T03 T05 *
Test Test_20 T01 T03 T08 T12 T04 T05 T08 T12 T08 T05
Test Test_21 T01 T03 T05 T05
Test Test_22 T01 T04  T08 T12 T03 T05 T05*
Test Test_23 T01 T04 T08 T12 T04 T05 T05 T08 T12 T03 T05*
Test Test_24 T01 T04 T05 T05*
Test Test_25 T01 T03 T08 T12 T04 T05 T05
Test Test_26 T01 T03 T05 T05 T08 T12 T04 T05
Test Test_27 T01 T04 T08 T12 T04 T05 T05*
Test Test_28 T01 T04 T05 T05 T08 T12 T04 T05*
Test Test_29 T01 T03 T08 T12 T03 T08 T12 T04 T05 T05*
Test Test_30 T01 T03 T08 T12 T03 T05 T05
Test Test_31 T01 T03 T05 T05 T08 T12 T03 T05
Test Test_32 T01 T04 T05 T05 T08 T12 T03 T05*
```

```
Test Test_33 T01 T04 T08 T12 T04 T05 T05 T08 T12 T03 T12*
Test Test_34 T01 T03 T08 T12 T03 T05 T05 T08 T12 T04 T05*
Test Test_35 T01 T04 T08 T12 T03 T05 T05 T08 T12 T04 T05*
Test Test_36 T01 T03 T08 T12 T04 T05 T05 T08 T12 T04 T05
Test Test_37 T01 T04 T08 T12 T03 T05 T05 T08 T12 T03 T05*
Test Test_38 T01 T04 T08 T12 T04 T05 T05 T08 T12 T04 T05*
Test Test_39 T01 T04 T08 T12 T03 T05 T05 T08 T12 T03 T05 *
Test Test_40 T01 T03 T08 T12 T04 T05 T05 T08 T12 T08 T05
```

TS_r for TUT = T8

```
Vending _Machine
T8
Test Test_1 T01 T03 T08
Test Test_2 T01 T04 T08*
Test Test_3 T01 T03 T05 T08
Test Test_4 T01 T04 T05 T08*
Test Test_5 T01 T03 T05 T08 T12 T03 T05 T08*
Test Test_6 T01 T04 T08 T12 T03 T05 T08*
Test Test_7 T01 T03 T08 T12 T04 T05 T08
Test Test_8 T01 T04 T08 T12 T04 T05 T08*
Test Test_9 T01 T03 T05 T08 T12 T03 T05 T08*
Test Test_10 T01 T04 T05 T08 T12 T04 T05 T08*
Test Test_11 T01 T03 T05 T08 T12 T04 T05 T08
Test Test_12 T01 T04 T05 T08 T12 T04 T05 T08*
Test Test_13 T01 T03 T08 T12 T04 T08
Test Test_14 T01 T03 T05 T08 T12 T04 T08
Test Test_15 T01 T04 T08 T12 T04 T08*
Test Test_16 T01 T04 T08  T12 T04 T05 T08
Test Test_17 T01 T03 T05 T05 T08
Test Test_18 T01 T04 T05 T05 T08*
Test Test_19 T01 T03 T05 T05 T08 T12 T03 T05 T08*
Test Test_20 T01 T04 T08 T12 T03 T05 T05 T08*
Test Test_21 T01 T03 T08 T12 T04 T05 T05 T08
Test Test_22 T01 T04 T08 T12 T04 T05 T05 T08*
Test Test_23 T01 T03 T05 T08 T12 T03 T05 T05 T08*
Test Test_24 T01 T04 T05 T08 T12 T04 T05 T05 T08*
Test Test_25 T01 T03 T05 T08 T12 T04 T05 T05 T08
Test Test_26 T01 T04 T05 T08 T12 T04 T05 T05 T08
Test Test_27 T01 T03 T05 T05 T08 T12 T04 T08
Test Test_28 T01 T04 T08 T12 T04 T05 T05 T08
```

TS_r for TUT = T11

```
Vending _System
T11
Test Test_1 T01 T03 T08 T11
Test Test_2 T01 T03 T05 T08 T11
Test Test_3 T01 T04 T08 T11*
Test Test_4 T01 T04 T05 T08 T11
```

```
Test Test_5 T01 T03 T08 T12 T03 T05 T08 T11*
Test Test_6 T01 T04 T08 T12 T03 T05 T08 T11*
Test Test_7 T01 T03 T08 T12 T04 T05 T08 T11
Test Test_8 T01 T04 T08 T12 T04 T05 T08 T11*
Test Test_9 T01 T03 T08 T12 T04 T11
Test Test_10 T01 T04 T08 T12 T04 T11*
Test Test_11 T01 T03 T05 T08 T12 T04 T08 T11
Test Test_12 T01 T03 T05 T08 T12 T04 T05 T08 T11
Test Test_13 T01 T04 T05 T08 T12 T04 T08 T11*
Test Test_14 T01 T04 T05 T08 T12 T04  T05 T08 T11
Test Test_15 T01 T03 T05 T05 T08 T12 T04 T08 T11
Test Test_16 T01 T03 T05 T05 T08 T12 T04 T05 T08 T11
Test Test_17 T01 T04 T05 T05 T08 T12 T04 T08 T11*
Test Test_18 T01 T04 T05 T05 T08 T12 T04  T05 T08 T11
Test Test_19 T01 T04 T05 T05 T08 T11
Test Test_20 T01 T03 T08 T12 T03 T05 T05 T08 T11*
Test Test_21 T01 T04 T08 T12 T03 T05 T05 T08 T11*
Test Test_22 T01 T03 T08 T12 T04 T05 T05 T08 T11
Test Test_23 T01 T04 T08 T12 T04 T05 T05 T08 T11*
Test Test_24 T01 T03 T05 T05 T08 T11
```

TS_r for TUT = T12

```
Vending _System
T12
Test Test_1 T01 T03 T08 T12
Test Test_2 T01 T03 T05 T08 T12
Test Test_3 T01 T04 T08 T12*
Test Test_4 T01 T04 T05 T08 T04 T12*
Test Test_5 T01 T03 T08 T12 T03 T05 T08 T12*
Test Test_6 T01 T04 T08 T12 T03 T05 T08 T12*
Test Test_7 T01 T03 T08 T12 T04 T05 T08 T12
Test Test_8 T01 T04 T05 T08 T12 T04 T08 T12
Test Test_9 T01 T03 T08 T12 T04 T08 T12
Test Test_10 T01 T04 T08 T12 T04 T08 T12
Test Test_11 T01 T03 T05 T08 T12 T04 T08 T12
Test Test_12 T01 T03 T05 T08 T12 T04 T05 T08 T12
Test Test_13 T01 T04 T05 T08 T12 T04 T08 T12*
Test Test_14 T01 T04 T05 T12 T08 T04 T05 T08 T12*
Test Test_15 T01 T03 T05 T05 T08 T12 T04 T08 T12
Test Test_16 T01 T03 T05 T05 T08 T12 T04 T05 T08 T12
Test Test_17 T01 T04 T05 T05 T08 T12 T04 T08 T12*
Test Test_18 T01 T04 T05 T05 T12 T08 T04 T05 T08 T12*
Test Test_19 T01 T03 T05 T05 T08 T12
Test Test_21 T01 T04 T05 T05 T08 T04 T12*
Test Test_22 T01 T03 T08 T12 T03 T05 T05 T08 T12*
Test Test_23 T01 T04 T08 T12 T03 T05 T05 T08 T12*
Test Test_24 T01 T03 T08 T12 T04 T05 T05 T08 T12
```

APPENDIX C: Cruise Control System (CCS)

C.1 Requirements of the CCS

A Cruise Control System (CCS) maintains the speed of a car at a pre-selected value. The CCS calculates the required output to be passed on to the actuator and maintains the cruise speed by controlling the throttle. Physically the system consists of a control module, a sensor that measures wheel revolutions, the accelerator, the brake, and a control panel with all the displays and buttons to inform the user. There are six control buttons on the control panel: On, Off, Set, Resume, Accel and Coast. There are two displays on the control panel: the current speed and the cruise speed.

R = {r | r is a requirement which is a **transition**}

R = {$r1$, $r2$, $r3$, $r4$, $r5$, $r6$, $r7$, $r8$, $r9$, $r10$, $r11$} for the CCS EFSM where:

r1: When the CCS is on, the user can set the cruise speed as the current speed by using the *Set* button. The CCS is now activated. The CCS will store the current speed as the new set value for the cruise speed. The current speed must be at least 40km/h. The CCS displays the cruise speed set. The unit used in the speed display is km/h.

$r1$ = **T2**

r2: The CCS measures the current speed every 2 seconds and shows it on the control panel's speed display. The speed is measured by a sensor detecting wheel revolutions of the car.

$r2$ = **T3**

r3: If the current speed is greater than the cruise speed, the cruise control calculates the required speed value to decrease the current speed.

$r3$ = **T4**

r4: If the current speed is smaller than the cruise speed, the cruise control calculates the required speed value to increase the current speed.

$r4$ = **T5**

r5: If the current speed drops below 35km/h, the CCS is suspended automatically.

$r5$ = **T8**

r6: The user can increase the cruise speed using the *Accel* button. Every hit on the *Accel* button will increase the cruise speed by 1km/h.

$r6$ = **T6**

r7: The user can decrease the cruise speed by using the *Coast* button. Every hit on the *Coast* button will decrease the cruise speed by 1km/h.

$r7$ = **T7**

r8: When the user presses the *brake* pedal, the CCS is suspended and the car speed changes according to the manual control. Cruise control may be reactivated by using the *Resume* button.

$r8$ = **T9**

r9: When the user presses the *accelerator* pedal, the CCS is suspended and the car speed changes according to the manual control. The CCS may be reactivated by using the *Resume* button.

$r9$ = **T10**

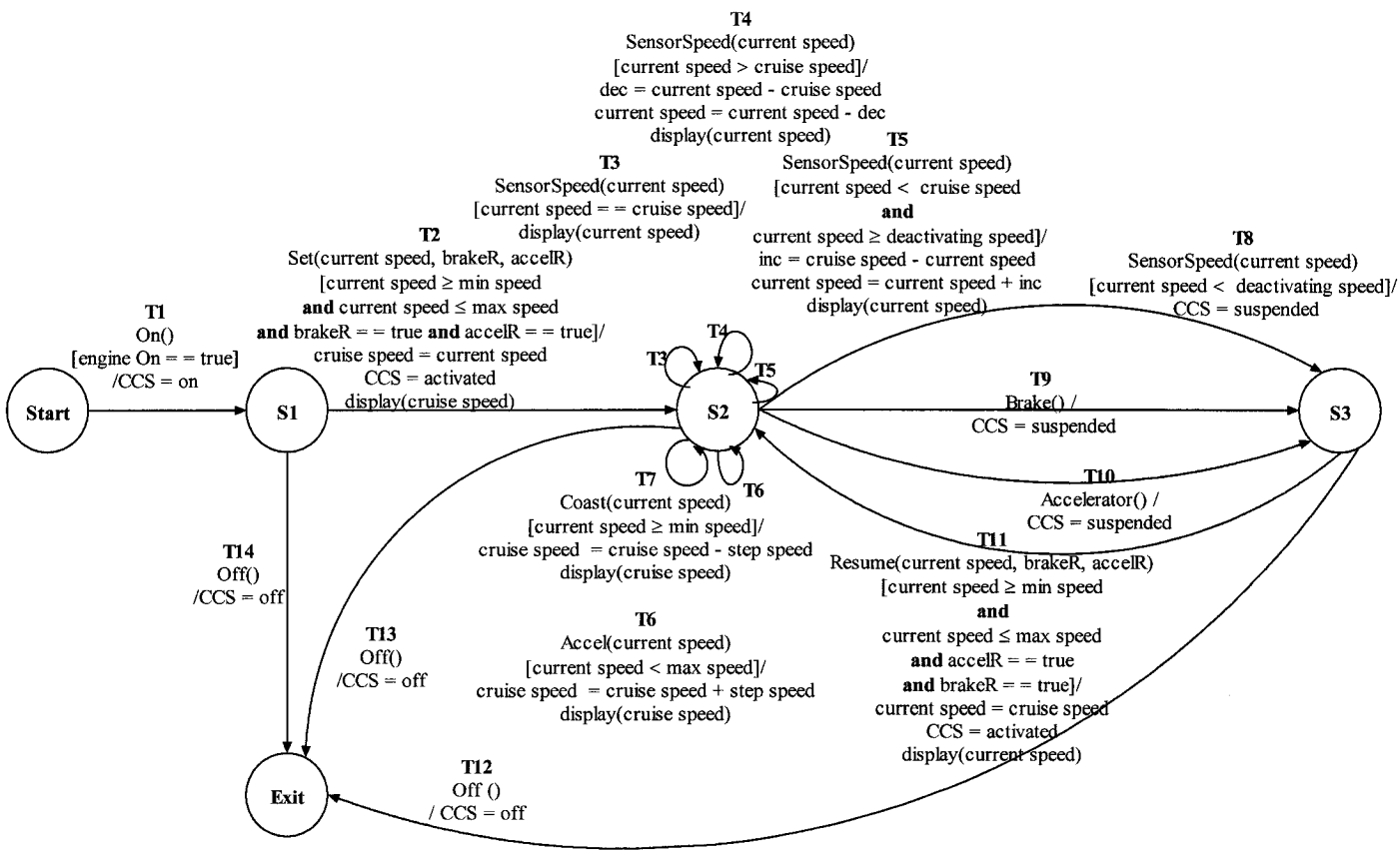
r10: The *Resume* button reactivates the CCS and the current speed is reset to the last set cruise speed.

$r10$ = **T11**

***r*11**: While CCS is activated, the user turns the CCS off by using the *Off* button.

$r11$ = **T13**

C.2 The EFSM model of the CCS



Symbol Definition

System inputs

- *On*(): the *On* button is pressed by the driver.
- *Off*(): the *Off* button is pressed by the driver.
- *Set*(*current speed*, *brakeR*, *accelR*): the *Set* button is pressed by the driver.

- *Resume*(*current speed*, *brakeR*, *accelR*): the *Resume* button is pressed by the driver.
- *SensorSpeed*(*current speed*): the CCS reads the current speed of the car.
- *Accel*(*current speed*): the *Accel* button is pressed by the driver.
- *Coast*(*current speed*): the *Coast* button is pressed by the driver.
- *Brake*(): the brake pedal is pressed by the driver.
- *Accelerator*(): the accelerator pedal is pressed by the driver.

System outputs

- *current speed*: current driving speed of the car.
- *cruise speed*: speed stored in the controller to be maintained.

System variables

- *CCS*: status of the CCS. $CCS \in \{on, off, activated, suspended\}$
- *dec*: amount of speed to be decreased by the accelerator to reach the cruise speed.
- *inc*: amount of speed to be increased by the accelerator to reach the cruise speed.
- *brakeR*: Boolean variable indicating that the brake is released if set to true.
- *accelR*: Boolean variable indicating that the accelerator is released if set to true.
- *engine On*: Boolean variable indicating that the engine is On if set to true.
- *current speed*
- *cruise speed*

Constants

- *max speed*: maximum cruise speed. *max speed* is 180 km/h.
- *min speed*: minimum cruise speed. *min speed* is 40 km/h.

- *deactivating speed*: speed that suspends the CCS automatically. *deactivating speed* is 35 km/h.
- *step speed*: *step speed* is 1 km/h.

States

Start: the CCS is Off.

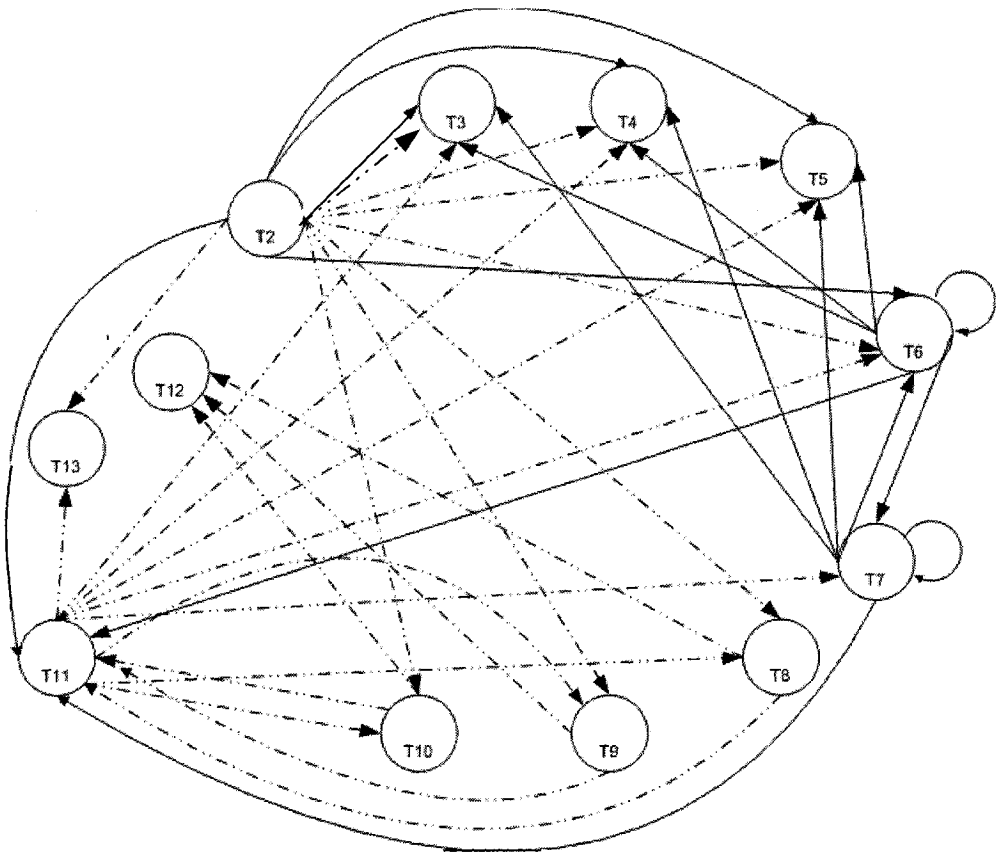
S1: Ready. The CCS is On and not yet activated. It is waiting for the driver to activate it by setting a cruise speed using the *Set* button.

S2: Set. A cruise speed is set and the CCS is maintaining it. The CCS is now activated.

S3: Wait. The CCS is suspended and there is a cruise speed set. The CCS is waiting until the driver presses the *Resume* button to reactivate the CCS.

Exit: The CCS is Off.

C.3 SDG of the CCS EFSM



SDG of Cruise Control EFSM

——► data dependence edge
– · – · ► control dependence edge

Data dependence edges in the SDG of the CCS

	Transition	has a data dependence on	Identified by variables
1	T2	T3	cruise_speed
		T4	cruise_speed
2		T5	cruise_speed
3		T6	cruise_speed
4		T11	cruise_speed
5	T6	T3	cruise_speed
6		T4	cruise_speed
7		T5	cruise_speed
8		T6	cruise_speed
9		T7	cruise_speed
10		T11	cruise_speed
11	T7	T3	cruise_speed
12		T4	cruise_speed
13		T5	cruise_speed
14		T6	cruise_speed
15		T7	cruise_speed
16		T11	cruise_speed

Control dependence edges in the SDG of CCS

	Transition	has a control dependence on
1	T2	T3
2	T2	T4
3	T2	T5
4	T2	T6
5	T2	T7
6	T2	T8
7	T2	T9
8	T2	T10
9	T2	T13
10	T8	T11
11	T8	T12
12	T9	T11
13	T9	T12
14	T10	T11
15	T10	T12
16	T11	T3
17	T11	T4
18	T11	T5
19	T11	T6
20	T11	T7

21	T11	T8
22	T11	T9
23	T11	T10
24	T11	T13

C.4 df-chains of the CCS EFSM

The set of df-chains generated for each *r* in *R* = {T03, T04, T05, T08, T11}

TUT	*df* #	*df*	# def-clear paths for du-pairs of *df*	$\|AP_{df}\|$
T3	1	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	32
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T03,2)}	32	
	2	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	2048
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T03,2)}	32	
	3	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	131072
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T03,2)}	32	
	4	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	2048
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T03,2)}	32	
	5	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	131072
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T03,2)}	32	
T4	1	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	32
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T04,4)}	32	
	2	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	2048
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T04,4)}	32	
	3	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	131072
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T04,4)}	32	
	4	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	2048
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T04,4)}	32	
	5	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	131072
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T03,2)}	32	

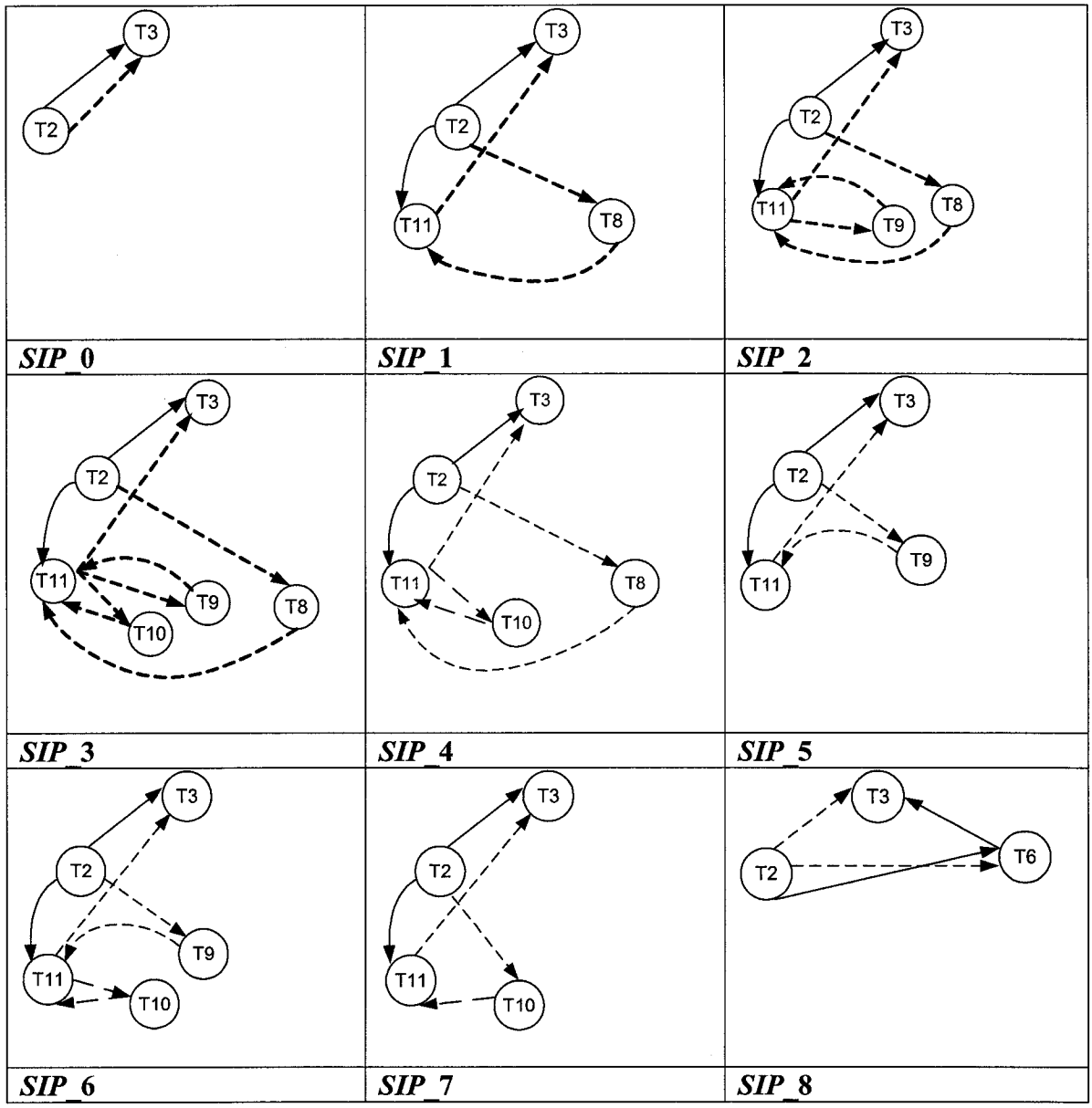
T5	**1**	{d(*current_speed*,T02,3),u(1,*current_speed*,T02,7)	1	32
		d(*cruise_speed*,T02,8), u(1,*cruise_speed*,T05,5)}	32	
	2	{d(*current_speed*,T02,3),u(1,*current_speed*,T02,7),	1	4069
		d(*cruise_speed*,T02,8),u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4),u(*cruise_speed*,T05,5)}	64	
	3	{d(*current_speed*,T02,3),u(*current_speed*,T02,7)	1	262144
		d(*cruise_speed*,T02,8),u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4),u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4),u(*cruise_speed*,T05,5)}	64	
	6	{d(*current_speed*,T02,3),u(*current_speed*,T02,7),	1	4069
		d(*cruise_speed*,T02,8),u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4),u(*cruise_speed*,T05,5)}	64	
	7	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	262144
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T05,5)}	64	
T6	**1**	{d(*current_speed*,T02,3),u(*current_speed*,T02,7),	1	64
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3)}	64	
	2	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	4069
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T06,3)}	64	
	3	{d(*current_speed*,T02,3),u(*current_speed*,T02,7),	1	262144
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4),u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4),u(*cruise_speed*,T06,3)}	64	
T7	**1**	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	64
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3)}	64	
	2	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	4069
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T07,3)}	64	
	3	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	262144
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T07,3)}	64	
T11	**1**	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	24
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T11,7)}	24	
	2	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	1536
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T11,7)}	24	
	3	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	98304
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T11,7)}	24	
	4	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	1536
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T11,7)}	24	
	5	{d(*current_speed*,T02,3), u(*current_speed*,T02,7),	1	98304
		d(*cruise_speed*,T02,8), u(*cruise_speed*,T06,3),	64	
		d(*cruise_speed*,T06,4), u(*cruise_speed*,T07,3),	64	
		d(*cruise_speed*,T07,4), u(*cruise_speed*,T11,7)}	24	

C.5 S_r and TS_r Generation for the Cruise Control System

S_r is generated by the *TSG* program for each r in R = {T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T13}. The *TSG* program output is summarized in the table below.

TUT	$\|S_{r\_elem}\|$	$\|S_r\|$	$\|TS_r\|$
T2	0	0	0
T3	429	1465	1465
T4	429	1465	1465
T5	428	1468	1468
T6	106	408	408
T7	114	440	440
T8	0	0	0
T9	0	0	0
T10	0	0	0
T11	129	435	435
T13	0	0	0

***SIP*_0 to *SIP*_8 in S_r for r = T3 in the CCS EFSM (for r = T3, $|S_r|$=1465)**



*SIP*_0	*SIP*_1	*SIP*_2
*SIP*_3	*SIP*_4	*SIP*_5
*SIP*_6	*SIP*_7	*SIP*_8

APPENDIX D: Algorithms

D.1 Algorithms to generate additional EFSM data structures

First, V is formed by using V_Alg.

Then, $Def(v)$ and $Use(v)$ are formed by using the Def_Use_Alg,

$S_{in}(Z)$ and $T_{in}(Z)$ are formed by using the inAdjAlg algorithm for every state Z in M except for the *Start* state,

$S_{out}(Z)$, $T_{out}(Z)$ are formed by using the outAdjAlg algorithm for every state Z in M except for *Exit* state.

Algorithm V_Alg

Generates V, the set of variables in M.

Input: EFSM M

Output: V, the set of variables in M.

$V \leftarrow \{\}$

For each transition t in M do

for each $(v, Label, OType, OOrder) \in Volist(t)$ do

if $v \notin V$

then $V \leftarrow V \cup \{v\}$

Algorithm Def_Use_Alg

Generates $Def(v)$ and $Use(v)$ for all variables v in V.

Input: EFSM M, V

Output: $Def(v)$, $Use(v)$

For each variable v in V do

$Def(v) \leftarrow \{\}$

$Use(v) \leftarrow \{\}$

for each transition t in M do

for each $(v, Label, OType, OOrder) \in Volist(t)$

if $OType = def$

then $Def(v) \leftarrow Def(v) \cup \{t\}$

else $Use(v) \leftarrow Use(v) \cup \{t\}$

Algorithm inAdjAlg

Generates the list of incoming transitions for each state Z and their starting states.

Input: EFSM M

Output: $T_{in}(Z)$ and $S_{in}(Z)$ for each state $Z \neq Start$

for each state Z, except *Start*, in M do

$S_{in}(Z) \leftarrow \{\}$

$T_{in}(Z) \leftarrow \{\}$

for each incoming transition T_i to state Z do

$T_{in}(Z) \leftarrow T_{in}(Z) \cup \{T_i\}$

for each T_i in $T_{in}(Z)$

$S_{in}(Z) \leftarrow S_{in}(Z) \cup \{s_b(T_i)\}$

Algorithm outAdjAlg

Generates the list of outgoing transitions for each state Z and their terminating states.

Input: EFSM M

Output: $T_{out}(Z)$ and $S_{out}(Z)$ for each state $Z \neq Exit$

for each state Z, except *Exit*, in M do

$T_{out}(Z) \leftarrow \{\}$

$S_{out}(Z) \leftarrow \{\}$

for each outgoing transition T_i from state Z do

$T_{out}(Z) \leftarrow T_{out}(Z) \cup \{T_i\}$

for each T_i in $T_{out}(Z)$ do

$S_{out}(Z) \leftarrow S_{out}(Z) \cup \{s_e(T_i)\}$

D.2 Detailed version of the SDG algorithm

Algorithm SDG Generation

Generates the Static Dependence Graph for an EFSM.

Input: M, the data structure for the EFSM,

Output: the adjacency matrix of the SDG for M.

Let M denote (*name*, *Start*, *Exit*, {*transition*}) where

name is the name of the EFSM,

Start is the start state of M,

Exit is the exit state of M and

transition is (*Label*, s_b, s_e, *Volist*, *listOfActions*)

Let t, T_i, T_j, T_k denote a transition in the EFSM or a node in the SDG.

For a transition t,

Label(t) denotes its label,

$s_b(t)$ denotes its beginning state by the index of the state,

$s_e(t)$ denotes its ending state by the index of the state,

Volist(t) denotes {(v, *Label*, *OType*, *OOrder*) | variable v occurs in transition with *Label* t as *OType* in order *OOrder*} where

OType denotes the type of occurrence which is either a definition denoted *def*, or a *c*-use denoted *c-use*, or a *p*-use denoted *p-use*,

OOrder denotes the order of occurrence,

Elements in *Volist*(t) set are referenced by an index between 0 and size of *Volist*(t) – 1, corresponding to their order in the set.

ListOfActions(t) denotes {(*Index*, *Id*, *AType*, *List*)} where for an action a

Index(a) denotes its index,

Id denotes its id,

AType(a) denotes the action type, which is either an input denoted *INPUT*, an output denoted *OUTPUT*, an assignment statement denoted *ASSIGN*, a set statements denoted *SET*, a reset statement denoted *RESET*, a predicate denoted *PRED* or a procedure call denoted *PROC*,

List(a) denotes {(v, *Label*, *OType*, *OOrder*) | variable v occurs in transition with *Label* t in action a as *OType* in order *OOrder*}.

Let V denote {v | v is a variable in (EFSM)M},

Def(v) denote {t | t is a transition in M which defines v},

Use(v) denote {t | t is a transition in M which uses v},

SDG[T_i, T_k] denote {(v, *dependenceType*, v_{ik}) | T_i and T_k are nodes in *SDG*, v is a variable in $V \cap T_i$ has an outgoing edge e to node T_k and the type of dependence on e is equal to *dependenceType* wrt variable v that occurs in T_i as in *Volist*(T_i) and in T_k as in *Volist*(T_k)},

dependenceType denote the type of dependence which is either a control dependence denoted *cd* or a data dependence denoted *dd* or a du-pair based on a *c*-use denoted *dcu-pair* or a du-pair based on a *p*-use denoted *dpu-pair*,

v_{ik} denote (*oo_def*, *oo_use*) | variable v in V occurs in transition T_i in M as a definition denoted *def*, in the order of occurrence *oo_def*, and in T_k in M as a use denoted *use*, in the order of occurrence *oo_use*,

$T_{in}(Z)$ denote $\{t \mid t$ is an incoming transition to state Z in $M\}$, for $Z \neq Start$,

$T_{out}(Z)$ denote $\{t \mid t$ is an outgoing transition from state Z in $M\}$, for $Z \neq Exit$,

$S_{in}(Z)$ denote $\{s \mid s = s_b(t)$ and $t \in T_{in}(Z)\}$, for $Z \neq Start$,

$S_{out}(Z)$ denote $\{s \mid s = s_e(t)$ and $t \in T_{out}(Z)\}$, for $Z \neq Exit$,

$S_{in}^*(Z)$ denote $\{s \mid s$ is a state and state s reaches state Z in $M\}$,

$S_{out}^*(Z)$ denote $\{s \mid s$ is a state and state Z reaches state s in $M\}$,

$PD(Z)$ denote $\{Y \mid Y$ is a state in $M \cap$ state Z post-dominates state $Y\}$,

$nPD(Z)$ denote $\{Y \mid Y$ is a state in $M \cap$ state Z is reachable from state Y and state Z does not post-dominate state $Y\}$,

$DD(T_k)$ denote $\{(T_i, v) \mid T_i$ and T_k are nodes in the *SDG* and T_k has an incoming data dependence edge, wrt variable v, from $T_i\}$,

$CD(T_k)$ denote $\{T_i \mid T_i$ and T_k are nodes in the *SDG* and T_k has an incoming control dependence edge from $T_i\}$.

Input to be formed from the EFSM data structure

First, V, $Def(v)$ and $Use(v)$ for every v in M are formed by using V_Alg and Def_Use_Alg. Then $S_{in}(Z)$ and $T_{in}(Z)$ are formed by using the inAdjAlg algorithm for every state Z in M except for *Start*. $S_{out}(Z)$, $T_{out}(Z)$ are formed by using the outAdjAlg algorithm for every state Z in M except for *Exit*.

SDG algorithm steps

Initialization

Execute V_Alg(M), and Def_Use_Alg(M, V), inAdjAlg(M) and outAdjAlg(M) (given in Appendix D.1).

$SDG[T_i, T_k] \leftarrow \{\}$ for all transitions T_i and T_k in M.

$DD(T_k) \leftarrow \{\}$ and $CD(T_k) \leftarrow \{\}$ for all transitions T_k in M.

Unmark all states Y in M.

1. Data dependence identification

For each variable v in V do

for each transition T_i in $Def(v)$ do

for each (v, *def*, *oo_def*) in $Volist(T_i)$ do

for each transition T_k in $Use(v)$ do

for each (v, *use*, *oo_use*) in $Volist(T_k)$ do

//use is a c-use or a p-use //

(*dependenceType*, v_{ik}) ← *dataDependence* (T_i, T_k, v, *oo_def*, *oo_use*, *use*)

//Call the dataDependence function that returns //

//the type of dependence between T_i and T_k wrt v //

// that is defined in T_i in oo_def and used in T_k in oo_use //

if (*dependenceType*, v_{ik}) ≠ *Nil*
//∃ a def-clear path from T_i to T_k wrt v//

then

$SDG[T_i, T_k] \leftarrow SDG[T_i, T_k] \cup \{(v, dependenceType, v_{ik})$

if *dependenceType* = *dd*

then

if $(T_i, v) \notin DD(T_k)$

then

$DD(T_k) \leftarrow DD(T_k) \cup \{(T_i, v)\}$

// T_i and T_k are nodes in the SDG//

2. Control dependence identification

// Find, for each state in the EFSM, the set of states that it post-dominates//

a. For each state *Z*, except *Exit*, in *M* do

//Find all states that Z post-dominates//

$PD(Z) \leftarrow \{Z\}$

$nPD(Z) \leftarrow \{\}$

//Call the inTransitiveClosure procedure to form the set S_{in}(Z), of states//*

//that reach state Z in M //

$S_{in}^*(Z) \leftarrow \{\}$

inTransitiveClosure(Z, S_{in}*(Z), $S_{in}(Y)$ ∀ state Y in M)

for each state Y in S_{in}*(Z) such that $Y \neq Z$ do

if FindPath(Y, Z) = *false* *//Boolean function that returns true //*
// if there is a path from Y to Exit that//
// doesn't go through Z //

then

$PD(Z) \leftarrow PD(Z) \cup \{Y\}$

for each state X in $S_{out}(Y)$ such that $X \neq Z$ and $X \neq Exit$ do

$PD(Z) \leftarrow PD(Z) \cup \{X\}$

else $nPD(Z) \leftarrow nPD(Z) \cup \{Y\}$

b. For each state Z, except *Exit*, in M do

//Find all transitions that Z post-dominates//

for each state Y in $nPD(Z)$ do

for each transition T_i in $T_{out}(Y)$ do

if $s_e(T_i)$ is in $PD(Z)$

then *//Z post-dominates T_i//*

for each transition T_k in $T_{out}(Z)$ do

// there is control dependence between T_i and//
//each outgoing transition T_k from Z//

$SDG[T_i, T_k] \leftarrow SDG[T_i, T_k] \cup \{(Nil, cd, Nil)\}$

$CD(T_k) \leftarrow CD(T_k) \cup \{T_i\}$

3. Procedures and functions

dataDependence(T_i, T_k, *v, oo_def , oo_use, use)*

a. If ($T_i = T_k$ and *oo_def* > *oo_use*) or ($T_i \neq T_k$)

then

//check whether (v, Label, def, oo_def) is the last def of v in//

//Volist(T_i) //

index ← index of (*v*, *Label*, *def*, *oo_def*) in *Volist*(T_i) + 1

while *index* ≠ 0 and *index* ≠ size of *Volist*(T_i)

(*var*, *Label*, *OType*, *OOrder*) ← element in *Volist*(T_i) whose index is *index*

if (*var*, *Label*, *OType*, *OOrder*) = (*v*, *Label*, *def*, *OOrder*)

then *index* ← 0

else *index* ← *index* + 1

//end of while loop//

if *index* = 0

then return *Nil*

else *//check whether (v, Label, def, oo) is in Volist(T_k)//*

// such that oo < oo_use //

index ← index of (*v*, *Label*, *use*, *oo_use*) in *Volist*(T_k) - 1

found ← *false* *// found is a Boolean variable//*

while *index* ≠ -1 and *found* = false

(*var*, *Label*, *OType*, *OOrder*) ← element in *Volist*(T_k) whose index is *index*

if (*var*, *Label*, *OType*, *OOrder*) = (*v*, *Label*, *def*, *OOrder*)

then *found* ← *true*

else *index* ← *index* – 1

//end of while loop//

if *found* = *true*

then return *Nil*

else

//Call the outTransitiveClosure procedure to//

// form the set S_{out}($s_e(T_i)$) of states that reach //*

// state ($s_e(T_i)$) in M//

S_{out}*($s_e(T_i)$) ← {}

outTransitiveClosure($s_e(T_i)$, S_{out}*($s_e(T_i)$),$S_{out}(Y)$ ∀ state Y in M)

if $s_b(T_k) \in S_{out}$*($s_e(T_i)$)

//if T_i reaches T_k//

then

if *DefClearPath*($s_e(T_i)$, $s_b(T_j)$, v) = *true*

then

if *use* = *c-use*

then

v_{ik} ← (*oo_def* , *oo_use*)

return (*dd*, v_{ik})

else *// use = p-use//*

v_{ik} ← (*oo_def* , *oo_use*)

return (*dd*, v_{ik})

else return *Nil*

else return *Nil*

b. else if $T_i = T_k$ and *oo_def* < *oo_use*

then *//check whether there is (v, Label, def, oo) in Volist(T_i) //*

//such that oo_def < oo < oo_use //

index ← index of (*v*, *Label*, *def*, *oo_def*) in *Volist*(T_i) +1

while *index* ≠ 0 and *index* ≠ index of (*v*, *Label*, *use*, *oo_use*) in

Volist(T_i)

(*var*, *Label*, *OType*, *OOrder*) ← element in *Volist*(T_i)
whose index is *index*

if (*var*, *Label*, *OType*, *OOrder*) = (*v*, *Label*, *def*, *OOrder*)

then *index* ← 0

else *index* ← *index* + 1

//end of while loop//

if *index* = 0

then return *Nil*

else *//oo_use is reached without encountering a def of v//*

if *use* = *c-use*

then

v_{ii} ← (*oo_def* , *oo_use*)

return (*dcu-pair*, v_{ii})

else *// use = p-use //*

v_{ii} ← (*oo_def* , *oo_use*)

return (*dpu-pair*, v_{ii})

else return *Nil*

inTransitiveClosure(*Y*, S_{in}*(*Y*), S_{in}(*Z*) ∀ state *Z* in *M*)

//Generates S_{in}(Y) for state Y in M//*

S_{adj} ← S_{in}(*Y*)

for each state *Z* in S_{adj} do

S_{in}*(*Y*) ← S_{in}*(*Y*) ∪ {*Z*}

$S_{temp} \leftarrow S_{in}(Z)$

for each state S in S_{temp} do

$S_{adj} \leftarrow S_{adj} \cup \{S\}$

endfor

endfor

outTransitiveClosure(Y, S_{out}*(Y), S_{out}(Z) $\forall$ state Z in M)

//Generates S_{out}(Y) for state Y in M//*

$S_{adj} \leftarrow S_{out}(Y)$

for each state Z in S_{adj} do

S_{out}*(Y) $\leftarrow$ S_{out}*(Y) $\cup$ $\{Z\}$

$S_{temp} \leftarrow S_{out}(Z)$

for each state S in S_{temp} do

$S_{adj} \leftarrow S_{adj} \cup \{S\}$

endfor

endfor

DefClearPath(*start*, *end*, *v*) *//Boolean recursive function that returns true if there //*
// is a def-clear path wrt v between states start and end in M//

S_{out}* (*start*) $\leftarrow$ {}

if (*start* = *end*)

then return *true*

else *//call the outTransitiveClosure procedure to find the states that start reaches//*

outTransitiveClosure (*start*, S_{out}*(*start*), $S_{out}(Y)$ $\forall$ state Y in M)

for each state Z in S_{out}*(*start*) do

for each transition T in $T_{in}(Z)$ do

if $T \notin Def(v)$ *//if v is not defined in T//*

then *//Recursive call for the DefClearPath procedure //*

DefClearPath(Z, *end*, v)

return *false*

FindPath(*Y*, *Z*) *//Boolean function that returns true if there is a path from Y to the exit// //state that does not go through Z //*

$S_{out\_minus\_Z}(Y) \leftarrow \{\}$

$S_{out\_temp}(Y) \leftarrow S_{out}(Y)$

for each state S in $S_{out\_temp}(Y)$ do

if $S \neq Z$

then

$S_{out\_minus\_Z}(Y) \leftarrow S_{out\_minus\_Z}(Y) \cup \{S\}$

for each state Q in $S_{out}(S)$ do

if ($Q \neq Z$)

then

$S_{out\_temp}(Y) \leftarrow S_{out\_temp}(Y) \cup \{Q\}$

if $Exit \in S_{out\_minus\_Z}(Y)$

then return *true* *//there is a path from Y to the exit state that does not go through Z//*

else return *false*