AUTOMATA THEORETIC MODEL CHECKING OF WEB SERVICE SYSTEMS

A PROJECT REPORT

Submitted By

ABISHEK SUDARSHAN K. 312211104002

BHARATH RAJ T. 312211104023

KARTHIK PERUMAL P. 312211104053

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

SSN COLLEGE OF ENGINEERING

KALAVAKKAM 603110

ANNA UNIVERSITY :: CHENNAI - 600025

April 2015

ANNA UNIVERSITY : CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this project report titled **"AUTOMATA THEORETIC MODEL CHECKING OF WEB SERVICE SYSTEMS"** is the *bonafide* work of "**Abishek Sudarshan K (312211104002)**, **Bharath Raj T (312211104023)**, and **Karthik Perumal P (312211104053)**" who carried out the project work under my supervision.

Dr. Chitra Babu	**Dr. S. Sheerazuddin**
Head of the Department	**Supervisor**
Professor,	Associate Professor,
Department of CSE,	Department of CSE,
SSN College of Engineering,	SSN College of Engineering,
Kalavakkam - 603 110	Kalavakkam - 603 110

Place:
Date:

Submitted for the examination held on............

Internal Examiner **External Examiner**

ACKNOWLEDGEMENTS

We thank GOD, the almighty for giving us strength and knowledge to do this project.

We would like to thank and express our deep sense of gratitude to our guide **Dr. SHEERAZUDDIN**, Associate Professor, Department of Computer Science and Engineering, for his valuable advice and suggestions as well as his continued guidance, patience and support that helped us to shape and refine our work.

Our sincere thanks to **Dr. CHITRA BABU**, Professor and Head of the Department of Computer Science and Engineering, for her words of advice and encouragement and we would like to thank our project Coordinator **Dr. SHEERAZUDDIN**, Professor, Department of Computer Science and Engineering for his valuable suggestions throughout this project.

We express our deep respect to the founder **Dr. SHIV NADAR**, Chairman, SSN Institutions. We also express our appreciation to our **Dr. S. SALIVAHANAN**, Principal, for all the help he has rendered during this course of study.

We would like to extend our sincere thanks to all the teaching and non-teaching staffs of our department who have contributed directly and indirectly during the course of our project work.

Finally, we would like to thank our parents and friends for their patience, cooperation and moral support throughout our life.

Abishek Sudarshan K. **Bharath Raj T.** **Karthik Perumal P.**

ABSTRACT

Every web service system consists of a number of properties that define it and a number of agents which make use of them. Based on the values of these properties and the actions of these agents, the web service system appears to be at different states at different points of time. The agents cause a transition from one of these states to another. Thus it is possible to model such a web service system as an automaton a collection of interlinked states and transitions. Also, every agent needs to obtain a certain requirement from a web service. These requirements can be expressed as temporal logical formulae. Only if the temporal formulae hold true, the agent can obtain the requirements from the web service. This means that the web service system has to be checked whether it meets the specification the logical formula. The temporal formula can be converted to an automaton using on-the-fly construction methods and the model checking of the system can be performed using automata theoretic methods. Thus, this project aims to check whether the web service system conforms to the specification (defined in the form of temporal logic formulae) by modeling them as automata.

TABLE OF CONTENTS

ABSTRACT **iii**

LIST OF TABLES **vii**

LIST OF FIGURES **viii**

1 Introduction **1**

1.1 Web Services 1

1.2 Automata Theory 2

1.3 Formal Methods 2

1.4 Temporal Logic 3

2 Existing Systems and Tools **4**

2.1 PROMELA 4

2.2 SPIN Model Checker 5

2.3 Tableau Based Construction 6

2.4 Petri Nets 6

3 System Architecture **7**

3.1 Part A - Service Automaton 7

3.1.1 Module 1 - Data and Properties 7

3.1.2 Module 2 - Service System 8

3.1.3 Module 3 - Service Automaton Construction 8

3.2 Part B - Formula Automaton 8

3.2.1 Module 4 - LTL Operators and Formulae 9

3.2.2 Module 5 - Formula Automaton Construction 9

3.3 Part C - Product Automaton 9

3.3.1 Module 6 - Model Checking 10

4 Service Automaton 11

4.1 Data and Properties 11

4.1.1 Data 11

4.1.2 Properties 12

4.1.3 Example 12

4.2 Service Systems 12

4.2.1 Example 13

4.3 Service Automaton Construction 13

4.3.1 States 14

4.3.2 Transitions 14

4.3.3 Conditions for Transitions 15

4.3.4 Steps for Construction 15

4.3.5 Example 16

5 Formula Automaton 18

5.1 LTL Operators and Formulae 18

5.1.1 Linear Temporal Logic 18

5.1.2 LTL Operators 19

5.1.3 LTL Expressions 20

5.1.4 Tree Representation of LTL Formulae 20

5.1.5 Steps for Construction 20

5.1.6 Example 21
5.2 Formula Automaton Construction 23
5.2.1 Original Construction Method 23
5.2.2 On-the-Fly Construction Method 24
5.2.3 The Algorithm 25
5.2.4 Rules for On-the-Fly Construction 26
5.2.5 Kripke Structure 27
5.2.6 Kripke Structure to Automaton 28
5.2.7 Example 28

6 Product Automaton 35
6.1 Model Checking 35
6.1.1 Buchi Automaton 35
6.1.2 Finding Product Automaton 36
6.1.3 Checking for Accepting Cycles 37
6.1.4 Depth First Search 37
6.1.5 Steps for Model Checking 38
6.2 Case One 39
6.3 Case Two 42

7 Conclusion and Future Work 45

LIST OF TABLES

5.1 LTL Operators 19

LIST OF FIGURES

1.1 An Example Automaton 2

3.1 System Architecture 10

4.1 Service Automaton 17

5.1 Tree - Step 1 21
5.2 Tree - Step 3 21
5.3 Tree - Step 5 22
5.4 Tree - Step 7 22
5.5 Graph Node 28
5.6 Graph - Step 1 29
5.7 Graph - Step 2 29
5.8 Graph - Step 3 29
5.9 Graph - Step 4 30
5.10 Graph - Step 5 30
5.11 Graph - Step 6 30
5.12 Graph - Step 7 31
5.13 Graph - Step 8 31
5.14 Graph - Step 9 31
5.15 Graph - Step 10 32
5.16 Graph - Step 11 32
5.17 Graph - Step 12 32
5.18 Graph - Step 13 33
5.19 Graph - Step 14 33
5.20 Nodes after Construction 33
5.21 Generated Kripke Structure 34
5.22 Formula Automaton 34

6.1 System Automaton - Case 1 40
6.2 Formula Automaton - Case 1 40
6.3 Product Automaton - Case 1 Part 1 41
6.4 Product Automaton - Case 1 Part 2 41
6.5 System Automaton - Case 2 43
6.6 Formula Automaton - Case 2 43

6.7 Product Automaton - Case 2 44

CHAPTER 1

Introduction

This project aims to check whether a web service system meets a particular specification. This means that web services have to be defined in a form where they represent a system. Similarly, the specification required needs to be represented as a temporal formula. Since automata theoretic model checking is used, both the system and the specification need to be modelled as automata. In the case of the web service system, we can directly define the different states the system can be in and thus define transitions needed to reach the specified states. In the case of the formula, we have to use On-the-Fly construction methods to construct the automata in lesser time and using minimal amount of space. Model checking methods using automata involves finding the product of the system automaton with the negation of the formula automaton. This results in a product automaton which is checked for accepting cycles to find out whether the web service system meets the specification.

1.1 Web Services

A Web service is a method of communication between two electronic devices over a network. It is a software function provided at a network address over the Web with the service always on as in the concept of utility computing. The W3C defines a Web service generally as:- a software system designed to support interoperable machine-to-machine interaction over a network. For the purpose of this project, we will be modelling the web services as automata. Examples of web services are Ticket Reservation system, Task Planner system, Online Banking system etc.

1.2 Automata Theory

Automata theory is the study of abstract machines and automata, as well as the computational problems that can be solved using them and also the study of self-operating virtual machines to help in logical understanding of input and output process, without or with intermediate stage(s) of computation (or any function / process). An automaton, is essentially a set of states and transitions. Any process or algorithm in computer science is also a set of states and transitions. Hence, every entity in computer science can be modelled as automata. In this project we shall be representing the web service system and the specification formula as automata.

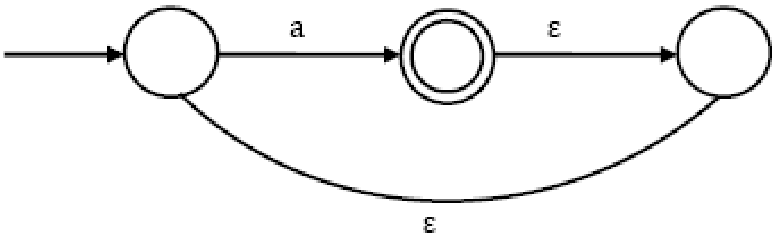


FIGURE 1.1: An Example Automaton

1.3 Formal Methods

Formal methods are techniques used to model complex systems as mathematical entities. By building a mathematically rigorous model of a complex system, it is possible to verify the system's properties in a more thorough fashion than empirical testing. In this project, we will be using the formal method known as automata theoretic model checking. In computer science, model checking or property checking refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification. Automata theoretic methods represent the model of a system as an automaton.

1.4 Temporal Logic

In logic, temporal logic is any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time.Temporal logic has found an important application in formal verification, where it is used to state requirements of hardware or software systems. For instance, one may wish to say that whenever a request is made, access to a resource is eventually granted, but it is never granted to two requestors simultaneously. Such a statement can conveniently be expressed in a temporal logic. In this project we shall be using linear temporal logic (LTL).

CHAPTER 2

Existing Systems and Tools

There are a few systems which perform model checking on web service systems. These systems use the Promela language to define the services and perform model checking directly using the Linear Temporal Logic (LTL) formula. The SPIN model checker is used for this purpose. These systems do not use automata theoretic methods. But, there exist systems which use automata for model checking. However, these systems construct the automata through alternating logic rather than use the On-the-Fly Construction. There exist systems which model specific web services but do not support representation of different web services using a single system. The proposed system makes use of automata theoretic methods to perform model checking. The LTL formulae are converted to automata through the simple On-the-Fly construction method. Moreover, the system can be used to represent different types of web services with varying properties and is not restricted to a specific type of web service.

2.1 PROMELA

PROMELA is a process modeling language whose intended use is to verify the logic of parallel systems. Given a program in PROMELA, Spin can verify the model for correctness by performing random or iterative simulations of the modeled system's execution, or it can generate a C program that performs a fast exhaustive verification of the system state space. During simulations and verifications SPIN checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to prove the correctness of system invariants and it can find non-progress execution cycles. Finally, it supports the verification of linear time temporal constraints; either with Promela never-claims or by directly formulating the constraints in temporal logic. Each model can be verified with Spin under different types of assumptions about the environment. Once the correctness of a model has been established with Spin, that fact can be used in the construction and verification of all subsequent models.

PROMELA programs consist of processes, message channels, and variables. Processes are global objects that represent the concurrent entities of the distributed system. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run.

2.2 SPIN Model Checker

SPIN is a general tool for verifying the correctness of distributed software models in a rigorous and mostly automated fashion. Systems to be verified are described in Promela (Process Meta Language), which supports modeling of asynchronous distributed algorithms as non-deterministic automata (SPIN stands for ”Simple Promela Interpreter”). Properties to be verified are expressed as Linear Temporal Logic (LTL) formulas, which are negated and then converted into Bchi automata as part of the model-checking algorithm. In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user.

2.3 Tableau Based Construction

In order to construct the formula automaton from the linear temporal logic formula, tableau based methods have been used in existing systems. This method involves processing each formula and generating the table of formulae which can hold in the next state thus creating a a new set of formulae. These formulae are once again processed. When a set of formulae can no longer be processes, the algorithm backtracks to a previous set of formulae. This method is very rigorous and does not produce efficient results when compared to On-the-Fly techniques.

2.4 Petri Nets

A Petri net (also known as a place/transition net or P/T net) is one of several mathematical modeling languages for the description of distributed systems. A Petri net is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, signified by bars) and places (i.e. conditions, signified by circles). The directed arcs describe which places are pre- and/or postconditions for which transitions (signified by arrows). Some existing systems use Petri Nets, rather than automata, to model the web systems.

CHAPTER 3

System Architecture

The system has been divided into six major modules. Each of these modules has a specific function and can be used as a separate unit by themselves. Some of the modules are also expanded further into sub-modules.

3.1 Part A - Service Automaton

This part pertains to the construction of the service automaton from the web service system and its properties.

3.1.1 Module 1 - Data and Properties

This module contains the generic data types such as date, which are required for all web services in general. The inputs to the service are specified in this module, each input is characterised by its type and name.They are modelled as classes, with the required functions to use and access them, eg. methods to initialise date and compare two dates.This module can be customised for a specific web service by adding some more classes.

3.1.2 Module 2 - Service System

This module describes the web service in detail, it makes use of the definitions in the data module. Each web service is characterised by its name and properties. The number of properties is also specified in this module. Methods to initialise the web service and methods to get information about the web service such as no. of properties are specified in this module.

3.1.3 Module 3 - Service Automaton Construction

This module converts the web service into an automata. It makes use of the data and services module. The states and transitions which are part of the automata are specifies here. The state is modelled as a class and each state has a name, state no., a flag which specifies whether it is start or final state. The transitions are modelled as a class where each transition has the service name, input and output states for the transition, the deciding property and the condition that has to be satisfied by the property.

3.2 Part B - Formula Automaton

This part pertains to the specification being represented in the form of a LTL formula which is then converted to a formula automaton.

3.2.1 Module 4 - LTL Operators and Formulae

The operator is modelled as a class where each operator has the operator no., a flag which specifies whether the operator is unary or binary and two instances of the class LTL Expression. Methods to initialise the operator and build the sub formula is specified in this module.

The expression is modelled as a class, it has the variables, operators, the number of operators, the number of variables, the formula and pointer to next formula. The method BuildExpression is used to build the sub formulae initially and then combine them into the final composite LTL expression. This expression is used to generate the Buchi automaton.

3.2.2 Module 5 - Formula Automaton Construction

The Gerth, Peled, Vardi, Wolper algorithm is used to convert linear temporal logic formulae to automata using simple on-the-fly construction. It consists of a set of nodes which contain a set of logical formulae.

3.3 Part C - Product Automaton

This part pertains to the construction of the product automata from the service automaton and the specification negation formula automaton and model checking.

3.3.1 Module 6 - Model Checking

The service and the formula automata are converted to Buchi automata. The product automaton is generated and it is checked for accepting cycles using Depth First Search algorithm to check whether the system meets the specification.

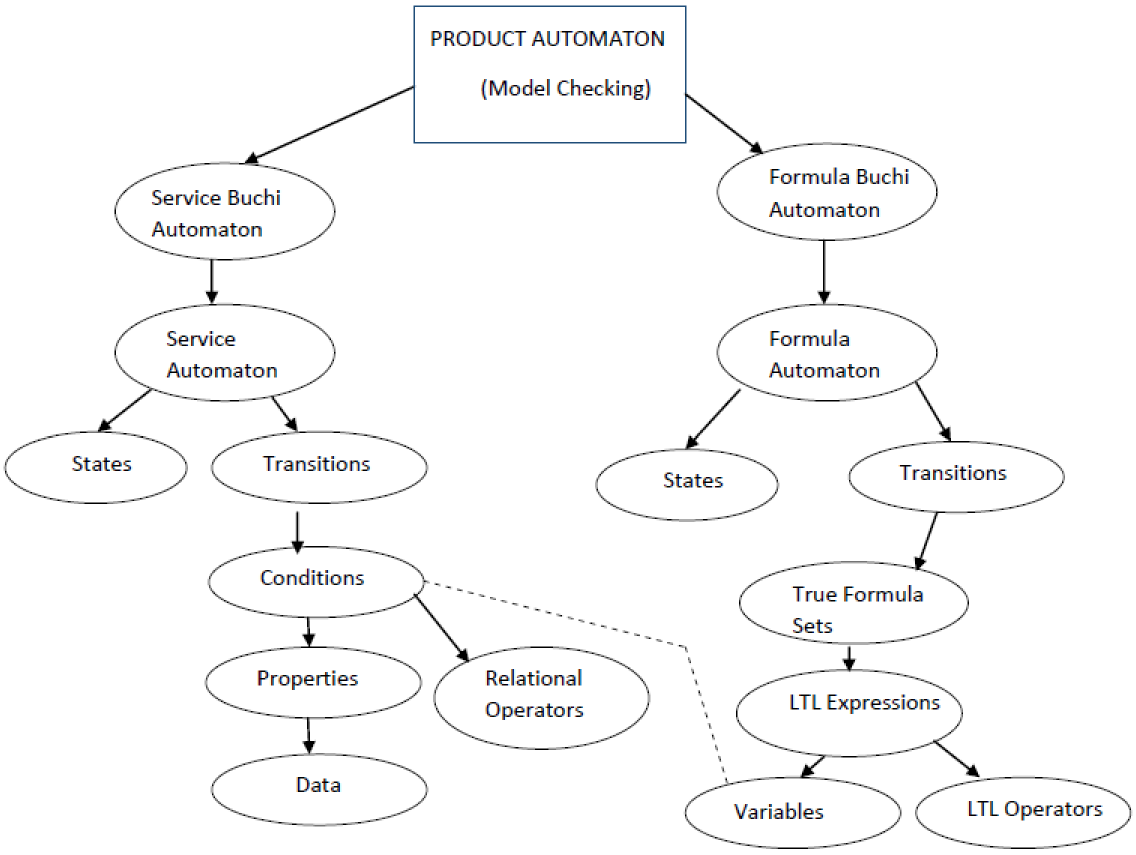


FIGURE 3.1: System Architecture

CHAPTER 4

Service Automaton

Three different modules are used to construct and represent the service automaton. The data and properties of the service are defined first. The collection of these properties are used to define the service and finally, the service is converted to a set of states and transitions.

4.1 Data and Properties

This module is serves two functions to define the different properties of a service system and to define the data types needed to assign values to these properties.

4.1.1 Data

The data class in the first module is used to define data types for the service system. The data types used are Integer, Decimal, Text and Date. The values of most of the properties of web services can be defined using the above four data types or a combination of such data types. Even though it is possible to store all kinds of data as text, using multiple data types is more advantageous in case of computation and comparison. The data types Integer and Decimal have been implemented using simple data types in the language while Text uses complex string data type. The Date data type has been modelled as a class.

4.1.2 Properties

The properties are defined as the Service Input class in the first module. Every web service has a certain number of properties which define it. The values of these properties control the ability of a web service to fulfil certain requirements. Each property is identified by a unique name and is of a particular data type. Every property has a certain value at each point of time.

4.1.3 Example

The following are examples of properties of web services:

Arrival-Date is a property of a web service such as Flight Booking.

It has a data type of Date.

Example Value: Day 24, Month 3, Year - 2015

Seats-Available is a property of a web service such as Train Reservation.

It has a data type of Integer.

Example Value: 25 seats available

4.2 Service Systems

A service is a self-functional unit which can be discretely invoked. A web service is defined as method of communication between a client and a server over a network. As mentioned before, every web service has a number of properties which define it. The client has certain requirements and makes use of a web service based on its properties and their values. These properties and their values are maintained on the server side. The client communicates with the server to request the service based on these properties. Each service consists of a list of properties, the number of properties and a unique service name.

4.2.1 Example

The following is an example of a web service in its simplest form i.e. a collection of properties:

Service Name: Train Reservation Service

Number of Properties: 4

List of Properties: Departure Time (Integer), Track Number (Integer), Train Name (Text), Seats Available (Integer)

4.3 Service Automaton Construction

A service automaton consists of all the possible states in which the web service can be seen in. The values of the properties of the service are important in defining each state. The changes in values of these properties are the transitions which lead from one state to another. Since, the service does not exist as a formula or any other mathematical formats; it can be directly represented as an automaton.

The components of the service automaton are the set of states and transitions, the number of states and transitions and also the name of the service that the automaton represents.

4.3.1 States

Every state belongs to a particular service which can be identified by the service name. Moreover, each state has its own name which has to be unique in that particular service. Any state of the automaton is of four basic types with respect to its position in the automaton. It can be a start state or an end state or neither or sometimes, even both. This information is also present along with each state.

4.3.2 Transitions

The client accessing the service finds the service at a particular state. The same client finds the service to be at different states at different times. This means that the state of the service changes with respect to time. This is because the values of the service properties changes with respect to time. The change in properties is represented in the form of a transition. The transition occurs because a particular condition with respect to a property has been satisfied. The definition of a transition means that it leads one state to another. This means that it has an input state - a state in which the system existed in before the transition condition was satisfied and an output state a state in which the system exists after the transition condition has been satisfied. Each transition has to be a part of a service which means the service name to which the transition belongs to must also be known.

4.3.3 Conditions for Transitions

Every transition occurs only when a certain condition triggers it. These conditions are the range of values of the properties of the web service system. In order to represent the conditions, the deciding property along with the boundary value is stored as a pair. Relational operators are used to represent the value range.

The relational operators used are:

EQUAL TO (=)

LESSER THAN

GREATER THAN

NOT EQUAL TO (!=)

GREATER THAN OR EQUAL TO $\geq$

LESSER THAN OR EQUAL TO $\leq$

By principle of duality, NOT EQUAL TO is the dual of EQUAL TO, GREATER THAN OR EQUAL TO is the dual of LESSER THAN, and LESSER THAN OR EQUAL TO is the dual of GREATER THAN. This means that the negation of the condition can be found by using the dual form of the relational operator.

4.3.4 Steps for Construction

1. Begin with the start state.

2. Find out all the conditions along with the deciding properties and values. Name each condition.

3. Generate the negation of these conditions using the principle of duality.

4. Create the different states which can be reached by the service. Name each state.

5. Assign whether a state is a finish state or not.

6. Create transitions using the existing states and conditions.

4.3.5 Example

The following is an example of a web service.

Web Service Name: Flight Reservation System

Properties: Flight Number (Integer), Destination (Text), Departure Date (Date), Departure Time (Integer), Cost (Decimal)

Conditions:

Destination = Chennai (A)

Departure Time $\geq$ 1200 (B)

Cost $\leq$ 20000.00 (C)

States:

Start

To Chennai

Departing Afternoon

Within Budget

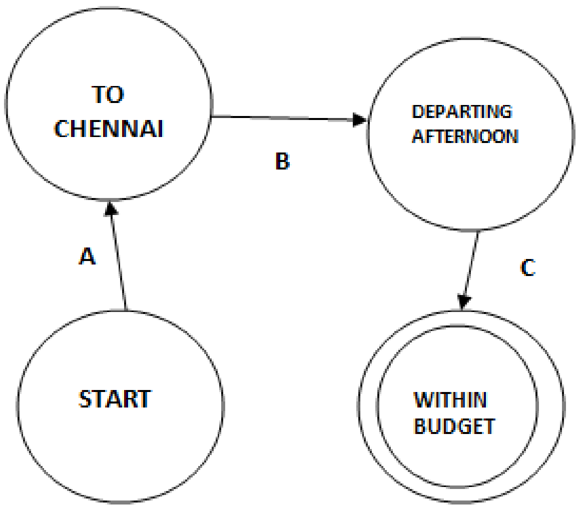


FIGURE 4.1: Service Automaton

CHAPTER 5

Formula Automaton

In order to represent the specifications, linear temporal logic formulae are used. These formulae have to be converted to the formulae automaton. For this we use the simple on-the-fly construction method. For model checking purposes, we find the negation of the specification and convert the negation formula to an automaton.

5.1 LTL Operators and Formulae

The conditions which are represented in the service automata can be combined together by propositional logic formulae. However, these conditions have to be checked with respect to time. The state of the service is not static, and that is why it is represented as an automaton. Hence the specifications also have to be represented with respect to time. That is why linear temporal logic is used.

5.1.1 Linear Temporal Logic

Linear temporal logic or linear-time temporal logic (LTL) is a modal temporal logic with modalities referring to time. LTL is used to represent the formulas for the specification. LTL consists of temporal operators such as:

Next (X) This is a unary operator. Next(A) implies that the variable A has to be true at the next state and may or may not be true in the present state.

TABLE 5.1: LTL Operators

SYMBOL	OPERATOR	TYPE
X	neXt	Unary
G	Globally	Unary
F	Future	Unary
U	Until	Binary
R	Release	Binary

Until (U) This is a binary operator. A Until B implies that the condition A has to hold until the condition B holds.

Release (R) This is a binary operator. A Release B implies that the condition B has to until the condition A holds and hold at the state, condition A first holds true. This operator is a dual of the operator Until.

Globally (G) This is a unary operator. Globally(A) implies that the condition A has to hold true at every state.

Finally (F) This is a unary operator. Finally(A) implies that the condition A will hold true at some state in the subsequent path.

In addition to the temporal operators, the operators from propositional logic such as AND, OR and NOT are also used to represent the formulae.

5.1.2 LTL Operators

The LTL operators are represented in the form of a class. Depending on whether they are unary or binary, they consist of one or two LTL expressions as their variables.

5.1.3 LTL Expressions

The LTL expressions consist of the list of operators in the expression, the list of variables in the expression, the number of operators and variables and the formula generated by the variables and operators in the expression. The formula is represented in form of a tree.

5.1.4 Tree Representation of LTL Formulae

Every LTL formula is represented as a tree. Each operator is a node and the variables they hold are the leaves. In the case of complex formulae, operators exist as children of other operator nodes. In case of the simplest expression which is the single variable, there are no operators and the root of the tree is the variable itself. In the formula tree, every binary operator has two children nodes, every unary operator has one child node and every variable is a leaf and hence has no child nodes. For a LTL formula to exist there must be at least one variable in the expression.

5.1.5 Steps for Construction

1. Create the formula from the root operator or in the case of the simplest formula, a single variable.

2. Depending on whether the operator is unary or binary, expand every operator to create the child nodes.

3. If another operator is encountered, repeat step 2 until all the leaf nodes are variables.

4. Perform inorder traversal of the tree to generate the formula.

5.1.6 Example

Consider the LTL formula:

F (A U (B v C))

This formula means that eventually, the condition A has to hold until either B or C hold.

Construction of the Formula Tree:

1. Identify the root of the formula tree. In this case, it is Finally.

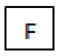


FIGURE 5.1: Tree - Step 1

2. Since, it is a unary operator it has only one child node.

3. The child of the node F will be the operator U.

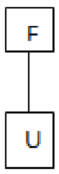


FIGURE 5.2: Tree - Step 3

4. U is a binary operator and hence it has two child nodes.

5. The children of U are the variable A and the operator OR.

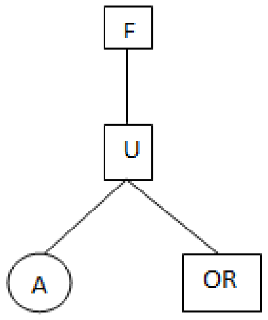


FIGURE 5.3: Tree - Step 5

6. The operator OR has to be expanded next.

7. Since it is a binary operator, it has two children, the variables B and C.

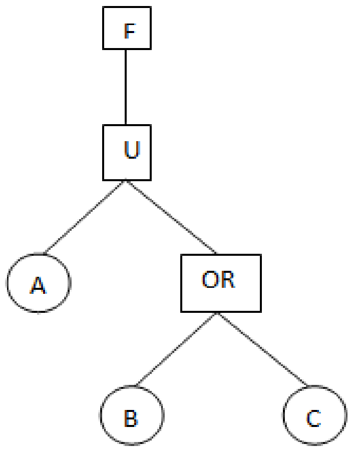


FIGURE 5.4: Tree - Step 7

The above tree represents the formula:

F (A U (B v C))

The circled nodes are the variables which are the leaves of the tree.

The boxed nodes are the operators.

The binary operators have two child nodes.

The unary operators have a single child node.

The variables have no child nodes.

5.2 Formula Automaton Construction

The formula automaton is constructed from the LTL formula using the simple On-the-Fly construction method. This method is advantageous because it eliminates unreachable states of the automaton, saves space and also, is faster than the original construction method. To prove the advantages of using the On-the-Fly method, the original method of construction is shown along with the method used.

5.2.1 Original Construction Method

The original construction method involves checking the truth of each condition at every generated state. Furthermore, all the possible states are generated whether they are reachable or not. This means there will be an exponential increase in the number of states at every step in the path. As the size of the formula increases, the size of the corresponding automaton increases exponentially which means there will be a high time and space complexity which will lead to an inefficient algorithm. Moreover, the formulas which hold true at the past states also have to be stored as they could impact the construction of the automata. This is because we may have to go back to a previous state if we are not able to continue the construction of the automata through a particular path.

5.2.2 On-the-Fly Construction Method

The simple on-the-fly construction method was developed by Gerth, Peled, Vardi and Wolper. It consists of a set of nodes which contain a set of logical formulae. Each node has the following:

Name: A string that is the name of the node.

Incoming: The incoming edges represented by the names of the nodes with an outgoing edge leading to the current node. A special name, init is used to mark initial nodes. init is not the name of any node, hence does not represent a real edge.

New: A set of temporal properties (formulas) that must hold at the current state and have not yet been processed.

Old: The properties that must hold in the node and have already been processed. Eventually, New will become empty, leaving all the obligations in Old.

Next: Temporal properties that must hold in all states that are immediate successors of states satisfying the properties in Old.

Father: During the construction, nodes will be split. This field will contain the name of the node from which the current one has been split. This field is used for reasoning about the correctness of the algorithm only, and is not important for the construction.

We keep a list of nodes Nodes Set whose construction was completed, each having the same fields as above.

The two integral methods used are drawgraph, which is a function used to specify the initial state of the automata and calls the expand function. The expand function is a recursive function and is used to generate all the states of the automata from the initial state and the set of logical formulae.

5.2.3 The Algorithm

```
1 record graph node = [Name:string, Father:string, Incoming:set of string,
2 New:set of formula, Old:set of formula, Next:set of formula];
3 function expand (Node, Nodes Set)
4 if New(Node)= $\emptyset$ then
5 if $\exists$ ND $\in$ Nodes Set with Old(ND)=Old(Node) and Next(ND)=Next(Node)
6 then Incoming(ND) = Incoming(ND) $\cup$ Incoming(Node);
7 return(Nodes Set);
8 else return(expand([Name$\Rightarrow$ Father$\Rightarrow$ new name(),
9 Incoming$\Rightarrow$ Name(Node)$\}$, New$\Rightarrow$ Next(Node),
10 Old $\Rightarrow \emptyset$ , Next $\Rightarrow \emptyset$ ], Node$\cup$ Nodes Set)
11 else
12 let $\eta \in$ New;
13 New(Node) := New(Node) $\setminus \eta$;
14 case $\eta$ of
15 $\eta$ = Pn, or $\neg$ Pn or $\eta$ = T or $\eta$ = F $\Rightarrow$
16 if $\eta$ = F or Neg($\eta$) $\in$ Old(Node) (* Current node contains a contradiction
*)
17 then return(Nodes Set) (* Discard current node *)
18 else Old(Node):=Old(Node) $\cup \{\eta\}$;
19 return(expand(Node, Nodes Set));
```

```
20 $\eta = \mu$ U $\psi$ , or $\mu$ V $\psi$, or $\mu$ v $\psi \Rightarrow$
21 Node1:=[Name $\Rightarrow$ new name(), Father $\Rightarrow$ Name(Node), Incoming $\Rightarrow$ Incoming(Node),
22 New $\Rightarrow$ New(Node) $\cup$ New1($\eta$ )$\}$ $\setminus$ Old(Node)),
23 Old $\Rightarrow$ Old(Node) $\cup$ $\{ \eta \}$, Next=Next(Node) $\cup$ $\{$Next1($\eta$ )$\}$ ];
24 Node2:=[Name $\Rightarrow$ new name(), Father $\Rightarrow$ Name(Node), Incoming $\Rightarrow$ Incoming(Node),
25 New $\Rightarrow$ New(Node) $\cup$ New2($\eta$ )$\}$ $\setminus$ Old(Node)),
26 Old $\Rightarrow$ Old(Node) $\cup$ $\{ \eta \}$, Next=Next(Node) $\cup$ $\{$Next2($\eta$ )$\}$ ];
27 return(expand(Node2, expand(Node1, Nodes Set)));
28 $\eta = \mu \wedge \psi \Rightarrow$
29 return(expand([Name $\Rightarrow$ Name(Node), Father $\Rightarrow$ Father(Node),
30 Incoming $\Rightarrow$ Incoming(Node), New $\Rightarrow$ New(Node)$\cup$ ($\{ \mu , \psi \}$ $\setminus$ Old(Node)),
31 Old(Old(Node) $\Rightarrow$ $\{ \eta \}$, Next=Next(Node)], Nodes Set))
32 end expand;
33 function create graph ($\varphi$ )
34 return(expand([Name $\Rightarrow$ Father $\Rightarrow$ new name(), Incoming $\Rightarrow$ init,
35 New $\Rightarrow$ $\{ \varphi\}$ , Old $\Rightarrow \emptyset$ , Next $\Rightarrow \emptyset$ ], $\emptyset$))
36 end create graph;
```

5.2.4 Rules for On-the-Fly Construction

1. The formula must be in negation normal form.

Eg. !(a∧b) is not allowed. !av!b is allowed.

2. The formula can contain the operators NOT, AND, OR, UNTIL, GLOBAL, FINAL, RELEASE and NEXT.

3. FINAL (F) formulas are converted to a different form during implementation.

F (a) = True Until a

4. Similarly for GLOBAL (G)

G (a) = False Release a

5. For OR, UNTIL and RELEASE a single node is branched into two separate nodes for converting to an automaton.

6. Expansion of the nodes takes place till every possible state of the automaton which can be reached is a part of the node set.

7. Since the construction is on-the-fly, not all the nodes generated are needed. The nodes which refer to unreachable states can be discarded.

8. The temporal logic formulas can be simplified to give an automaton with lesser states, though this does not always hold good.

5.2.5 Kripke Structure

The On-the-Fly construction generates all the nodes of the graph from the LTL formula. The Old field of the node contains all the formulas and sub-formulas which hold true in that particular node. However, this is not yet an automaton.

The present structure of the graph is called as a Kripke structure which is just a state graph. In order to generate the automaton, we need to add transitions to the structure.

5.2.6 Kripke Structure to Automaton

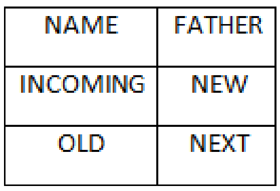
The Kripke structure has to be converted to the formula automaton. In order to do this we take each node and find out the incoming nodes. We add a transition from each incoming node to the present node. As the conditions for the transitions we add the formulas which must hold in the present state which are found in the Old field of the node data structure. The formulae which hold true are present as a number of sets. If any one of the sets completely holds, then the transition can occur.

5.2.7 Example

Consider the LTL formula:

A Until B

The node of the graph is represented as follows:



NAME	FATHER
INCOMING	NEW
OLD	NEXT

FIGURE 5.5: Graph Node

1. The initial node N1 is created with New formula A U B and the Old and Next fields as empty.

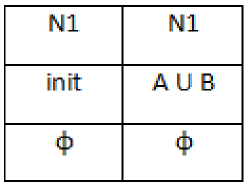


FIGURE 5.6: Graph - Step 1

2. The expand function is called with the node to be expanded as N1. Since the operator encountered is Until, the node is split into two and the formula is removed from N1.

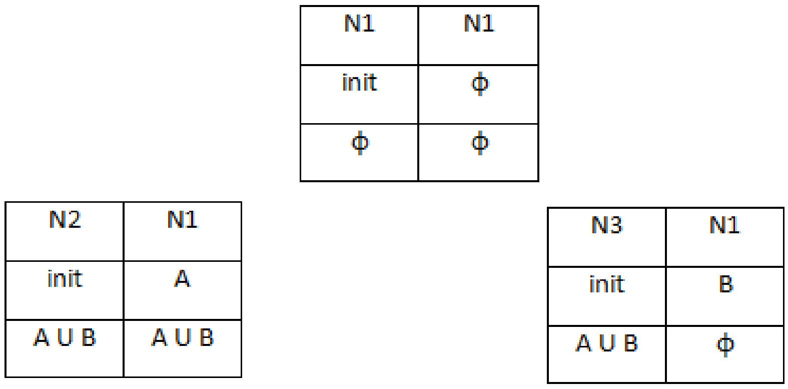


FIGURE 5.7: Graph - Step 2

3. Expansion continues with respect to N2. Since the formula in New of N2 is a simple variable, it is directly added to the Old field and expansion continues with respect to N2.

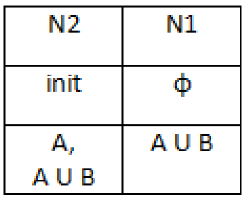
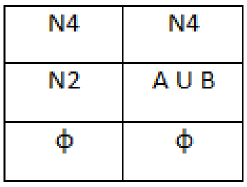


FIGURE 5.8: Graph - Step 3

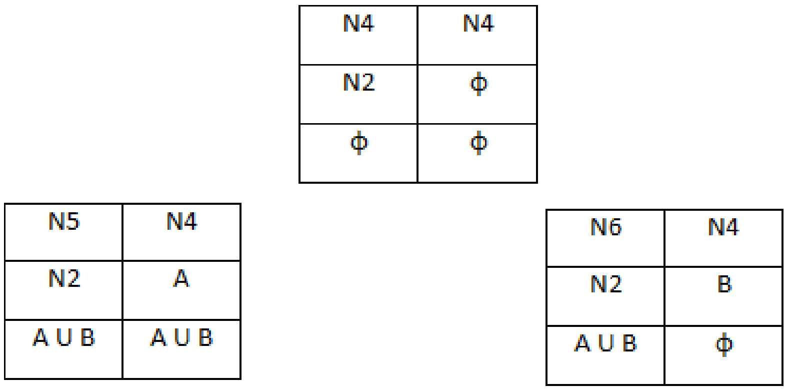
4. The New formula in N2 is empty. Hence N2 can be added to the Node Set provided there is no other node already existing with the same Old and Next fields. Since Node Set is empty, N2 is added to the Node Set and a new node N4 is created with the New field of N4 same as the Next field of N2.



N4	N4
N2	A U B
Φ	Φ

FIGURE 5.9: Graph - Step 4

5. Since the New formula in N4 is an Until formula, the node is split into two.



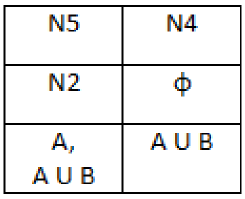
N4	N4
N2	Φ
Φ	Φ

N5	N4
N2	A
A U B	A U B

N6	N4
N2	B
A U B	Φ

FIGURE 5.10: Graph - Step 5

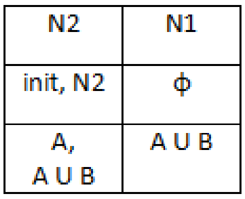
6. The expanded node is N5. Since A is a simple formula, it is directly processed to Old.



N5	N4
N2	Φ
A, A U B	A U B

FIGURE 5.11: Graph - Step 6

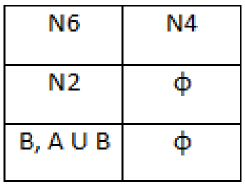
7. Since the New field of N5 is empty it can be added to the Node Set. However, the node n2 has the same Old and Next fields. This means that N5 is a duplicate node. The incoming of N5 is added to the incoming of N2.



N2	N1
init, N2	ϕ
A, A U B	A U B

FIGURE 5.12: Graph - Step 7

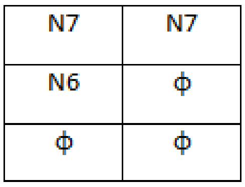
8. Since the nodes cannot be expanded further in this path, we back track and reach the node N6 which was one of the nodes created by splitting N4. Since N6 has only a simple formula in its New field, we can directly process it to the Old field.



N6	N4
N2	ϕ
B, A U B	ϕ

FIGURE 5.13: Graph - Step 8

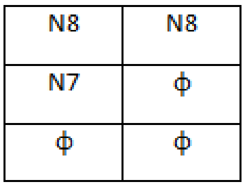
9. We create the node N7 with the Next of N6 as New field of N7 since the New field of N6 is empty. Since the New field of N6 is empty, we can add it to the Node Set. No other node in the Node Set has the same Old and Next fields.



N7	N7
N6	ϕ
ϕ	ϕ

FIGURE 5.14: Graph - Step 9

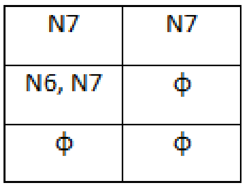
10. We create the node N8 with the Next of N7 as New field of N8 since the New field of N6 is empty. Since the New field of N7 is empty, we can add it to the Node Set. No other node in the Node Set has the same Old and Next fields.



N8	N8
N7	ϕ
ϕ	ϕ

FIGURE 5.15: Graph - Step 10

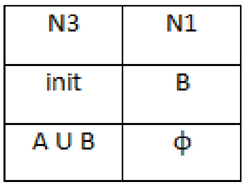
11. The node N8 can be added to the Node Set as its New field is empty. However, the node N7 has the same Next and Old fields in the Node Set. Hence, the node N8 is a duplicate. The incoming nodes of N8 are added to N7.



N7	N7
N6, N7	ϕ
ϕ	ϕ

FIGURE 5.16: Graph - Step 11

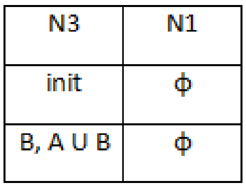
12. Since the expansion cannot continue in the present path, we back track and reach node N3 which was created during the splitting of node N1.



N3	N1
init	B
A U B	ϕ

FIGURE 5.17: Graph - Step 12

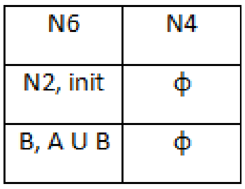
13. Since there is only a simple formula in the New field of N3, it can be directly processed to Old.



N3	N1
init	ϕ
B, A U B	ϕ

FIGURE 5.18: Graph - Step 13

14. Since the New field of N3 is empty, we can add the node to the Node Set. However the node N6 has the same Old and Next field values and hence N3 need not be added to the Node Set. The incoming of N3 is added to N6.

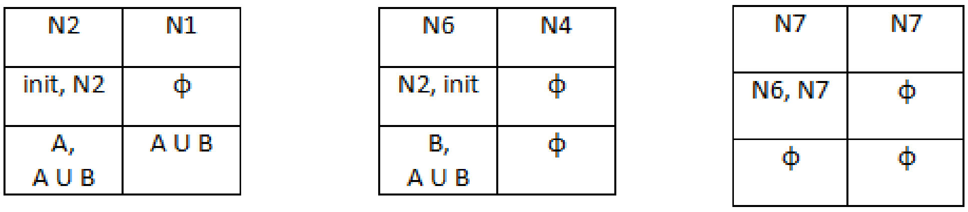


N6	N4
N2, init	ϕ
B, A U B	ϕ

FIGURE 5.19: Graph - Step 14

No further expansion can take place.

Final Node Set: N2, N6, N7, init



N2	N1
init, N2	ϕ
A, A U B	A U B

N6	N4
N2, init	ϕ
B, A U B	ϕ

N7	N7
N6, N7	ϕ
ϕ	ϕ

FIGURE 5.20: Nodes after Construction

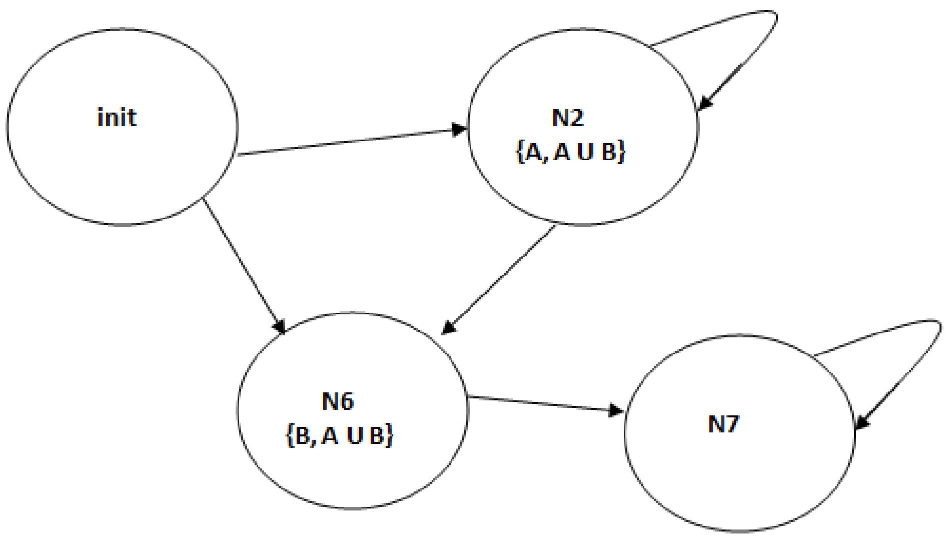


FIGURE 5.21: Generated Kripke Structure

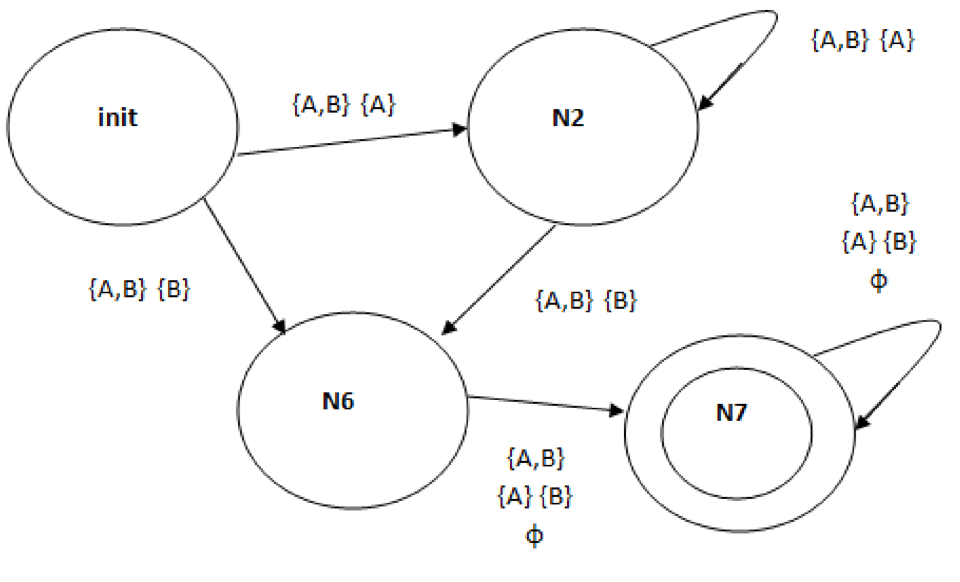


FIGURE 5.22: Formula Automaton

CHAPTER 6

Product Automaton

After the service automaton has been constructed, the specification is negated and converted to the formula automaton. After this, the product of these two automata has to be found in order to perform model checking. While, the construction of both the automata is independent, it is the product which actually determines whether the system meets the specification.

6.1 Model Checking

Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. In order to perform model checking, automata theoretic methods can also be used. In order to perform model checking, both the service automaton and the formula automaton have to be converted to the same form of automata.

6.1.1 Buchi Automaton

In computer science and automata theory, a Buchi automaton is a type of automaton, which extends a finite automaton to infinite inputs. It accepts an infinite input sequence if and only if there exists a run of the automaton that visits (at least) one of the final states infinitely often. Buchi automata are often used in model checking as an automata-theoretic version of a formula in linear temporal logic.

Formally, a deterministic Buchi automaton is a tuple A = (Q,Σ,δ,q_0,F) that consists of the following components:

- Q is a finite set. The elements of Q are called the states of A.
- Σ is a finite set called the alphabet of A.
- $\delta : Q \times \Sigma \rightarrow Q$ is a function, called the transition function of A.
- q_0 is an element of Q, called the initial state.
- F ⊆ Q is the acceptance condition. A accepts exactly those runs in which at least one of the infinitely often occurring states is in F.

Both the service automata and the negation of the formula automata are converted to Buchi automata to perform model checking.

6.1.2 Finding Product Automaton

Once both the service and the negation of the formula automata have been converted to Bchi automata, their product can be found out for model checking purposes. In order to find the product, pairs of states are formed with a state from each automaton. When the same transition condition leads to different states from the state pair, a new state pair can be formed. Thus these state pairs form the states of the product automaton. The transitions remain the same as the other two automata. For model checking, we consider all the states of the service automaton as accepting states and base the accepting states of the product automaton with respect to the accepting states of the formula automaton.

If the service and the negation of the formula automata have m and n states respectively, the product automaton can have a maximum of m*n states and a minimum of zero states.

6.1.3 Checking for Accepting Cycles

To check if the system meets the specification we find the product of the service automaton and the negation of the formula automaton. The resultant product automaton has a set of states and transitions. Some of the states are accepting states. To check if the system meets the specification, we need to check for accepting cycles i.e. a cycle in the graph consisting of an accepting state. If such a cycle exists, the system does not meet the specification. If it does not exist, the system meets the specification.

Depth First Search (DFS) traversal of graphs is used for checking for cycles and if the cycle consists of an accepting state, the product automaton has an accepting cycle.

6.1.4 Depth First Search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. In this case, DFS checks for cycles in a directed graph. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by DFS.

6.1.5 Steps for Model Checking

1. Generate the system automaton from the web service.

2. Negate the specification formula.

3. Generate the formula tree from the negated LTL formula.

4. Construct the Kripke structure using On-the-Fly methods.

5. Construct the formula automaton.

6. Mark all the states of the system automaton as accepting.

7. Convert the system automaton to Bchi form.

8. Convert the formula automaton to Bchi form.

9. Find the product automaton using the system and formula automata.

10. Run DFS on the product to check for cycles.

11. If a cycle does not exist, the system meets the specification. Otherwise, if there is no accepting state in the cycle, the system meets the specification. Otherwise, the system does not meet the given specification.

6.2 Case One

The following example shows model checking of a Ticket Booking web service system.

Web Service Name: Ticket Reservation

Properties: Destination (Text)

DepartureTime (Integer)

Cost (Decimal)

BookingStatus (Integer)

System Automaton:

States: Start [ST]

To Chennai [CH]

Afternoon Train [AT]

Within Budget [WB]

Ticket Confirmed [TC] (Final)

Conditions: Destination = Chennai [A]

DepartureTime $\geq$ 1200 [B]

Cost $\leq$ 5000.0 [C]

BookingStatus = 1 [D]

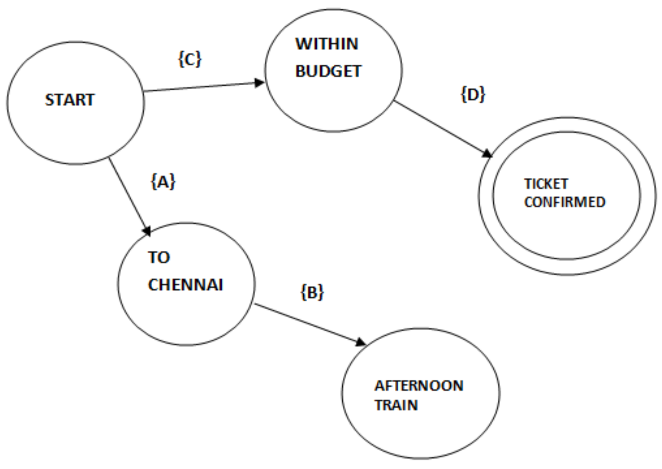


FIGURE 6.1: System Automaton - Case 1

Formula Automaton:

Specification: Ticket should not be confirmed until the cost is found to be within budget.

Formula: Not D Until C (!D U C)

Negation: [D U !C]

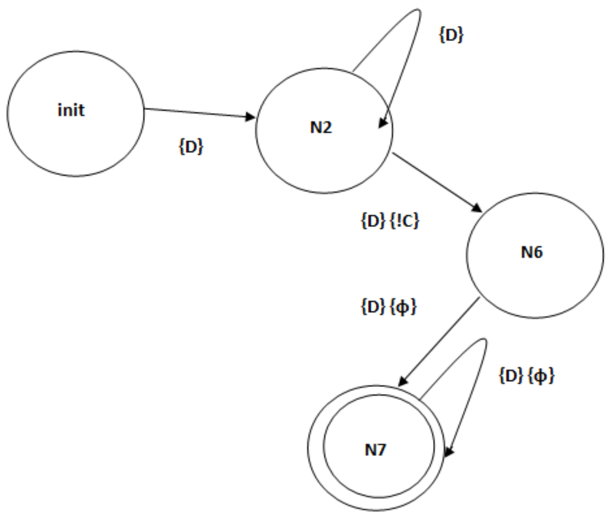


FIGURE 6.2: Formula Automaton - Case 1

Product Automaton:

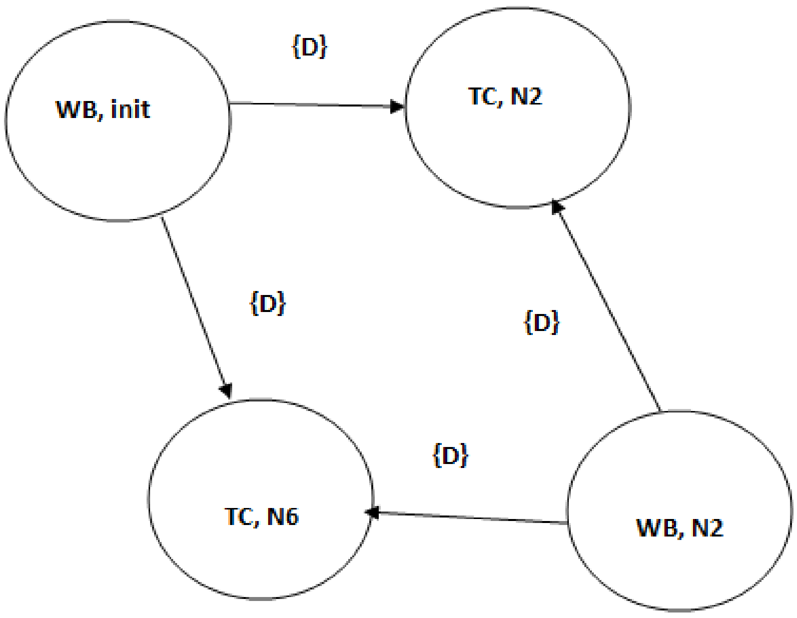


FIGURE 6.3: Product Automaton - Case 1 Part 1

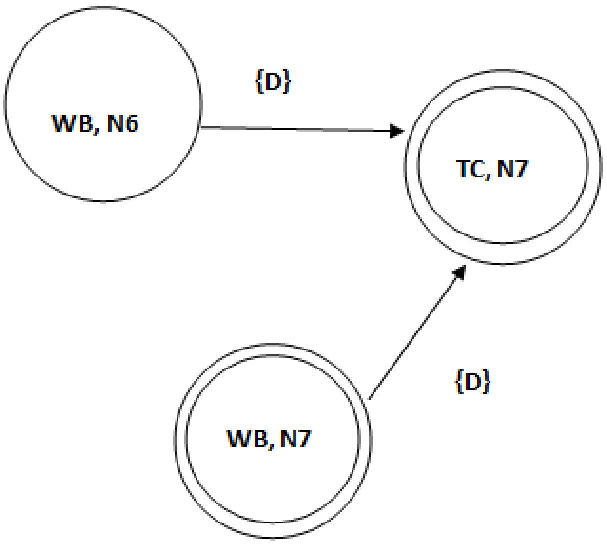


FIGURE 6.4: Product Automaton - Case 1 Part 2

The product automaton consists of two unconnected graphs which do not contain any accepting cycles.

Hence the system MEETS the specification.

6.3 Case Two

The following is the model checking of a task scheduler web service system.

Web Service Name: Task Scheduler

Properties: SystemBusyStatus (Integer)

P1Status (Integer)

P2Status (Integer)

System Automaton:

States: Start [ST]

P1

P2 (final)

Conditions: SystemBusyStatus = 1 [A]

P1Status = 1 [B]

P2Status = 1 [C]

Formula Automaton:

Specification: System status should be busy till process 1 is completed.

Formula: Not A Until B [!A U B]

Negation: [A U !B]

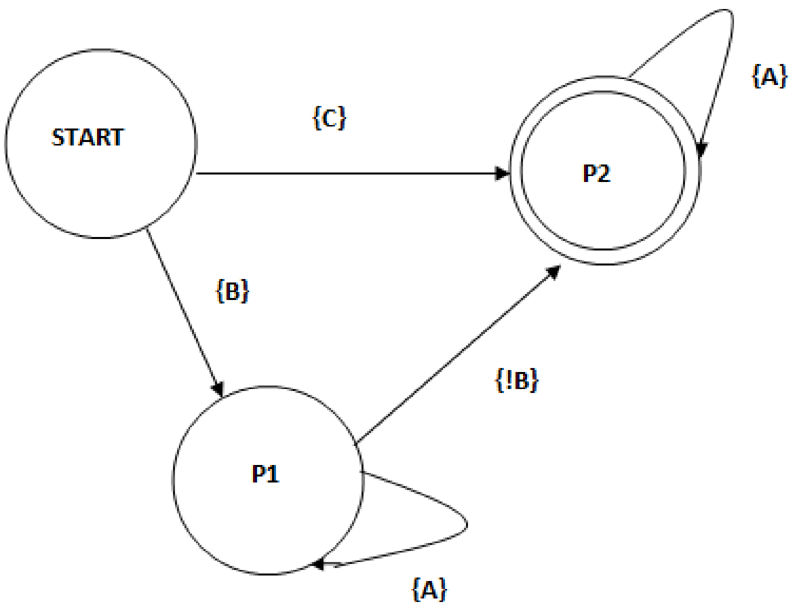


FIGURE 6.5: System Automaton - Case 2

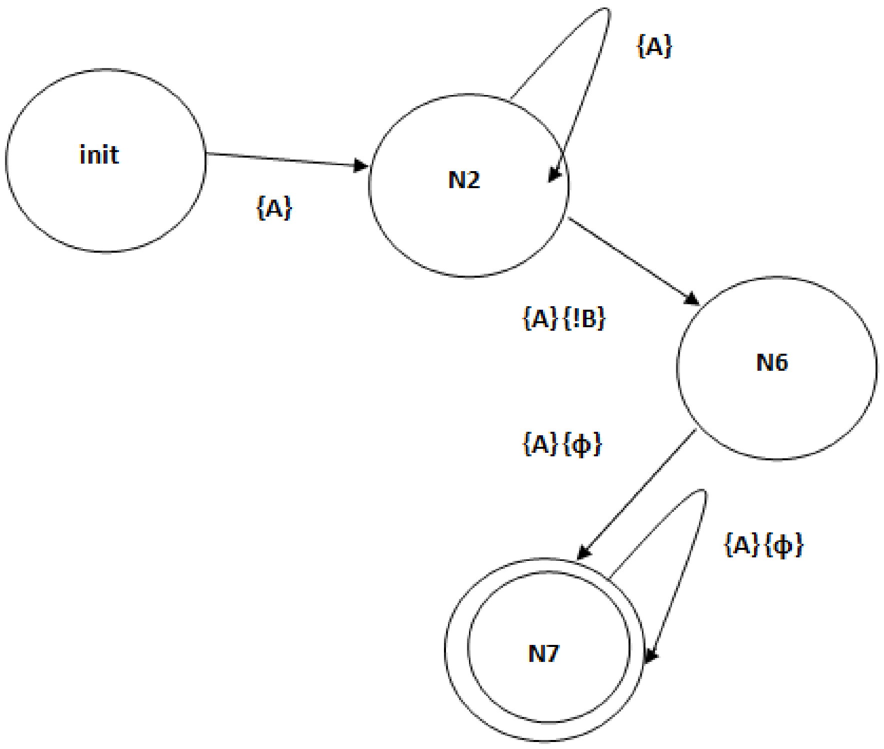


FIGURE 6.6: Formula Automaton - Case 2

Product Automaton:

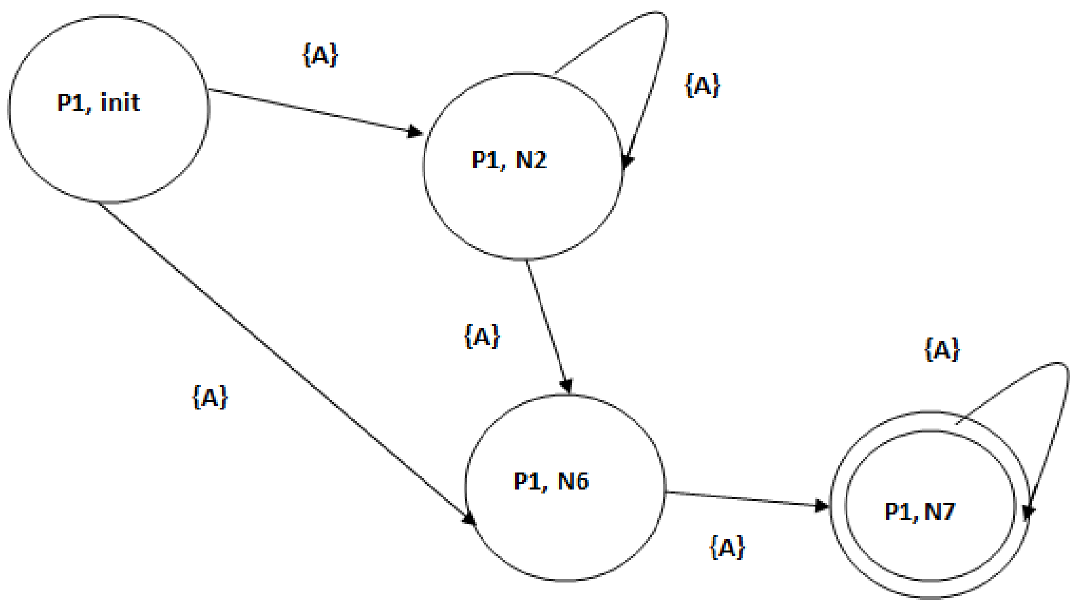


FIGURE 6.7: Product Automaton - Case 2

The above product automaton consists of an accepting cycle.

Hence, the system DOES NOT MEET the specification.

CHAPTER 7

Conclusion and Future Work

The proposed system for automata theoretic model checking of web service systems can be used for checking whether any web services conforms to a specification. The system can be used for any general web service and is not specific to a particular type of web service. Properties of any web service can be easily modelled. Moreover, using automata makes the model checking faster and easier to implement since an algorithm as simple as depth first search is only needed to check for accepting cycles. Also, generating the specification automaton using on-the-fly technique makes it a faster system than most methods existing. Thus this system can be used to perform automata theoretic model checking on web service systems.

For further development, multiple web service systems can be considered and the communications between them can be modelled. For this purpose, system of communicating automata have to be used. Also, the specifications or choreographies have to be described using local temporal logic and have to be modelled as distributed automata. Such as system can be used to check for choreography conformance of web service systems.

REFERENCES

1. Dimitra Giannakopoulou, Flavio Lerda (2001) ’Efficient translation of LTL formulae into Buchi automata’, RIACS Technical Report 01.29

2. Mukund, M. (1997) ‘Linear-Time Temporal Logic and Buchi Automata’, Tutorial talk, Winter School on Logic and Computer Science, ISI, Calcutta.

3. Ramanujam, R. (1996) ‘Locally linear time temporal logic’, In proc. of LICS’96, pp. 118-127.

4. Foster, C., Uchitel, C., Magee, J. and Kramer J. (2003) ‘Model-based verification of web service compositions’, In proc. of ASE’03, pp. 152-163.

5. Shin Nakajima (2009) ‘Model Checking Verification for Reliable Web Service’, Hosei University

6. Sankar Gurumurthy, Orna Kupferman, Fabio Somenzi and Moshe Y. Vardi (2003) ’On Complementing Nondeterministic Buchi Automata’ CHARME: 96-110

7. Meenakshi, B. (2004) ‘Reasoning about distributed message passing systems’, UNM TH81.

8. R. Gerth, D. Peled, M. Y. Vardi and P. Wolper (1995) ’Simple On-the-fly Automatic Verification of Linear Temporal Logic’, In Protocol Specification Testing and Verification : 3-18

9. Sheerazuddin, S. (2013) ‘Temporal specifications of client-server systems and unbounded Agents’, HBNI TH49.