CTL Model Checking with Java Path Finder

Submitted

In Partial Fulfilment of the Requirements for the Degree of

BACHELOR OF TECHNOLOGY

By

Prateek Kumar Sinha (20063032)
Sharad Jain (20064078)
Ankur Srivastava (20065016)
Hitesh Das (20054030)
Vivek Kumar Singh (20064083)

Under the guidance of Mr. D. K. Yadav
Associate Professor
Department of Computer Science and Engineering
MNNIT Allahabad

Department of Computer Science and Engineering
Motilal Nehru National Institute of Technology
Deemed University, Allahabad – 211004

UNDERTAKING

We declare that the work presented in this report titled "**CTL Model checking with Java Path Finder** ", submitted to the Computer Science and Engineering Department, Motilal Nehru National Institute of Technology, Allahabad, for the award of the ***Bachelor of Technology*** degree in ***Computer Science and Engineering***, is our original work. We have not submitted the same work for the award of any other degree. In case this undertaking is found incorrect, we accept that our degree may be unconditionally withdrawn.

Date : 07, May, 2010

Place : Allahabad

Prateek KumarSinha(20063032)

Sharad Jain (20064078)

Ankur Srivastava (20065016)

Hitesh Das (20064030)

Vivek Kumar Singh (20064083)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology

Allahabad

CERTIFICATE

This is to certify that the project entitled "**CTL Model checking with Java Path Finder**" submitted by Prateek Kumar Sinha (20063032), Sharad Jain (20064078), Ankur Srivastava (20065016), Hitesh Das (20064030) and Vivek Kumar Singh (20064083) in partial fulfillment of the requirement for the degree of **Bachelor of Technology** in **Computer Science and Engineering** at Motilal Nehru National Institute of Technology (Deemed University), Allahabad, during the academic year 2008-09 , is a bonafide record of their own work and has not been presented anywhere else.

Date: 07, May, 2010

Mr. D. K. Yadav
Lecturer,
Department of Computer Science and Engineering
Motilal Nehru National Institute of Technology
Allahabad - 211004

Synopsis

In this project, CTL model checking of a given Java Byte Code has been performed. The model has been extracted by implementing listeners in java path finder. The properties have been specified using CTL. The model checking has been performed by implementing model checking algorithm. Moreover, the visual representation of the model can be viewed with the help of Graphviz.

For this purpose we are using JPF tool provided by NASA. The JPF modes were used to verify Java programs basically to,

- Find and explain defects
- Collect "deep" runtime information like coverage metrics
- Deduce interesting test vectors and create corresponding test drivers

Acknowledgement

We are grateful to Mr. D.K. Yadav for his revered guidance and encouragement, which led to the completion of this project. Without his constant appraisal and efforts, this task would have been a mere dream. He was always there to help us throughout this project. He provided us with all the necessary resources and guidance during the project which helped us to complete the project successfully.

Finally, we deem it a great pleasure to thank one and all who helped us directly or indirectly in carrying out this project work. We are also thankful to our colleagues and friends for their support.

Date: 07, May, 2010

Prateek Kumar Sinha
(20063032)

Sharad Jain (20064078)

Ankur Srivastava (20065016)

Hitesh Das (20064030)

Vivek Kumar Singh (20064083)

Contents

—

List of Figures

Chapter 1

—

Introduction

—

This chapter introduces the project in context of Model checking in brief. In addition an introduction to Java Path Finder is also addressed.

CTL model checking is a formal verification technique. In order to solve such a problem algorithmically, both the model of the system and the specification are formulated in some precise mathematical language. To this end, it is formulated as a task in logic, namely to check whether a given structure satisfies a given logical formula. The concept is general and applies to all kinds of logics and suitable structures. A simple model-checking problem is verifying whether a given formula in the propositional logic is satisfied by a given structure.

The main objective of this project is to perform CTL model checking of a given Java Byte Code by implementing the **Labeling Algorithm** over the Computational Tree with help of **Listeners**; an extension to **JPF**. Although classic SW model checking is only one of these applications, it is still what JPF is mostly associated with. It is to be conjunct herein that, people often confuse this with testing, and indeed JPF's notion of model checking can be close to systematic testing.

1.1 Background

1.1.1 Model Checking

In the field of logic in computer science, **model checking** refers to the following problem: Given a model of a system, test automatically whether this model meets a given specification. Typically, the systems one has in mind are hardware or software systems, and the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash.
In order to solve such a problem algorithmically, both the model of the system and the specification are formulated in some precise mathematical language: To this end, it is formulated as a task in logic, namely to check whether a given structure satisfies a given logical formula. The concept is general and applies to all kinds of logics and suitable structures. A simple model-checking problem is verifying whether a given formula in the propositional logic is satisfied by a given structure.

Fig: 1.1 Model checking of a given code.

An important class of model checking methods has been developed for checking models of hardware and software designs where the specification is given by a temporal logic formula. Pioneering work in the model checking of temporal logic formulae was done by E. M. Clarke and E. A. Emerson and by J. P. Queille and J. Sifakis. Clarke, Emerson, and Sifakis shared the 2007 Turing Award for their work on model checking.

Model checking is most often applied to hardware designs. For software, because of undecidability (see computability theory) the approach cannot be fully algorithmic; typically it may fail to prove or disprove a given property.

The structure is usually given as a source code description in an industrial hardware description language or a special-purpose language. Such a program corresponds to a finite state machine (FSM), i.e., a directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node, typically stating which memory elements are one. The nodes represent states of a system, the edges represent possible transitions which may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution.

1.1.2 Java Path Finder

JPF is an explicit state software model checker for Java byte code. Today, JPF is a swiss army knife for all sorts of runtime based verification purposes.

This basically means JPF is a Java virtual machine that executes your program not just once (like a normal VM), but theoretically in all possible ways, checking for property violations like deadlocks or unhandled exceptions along all potential execution paths. If it finds an error, JPF reports the whole execution that leads to it. Unlike a normal debugger, JPF keeps track of every step how it got to the defect.

What can be checked with JPF?

Fig: 1.2 Java Path Finder Model

Out of the box, JPF can search for

- Deadlocks
- Unhandled exceptions (e.g. NullPointerExceptions and Assertion Errors),

But the user can provide own property classes, or write listener-extensions to implement other property checks. A number of such extensions like race condition and heap bounds checks are included in the JPF distribution

In general, JPF is capable of checking every Java program that does not depend on unsupported native methods. The JPF VM cannot execute platform specific, native code. This especially imposes a restriction on what standard libraries can be used from within the application under test. While it is possible to write these library

versions (especially by using the Model Java Interface - MJI mechanism) there is currently no support for java.awt, java.net, and only limited support for java.io and Java's runtime reflection. Another restriction is given by JPF's state storage requirements, which effectively limits the size of checkable applications to ~10kloc (depending on their internal structure) if no application and property specific abstractions are used. Because of these library and size limitations, JPF so far has been mainly used for
Applications that are models but require a full procedural programming language. JPF is especially useful to verify concurrent Java programs, due to its systematic exploration of scheduling sequences - an area which is particularly difficult for traditional testing.

1.2 Motivation

Certain aspects like scheduling sequences cannot be controlled by a test driver, and require help from the execution environment (VM). Other sources of non-determinism like random input data are supported with special APIs which can significantly ease the creation of test drivers. Simulating non-determinism requires more than just the systematic generation of all non-deterministic choices.

In theory, explicit state model checking is a rigorous method - all choices are explored, if there is any defect, it will be found. Unfortunately, software model checking can only provide this rigor for reasonably small programs since the number of states rapidly exceeds computational limits for complex programs.

JPF addresses this scalability problem in three ways:

- Configurable search strategies.
- Reducing the number of states.
- Reducing state storage costs

JPF has come a long way from its beginnings in 1999. Five major phases stand out

- 1999 Java-to-Promela translator (using Spin as the model checker)
- 2000 JVM / standalone checker
- 2003 design and implementation of extension structure
- 2005 open sourcing of JPF
- 2006 JPF4 with unified Choice Generators and new execution engine

During this time, many people and institutions have worked on and with JPF. The majority of work is still done by the Robust Software Engineering (RSE) group at the NASA Ames Research Center.

Though JPF is an efficient model checking tool, nevertheless it cannot be used as a standalone tool to verify CTL properties over a given java byte code

Chapter 2

—

Theory

—

This chapter discusses the concepts Java Path Finder, Listeners as an extension. In addition to it The Model Checking Algorithm for satisfiability of a CTL property has been discussed.

2.1 Principle

JPF provides an implementation that notifies registered observer instances about certain events at the search- and JVM- level. These notifications cover a broad spectrum of JPF operations, from low level events like instruction executed to high level events like search finished. Each notification is parameterized with the corresponding source (either the Search or the JVM instance), which can be then used by the notified listener to obtain more information about the event / JPF's internal state.
Configuration is usually done with the jpf.listener property, either from the command line, or a *.jpf property file.

2.2 Listener

Listeners are the most important extension mechanism of JPF. They provide a way to observe, interact with and extend JPF execution with your own classes. Since listeners are dynamically configured at runtime, they do not require any modification to the JPF core. Listeners are executed at the same level like JPF.

Fig: 1.3Interaction of listener with JPF

2.3 Kripke Structure

A Kripke structure is a type of nondeterministic finite state, used in model checking to represent the behavior of a system. It is basically a graph whose nodes represent the reachable states of the system and whose edges represent state transitions. A labeling function maps each node to a set of properties that hold in the corresponding state. Temporal logics are traditionally interpreted in terms of Kripke structures.

A Kripke structure is a 4-tuple **M = (S, I, R, L)** consisting of

- a finite set of states S
- a set of initial states $I \subseteq S$
- A transition relation $R \subseteq S \times S$ where $\forall s \in S, \exists s' \in S$ such that $(s, s') \in R$
- A labeling (or interpretation function) $L : S \Rightarrow 2^{AP}$

A Kripke Structure can be unwinded to form a Computational Tree

Fig: 1.4 Kripke Structure for a coffee vending Machine

Chapter 3

—

Model Checking Algorithm

—

This chapter deals with the Labeling Algorithm for the satisfiability of a CTL property.

3.1 Linear Time Temporal Logic

Linear-time temporal logic, or LTL for short, is a temporal logic, with connectives that allow us to refer to the future. It models time as a sequence of states, extending infinitely into the future. This sequence of states is sometimes called a computation path, or simply a path. In general, the future is not determined, so we consider several paths, representing different possible futures, any one of which might be the 'actual' path that is realized.

3.1.1 Syntax of LTL

Φ: = |_|| p | (φ) | (φ φ) | (φ φ) | (φ → φ) | (X φ)|(F φ) | (G φ) |(φ U φ) | (φ W φ) |(φ R φ)

where p is any propositional atom from some set Atoms. Thus, the symbols _ and are LTL formulas, as are all atoms from Atoms; and φ is an LTL formula if φ is one, etc. The connectives X, F, G, U, R, and W are called temporal connectives. X means 'neXt state,' F means 'some Future state,' and G means 'all future states (Globally).' The next three, U,

R and W are called 'Until,' 'Release' and 'Weak-until' respectively.

3.2 Branching Time Logic

In our analysis of LTL we noted that LTL formulas are evaluated on paths. We defined that a state of a system satisfies an LTL formula if all paths from the given state satisfy it. Thus, LTL implicitly quantifies universally over paths. Therefore, properties which assert the existence of a path cannot be expressed in LTL. This problem can partly be alleviated by considering the negation of the property in question, and interpreting the result accordingly .To check whether there exists a path from s satisfying the LTL formula φ, we check whether all paths satisfy φ; a positive answer to this is a negative answer to our original question, and vice versa. We used this approach when analyzing the ferryman puzzle in the previous section. However, as already noted, properties which mix universal and existential path quantifiers cannot in general be model checked using this approach, because the complement formula still has a mix. Branching-time logics solve this problem by allowing us to quantify explicitly over paths. We will examine a logic known as Computation Tree Logic, or CTL. In CTL, as well as the temporal operators U, F, G and X of LTL we also have quantifiers A and E which express ‘all paths’ and ‘exists a path’, respectively.

3.2.1 Syntax of CTL

Computation Tree Logic, or CTL for short, is branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the ‘actual’ path that is realized.

As before, we work with a fixed set of atomic formulas/descriptions (such as p, q, r, or p1, p2,).We define CTL formulas inductively via a Backus Naur form as done for LTL.

$\varphi ::= \mid \_ \mid p \mid (\ \varphi) \mid (\varphi\ \varphi) \mid (\varphi\ \varphi) \mid (\varphi \to \varphi) \mid AX\varphi \mid EX\varphi \mid$
$AF\varphi \mid EF\varphi \mid AG\varphi \mid EG\varphi \mid A[\varphi\ U\ \varphi] \mid E[\varphi\ U\ \varphi]$

Where p ranges over a set of atomic formulas.

3.3 The Labeling Algorithm

We present an algorithm which, given a model and a CTL formula, outputs the set of states of the model that satisfy the formula. The algorithm does not need to be able to handle every CTL connective explicitly, since we have already seen that the connectives, and form an adequate set as far as the propositional connectives are concerned; and AF, EU and EX form an adequate set of temporal connectives. Given an arbitrary CTL formula φ, we would simply pre-process φ in order to write it in an equivalent form in terms of the adequate set of connectives, and then call the model-checking algorithm. Here is the algorithm:

INPUT : a CTL model $M = (S, \to, L)$ and a CTL formula φ.
OUTPUT: the set of states of M which satisfy φ.

First, change∧ φ to the⊥ output of TRANSLATE (φ), i.e., we write φ in terms of the connectives AF, EU, EX, ¬ and using the equivalences given earlier in the chapter. Next, label the states of M with the sub formulas of φ that are satisfied there, starting with the smallest sub formulas and working outwards towards φ.

Suppose ψ is a sub formula of φ and states satisfying all the immediate sub formulas of ψ have already been labeled. We determine by a case analysis which states to label with ψ.
If ψ is

_: then no states are labeled with .
_ P: then label s with p if p L(s).

_ ψ1 ψ2: label s with ψ1 ψ2 if s is already labeled both with ψ1 and with ψ2. _ ψ1: label s with ψ1 if s is not already labeled with ψ1.
_ AFψ1:
– If any state s is labeled with ψ1, label it with AFψ1.

– Repeat: label any state with AFψ1 if all successor states are labelled with AFψ1, until there is no change.
_ E [ψ1 U ψ2]:
– If any state s is labeled with ψ2, label it with E[ψ1 U ψ2].
– Repeat: label any state with E [ψ1 U ψ2] if it is labeled with ψ1 and at leastone of its successors is labeled with E [ψ1 U ψ2], until there is no change. .
_ EXψ1: label any state with EXψ1 if one of its successors is labeled with ψ1.

Having performed the labeling for all the sub formulas of φ (including φ itself), we output the states which are labeled φ. The complexity of this algorithm is O (f V (V + E)), where f is the number of connectives in the formula, V is the number of states and E is the number of transitions; the algorithm is linear in the size of the formula and quadratic in the size of the model.

3.3.1 The Pseudo Code for the algorithm

We present the pseudo-code for the basic labeling algorithm. The main function SAT (for ‘satisfies’) takes as input a CTL formula. The program SAT expects a parse tree of some CTL formula constructed by means of the grammar in Definition 3.12. This expectation reflects an important precondition on the correctness of the algorithm SAT. For example, the program simply would not know what to do with an input of the form X (_ EFp3), since this is not a CTL formula.

```
function SAT (φ)

/* determines the set of states
satisfying φ */ begin
case
φ is _ : return S
φ is   : return
φ is atomic: return {s   S | φ   L(s)}
φ is  φ1 : return S − SAT (φ1)
φ is φ1   φ2 : return SAT (φ1) ∩ SAT (φ2)
φ is φ1   φ2 : return SAT (φ1)   SAT (φ2)
φ is φ1 → φ2 : return SAT (  φ1   φ2)
φ is AXφ1 : return SAT (  EX   φ1)
φ is EXφ1 : return SATEX(φ1)
φ is A[φ1 U φ2] : return SAT(  (E[   φ2 U (  φ1    φ2)]   EG   φ2))
φ is E[φ1 U φ2] : return SATEU(φ1, φ2)
φ is EFφ1 : return SAT (E(_ U φ1))
φ is EGφ1 : return SAT(  AF   φ1)
φ is AFφ1 : return SATAF (φ1)
φ is AGφ1: return SAT (EFφ1)
end
case
end
function
```

The algorithm and its sub functions presented above use program variables X, Y , V and W which are sets of states. The program for SAT handles the easy cases directly and passes more complicated cases on to special procedures, which in turn might call SAT recursively on sub expressions. These special procedures rely on implementations of the functions

Pre :

(Y) = {s S | exists s_, (s → s_ and s_ Y)}
pre (Y) = {s S | for all s_, (s → s_ implies
s_ Y)}.

'Pre' denotes travelling backwards along the transition relation. Both functions compute a pre-image of a set of states. The function pre

(Instrumental in SATEX and SATEU) takes a subset Y of states and returns the set of states which can make a transition into Y. The function pre , used in SATAF, takes function SATEX (φ)

```
/* determines the set of states satisfying
EXφ */ local var X, Y
begin
X: = SAT (φ);
Y: = pre(X); return Y
End
```

The function SATEX. It computes the states satisfying φ by calling SAT. Then, it looks backwards along to find the states satisfying EX φ.

Function SATAF (φ)

```
/* determines the set of states satisfying
AFφ */

local var X, Y
Begin X: = S;
Y:    =    SAT
(φ);     repeat
until  X  =  Y
begins
X: = Y;
Y: = Y     pre
(Y)
End
retu
rn Y
end
```

A set Y and returns the set of states which makes transitions only into Y. Observer that pre can be expressed in terms of completion and pre as follows.

Pre
  (Y) = S – pre(S-Y) 3.8
Where we write S − Y for the set of all s, S which are not in Y.

Chapter 4

—

Correction of Algorithm

—

This chapter deals with illustrating the labeling algorithm for model checking just presented above.

4.1 The correctness of SATEG

We saw at the end of the last section that $[[EG\ \varphi]] = [[\varphi]] \cap pre\ ([[EG\ \varphi]])$
This implies that $EG\varphi$ is a fixed point of the function $F(X) = [[\varphi]] \cap pre\ (X)$

In fact, F is monotone, $EG\varphi$ is its greatest fixed point and therefore $EG\varphi$ can be computed using the theorem stated below

Theorem:

Let F be as defined above and let S have n + 1 elements.
Then F is monotone, $[[EG\ \varphi]]$ is the greatest fixed point of F, and $[[EG\varphi]] = FN+1(S)$.

Let F be as defined above and let S have n + 1 elements. Then F is monotone, $[[EG\ \varphi]]$ is the greatest fixed point of F, and $[[EG\varphi]] = FN+1(S)$.

PROOF:

1. In order to show that F is a monotone, we take any two subsets X and Y of S such that X Y and we need to show that F(X) is a subset of F(Y). Given s0 such that there is some s1 X with $s0 \rightarrow s1$, we certainly have $s0 \rightarrow s1$, where s1 Y, for X is a subset of Y. Thus we showed pre

(X) pre
(Y) from which
We readily conclude that $F(X) = [[\varphi]] \cap pre\ (X)\ [[\varphi]] \cap pre\ (Y) = F(Y)$.

2. We have already seen that $[[EG\ \varphi]]$ is a fixed point of F. To show that it is the greatest fixed point, it suffices to show here that any set X with F(X) = X has to be contained in [[EG φ]]. So let s0 be an element of such a fixed point X. We need to show that s0 is in [[EG φ]] as well. For that we use the fact that

s0 X = F(X) = [[φ]] ∩ pre (X)

to infer that s0 [[φ]] and s0 → s1 for some s1 X; but, since s1 is in X, we may apply that same argument to s1 X = F(X) = [[φ]] ∩ pre (X) and we get s1 [[φ]] and s1 → s2 for some s2 X. By mathematical induction, we can therefore construct an infinite path s0 → s1 → →sn → sn+1 → . . . such that si [[φ]] for all i ≥ 0.
By the definition of [[EG φ]], this entails s0 [[EG φ]].

3. The last item is now immediately accessible from the previous one and theorem now we can see that the procedure SATEG is correctly coded and terminates. First, note that the line Y: = Y ∩ pre (Y) in the procedure SATEG(Y) without changing the effect of the procedure. To see this, note that the first time round the loop, Y is SAT (φ); and in subsequent loops, Y SAT (φ), so it doesn't matter whether we intersect with Y or SAT (φ) 2.

With the change, it is clear that SATEG is calculating the greatest fixed point of F; therefore its correctness follows from Theorem 3.25

4.2 The correctness of SATEU

Proving the correctness of SATEU is similar. We start by noting the equivalence
E [φ U ψ] ≡ ψ (φ EXE [φ U ψ]) and we write it as [[E [φ U ψ]]] = [[ψ]] ([[φ]] ∩ pre
[[E[φ U ψ]]]).
That tells us that [[E[φ U ψ]]] is a fixed point of the
function G(X) = [[ψ]] ([[φ]] ∩ pre(X)).
As before, we can prove that this function is monotone. It turns out that [[E [φ U ψ]]] is its least fixed point and that the function SATEU is actually computing it in the manner of.

Theorem:
Let G be defined as above and let S have n + 1 elements. Then G is monotone, [[E (φ U ψ)]] is the least fixed point of G, and we have
[[E (φ U ψ)]] = Gn+1( ).

2. If you are skeptical, try computing the values Y0, Y1, Y2, where Yi represents the value of Y after i iterations round the loop. The program before the change computes as follows:
Y0 = SAT (φ)
Y1 = Y0 ∩ pre (Y0)

Y2 = Y1 ∩ pre (Y1)
= Y0 ∩ pre (Y0) ∩ pre (Y0 ∩ pre (Y0))
= Y0 ∩ pre (Y0 ∩ pre (Y0)).
The last of these equalities follows from the monotonicity of pre .
Y3 = Y2 ∩ pre (Y2)
= Y0 ∩ pre (Y0 ∩ pre (Y0)) ∩ pre (Y0 ∩ pre (Y0 ∩ pre (Y0)))
= Y0 ∩ pre (Y0 ∩ pre (Y0 ∩ pre (Y0))).

Again the last one follows by monotonicity. Now look at what the program does after the change:
Y0 = SAT (φ)
Y1 = SAT (φ) ∩ pre (Y0) =Y0 ∩ pre (Y0)
Y2 = Y0 ∩ pre (Y1)
Y3 = Y0 ∩ pre (Y1) Y0 ∩ pre (Y0 ∩ pre (Y0)).

A formal proof would follow by
induction on i.

PROOF:

1. Again, we need to show that X Y implies G(X) G(Y), but that is essentially the same argument as for F, since the function which sends X to pre
(X) is monotone and all that G now does is to perform the intersection and union of that set with constant sets [[φ]] and [[ψ]].

2. If S has n+1 elements, then the least fixes point of G equals Gn+1() by theorem. Therefore if suffices to show that this set equals [[E (φ U ψ)]].

Simple observe what kind of states we obtain by iterating G on the empty set.

G1 ( ) = [[ψ]] (φ]] ∩ pre ([[ ]])) = [[ψ]] ([[φ]] ∩) = [ψ]] = [[ψ]] which are all the states s0 [[]E(φ U ψ)] where we choose i=0 according to the definition of Untill.

Now, G2 ( ) = [[ψ]]([[φ]]) ∩ pre (G1 ( ))) tells us that the elements of G2 ( ) are all those s0 E(φ U ψ)]] where we chose i ≤ 1. By mathematical induction , we see that Gk+1 ( ) is the set of all states s0 for which we choose i≤ k to secure s0 [[E(φ U ψ)]]. Since this holds for all k, we see that [[E (φ U ψ)]] is nothing but the union of all sets
Gk+1( ) with k ≥ 0; but, since Gn+1( ) is a fixed point of G, we see that this union is just Gn+1( ).

The correctness of the coding of SATEU follows similarly to that of SATEG. We change the line Y: = Y (W ∩ pre (Y)) into Y: = SAT (ψ) (W ∩ pre(Y)) and observe that this does not change the result of the procedure, because the first time round the loop, Y is SAT(ψ); and, since Y is always increasing, it makes no difference whether we perform a union with Y or with SAT(ψ). Having made that change, it is then clear that

SATEU is just computing the least fixed point of G using Theorem 3.24.

We illustrate these results about the functions F and G above through an example. Consider the system in Figure 3.38. We begin by computing the set [[EF p]]. By the definition of EF this is just

[[E (_ U p)]]. So we
have φ1 def = _ and
φ2

def = p. From Figure 3.38, we obtain [[p]] = {s3} and of course [[_]] =S. Thus, the function G above equals G (X) = {s3} pre (X). Since [[E(_Up)]] equals the least fixed
point of G, we need to iterate G on until this process stabilizes.
First, G1 ( ) = {s3} pre ( ) = {s3}.
Second, G2 ( ) = G (G1 ( )) = {s3} pre ({s3}) = {s1, s3}.
Third, G3 ( ) = G (G2 ( )) = {s3} pre ({s1, s3}) = {s0, s1, s2, s3}.
Fourth, G4 ( ) = G (G3 ( )) = {s3} pre ({s0, s1, s2, s3}) = {s0, s1, s2, s3}.
Therefore,{s0, s1, s2, s3} is the least fixed point of G, which equals [[E(_ U p)]] by . But then [[E(_ U p)]] = [[EF p]]. The other example we study is the computation of the set [[EG q]]. By Theorem, that set is the greatest fixed point of the function F above,
where φ def = q. From Figure 3.38 we see that [[q]] = {s0, s4} and so F(X) = [[q]] ∩ pre (X) = {s0, s4} ∩ pre (X). Since [[EG q]] equals the greatest fixed point of F, we need to iterate F on S until this process stabilizes.
First, F1(S) = {s0, s4} ∩ pre (S) = {s0, s4} ∩ S since every s has some s_ with s → s_. Thus, F1(S) = {s0, s4}. Second, F2(S) = F(F1(S)) = {s0, s4} ∩ pre ({s0, s4}) = {s0, s4}. Therefore, {s0, s4} is the greatest fixed point of F, which equals [[EG q]] by Theorem.

Chapter 5

Conclusion

In this report we have studied about the various approaches of Model checking and all the theory related to it. We also came to know about Java Path Finder by which we performed Model checking. The unique thing which we have noticed about the project is that we took Java byte Code as input to perform Model checking instead of the Model which the other Model checkers use. Thus, the project can be used to easily check properties of given software just by using its code.

Chapter 6

References

- Logic in Computer Science by Michael Huth and Mark Ryan
- Modeling Systems by Edmund Clarke
- http://babelfish.arc.nasa.gov/trac/jpf
- http://babelfish.arc.nasa.gov/hg/jpf/
- http://javapathfinder.sourceforge.net/
- http://en.wikipedia.org/wiki/Java_Pathfinder