ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Nguyễn Đình Phúc

VERIFYING LINEAR TEMPORAL LOGIC SPECIFICATION FOR FINITE JAVA PROGRAMS

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành: Công Nghệ Thông Tin

HÀ NỘI - 2011

ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Nguyễn Đình Phúc

VERIFYING LINEAR TEMPORAL LOGIC SPECIFICATION FOR FINITE JAVA PROGRAMS

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành: Công Nghệ Thông Tin

Cán bộ hướng dẫn: TS. Trương Anh Hoàng

Cán bộ đồng hướng dẫn: ThS. Phạm Thị Kim Dung

HÀ NỘI - 2011

ACKNOWLEDGEMENTS

Most importantly, I would like to thank PhD. Truong Anh Hoang and MSc. Pham Thi Kim Dung. Your support makes everything I do in the study possible, and your help ensured that this thesis was the best it could be.

I also want to say thank you all of my teachers in the University of Engineering and Technology. Through the years, you taught me necessary knowledge for study and research with all of your very best.

A big thank-you goes to Google for the sponsorship in the Google Summer of Code 2010 program, and to PhD. Franco Raimondi for your mentoring experience. Your guidance helped me find out the right way in the study process, for that I prepared the necessary background knowledge of the project. Without those supports, it would never have happened.

I also thank Ewgenij Starostin, and Luu Chung Tuyen for discussing with me and collaborating during the GSoC program. It has been a really nice experience working with you.

Special thanks go out to my family, my classmates and all of my friends that helped me so much, made me believe that I can do it successfully. Personally, your warmly friendship is the most important thing in my life.

Once again, thank all of you so much!

Hanoi, May, 20th, 2011

Nguyen Dinh Phuc

ABSTRACT

This thesis represents an approach to combining symbolic execution with program monitoring for the verification of finite Java programs against Linear Temporal Logic (LTL) specifications. LTL has been widely used for expressing temporal properties of programs viewed as transition systems. Many formal methods require significant manual effort and not scalable for real sized system; typical model checking environments use Buchi automata which are not designed to deal with finite execution traces. Hence, the approach presented here consists of modifying the standard LTL to Buchi automata conversion technique to generate finite-automata, which are used to check finite program traces. Besides, the verification can combine with symbolic execution to allow automatically detect counter-examples in all feasible paths of the program. The approach has been then implemented in a tool, which is an extension of the Java Pathfinder framework for runtime analysis of Java programs.

LỜI CAM ĐOAN

Tôi xin cam đoan khoá luận "verifying linear temporal logic specification for finite Java programs" là công trình nghiên cứu của riêng tôi. Các số liệu, kết quả được trình bày trong luận án là hoàn toàn trung thực. Tôi đã trích dẫn đầy đủ các tài liệu tham khảo, công trình nghiên cứu liên quan đến công trình này.

Khóa luận được hoàn thành trong thời gian tôi làm Sinh viên tại Bộ môn Công nghệ phần mềm, Khoa Công nghệ Thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội.

Hà Nội, ngày 20 tháng 05 năm 2011

Sinh viên

Nguyễn Đình Phúc

CONTENTS

LIST OF ABBREVIATIONS

Atom	Atomic proposition
FSA	Finite state automata
JPF	Java Pathfinder
LT	Linear temporal
LTL	Linear temporal logic
PC	Path condition

LIST OF FIGURES

Chapter 1. Introduction

1.1. Motivation

The conformance of computer programs against their specification has concerned the industry and has been studied extensively in academia for many years. Especially, it took scientists a lot of time and effort on the development of semantic systems for formalizing logics that can formally justify that a computer program conforms to a formula in some logics. On the other hands, there are many studies tried to find ways to apply the formal logic in the analysis of computer programs and their design. In the last decade, various works have explored model checking as a promising technique for mechanizing proofs of correctness. A typical example in this direction is the model checker SPIN [1], another one is the SLAM toolkit [2] which combines techniques from program analysis, model checking and automated deduction to verify if a program satisfies a given safety property. The tool jStar[3] for Java code employs theorem proving and abstract interpretation based on separation logic to verify interaction protocols among a number of objects. However, such *heavy-weight* formal proofs techniques are usually not scalable to real sized system without significant manual effort to abstract the system to a model which is then analyzed.

Hence, a more *light-weight* utilization of formal techniques will be very useful and more practically feasible in the shorter term. The light-weight formal method is here defined as an automatic, regardless of the size of the program under verification. This recent research direction gained plenty of great interest [4, 5]. A sample of such light-weight techniques is usually referred to as program monitoring in which the idea is to monitor the execution of a program against a formal specification written in some formal specification logic. In that way, the technique is practically attainable because only one trace is examined, and it is also advantageous since we can state more complicated properties than normally applicable in typical testing environments.

The work in this thesis is the result of trying to develop such a method for monitoring running programs against Linear Temporal Logic (LTL) requirement specification. This temporal logic has been used as the logic for representing high-level linear-time properties in many model checkers such as SPIN [1]. The negation of such a LTL requirement is then automatically translated into a Buchi automaton that accepts all infinite words that violate this specification (Buchi automata are finite automata on infinite words). The obtained synchronous interleaving product of the model/program with the automaton will be checked whether or not an accepting cycle

exists. The obtained global system behavior is itself again represented by a Buchi automaton, and often referred as the *state space* of the system. If the language accepted by this product automaton is empty, it contains exactly those behaviors that satisfy the original claim. Otherwise, this means that the given system does not satisfy the original temporal LTL formula. This interleaving product also can be represented as a graph which is commonly also referred as the global *reachability graph*. Obviously, this product automaton needs to be stored, so this may lead to the state-explosion problem [6].

Buchi automata are designed to operate on infinite execution traces. However, the distinction between finite and infinite traces is necessary because the acceptance automata are different (e.g., the Buchi automaton for the formula G(F(a) Λ F(b)) is different from the finite automaton), and adding self-loops to final states to make all execution infinite may modify the intended semantics of Java code (e.g., some Java procedures may be part of a bigger system and be required to terminate). Therefore, there is a raised question is that whether Buchi automaton can also be used to efficiently monitor finite traces of computer programs. Even if we handle this problem by repeating the last state indefinitely, the model checker would still require storing the combined state space of the Buchi automaton and particular program trace, in order to check for accepting cycles.

In addition, the symbolic execution – a case of abstract interpretation – has recently been raised as a useful software testing technique. It can be seen as the analysis of programs by tracking symbolic rather than actual concrete values. Based on symbolic execution, we can explore all feasible paths in the program, theoretically. Hence, it can detect more hidden bugs than typical testing techniques that require tester to design test case and test driver manually.

1.2. Purpose and scope

In this thesis, an effort to provide a more efficient alternative combining program monitoring with symbolic execution is presented for building a tool to verifying Java programs against its LTL specification. The work presented here includes defining an accurate LTL grammar in order to make a parser over LTL formulae, and an algorithm [7], based on standard LTL to Buchi automata construction technique [9], that produces typical finite-state automata which can be used to monitor the execution of Java programs against LTL formulae on finite traces. The monitor tool has been

developed as an extension of Java Pathfinder (JPF)[1], a Virtual Machine for Java bytecode that enables exploration of a program's execution and supports various verification activities. Lastly, the tool can also be applied to verify symbolic executions [8].

1.3. Thesis structure

The remainder of this thesis consists of four sections and is organized as follows. Section 2 discusses background knowledge that related to the development process presented in this thesis, and introduces Java Pathfinder – the framework that our tool will become an extension of it. Section 3 describes in detail the implementation part of this work; it will also provide the pseudocode describing how the proposed algorithm was implemented in the tool. Section 4 illustrates how the developed tool works via few experimental examples. Lastly, Section 5 closes the thesis with discussion and conclusion.

[1] http://babelfish.arc.nasa.gov/trac/jpf

Chapter 2: Background

2.1. Transition systems

In this section, we will consider state-changing systems and discuss a formal technique to represent it – *transition systems.* State-changing systems are the systems that change over time from one state to others under the influence of various actions. Transition systems are a particular formalization of state-changing systems in which a system is described by a set of variables and a state comprises of an assignment of values to these variables.

The normal proposition logic can be used to describe a static system. By "static" we mean that the system does not change over time, or that we are only care about one specific state of the system. There are many applications where we are interested in systems whose state varies in time. Examples of such changing-systems include operating systems, networks, scheduler, autonomous devices, and many others.

For instance, when we reason about the safety part of cryptographic protocols, we must consider all probable sequences of actions undertaken by an intruder. Each action changes the state of the system that uses the protocol. When we reason about functioning of hardware, we also talk about probable sequence of state changes in hardware, for example, the sequences of state through which a processor goes when executing a sequence of instructions.

A *state-changing system* is usually featured by the following properties.

- At each particular time moment, the system is in a specific *state*.
- The state of the system may change, typically under the influence of some kind of *action*. There are many kinds of action. Actions can occur internally within the system but can also be out of its control, for instance, performed by the system environment.

From these properties, we can build a mathematical model of such systems which are usually based on the following abstractions of the notions of the state and action:

- We introduce *variables* to characterize some attributes of the system and assume that the state is identified by assigning the values to the variables at a given state. These variables are also known as the *state variables* of this system.
- When we formalize an *action*, we try to determine whether or not the current state of the system changes induced by this action, and how the state variables change after this action.

However, when talking about transition system, the word *transition* is usually used instead of action, hence the name "transition systems". The following samples of state-changing systems typically arise in numerous applications.

- *Reactive systems.* These are the systems that "react" to their environment on a continuous basis, responding appropriately to each action.
- *Concurrent systems.* These are the systems that consist of a set of components functioning together. Usually, each component runs independently in parallel, but they also communicate through shared variables or some sort of *communication channels*.

Based on the above discussion, we can now define *transition systems as* a formalization of state-changing systems.

Definition 1. Transition System

A transition system is a tuple $S = (X, D, dom, In, T)$,

Where

- X is a finite set of *state variables*
- D is a non-empty set, called *domain*. Elements of D are called values.
- *Dom* is a mapping from X to the set of non-empty subsets of D. For each state variable $x \epsilon X$, the set $dom(x)$ is called the *domain for x.*

The meaning of *In* and *T* will be described below after the notions of state and transition have been defined.

Now we define a state of a transition system S as a function $s : X \rightarrow D$ such that for every $x \epsilon X$ there is an $s(x) \epsilon dom(x)$. A *transition* is a set of pairs of states. About *In* and *T*, we have the following definitions.

- *In* is a set of begin states, called the *initial states* of *S.*
- *T* is a finite set of transitions.

A transition *t* is said to be *applicable* to a state *s* if there exists a state *s'* such that (s; s') ϵ *t.* A transition *t* is said to be *deterministic* if for each state *s* there exists at most one state *s'* such that *(s; s') ϵ t*, otherwise, the transition is said to be *non-deterministic*.

The *transition relation* of *S*, denoted by Tr_S, is the set of all transitions in the system, i.e., it is the union of all pairs of state $\cup_{t \epsilon T}$ t.

A transition system S is considered to be finite-state if *X* is finite and infinite-state otherwise.

Let *t* be a transition. Obviously, t is non-deterministic if and only if there exist states *s, s_1', s_2'* such that $s_1' \neq s_2'$, $(s, s_1') \in t$ and $(s, s_2') \in t$.

Consider a transition system *S*. It is easy to see that S is a finite-state system if and only if for every state variable *x* of *S* the domain for this variable is finite. Note that according to our definition, the states are only those mappings *s* which map *x* into a value in *dom(x)*. Therefore, if we replace *D* by $U_{x \in X}$ *dom(x)*, i.e., use only those values that belong to a domain at least one variable, then the set of states of the system does not change. Therefore, we can assume that $D = U_{x \in X}$ *dom(x)*. Under this assumption, finite-state systems are exactly those whose domain D is finite. In this thesis we will only care about finite-state systems.

2.2. Temporal properties

2.2.1. Traces

Executions are different sequences comprising of states and actions. Actions are mostly used to model the interaction of the transition systems, via synchronous or asynchronous communication channel. In the end, action is not our prime interest, but instead we are intersted in the visited states during executions. Obviously, the sates themselves are not "observable", but just their atomic propositions. Thus, rather than having an execution of the form $s_0 \rightarrow s_1 \rightarrow s_2 \ldots$ we consider sequences of form $L(s_0)$ $L(s_1)$ $L(s_2) \ldots$ that register the (set of) atomic propositions that are valid along the execution. Such sequences are called traces.

The traces of a transition system are thus words over the alphabet 2^{AP}. In common model checking environments, all traces are considered as infinite words (without terminal states). This assumption is made just for simplicity and does not impose any serious restriction. First of all, before checking any (linear-time) property, a reachability analysis could be done to determine the set of terminal states. If indeed some terminal state is encountered, the system contains a deadlock and has to be repaired before any further analysis. Alternatively, each transition system TS (that probably has t terminal state) can be extended such that for each terminal state s in TS there is a new state s_{stop}, transition $s \rightarrow s_{stop}$, and s_{stop} is equipped with a self-loop, i.e., $s_{stop} \rightarrow s_{stop}$. The resulting "equivalent transition system obviously has no terminal states. However, as discussed formerly, some systems are required to be terminated (maybe they are a part of a larger state), so a further alternative is to adapt the linear-time framework for transition system with terminal states. In this circumstance, traces may be finite words that are targets to be handled by the tool presented in this work.

2.2.2. Linear temporal properties

Linear temporal properties specify the traces that a transition system should exhibit. Informally, a linear-time property specifies the desired behavior of the system under verification. The following section provides a formal definition of such properties. This definition is rather elementary, and gives a good basic understanding of what a linear temporal property is. In section 2.3, a logical formalism will be introduced that allows reprsenting linear temporal properties by using Linear Temporal Logic.

In the following, we assume a fixed set of atomic propositions AP. A linear temporal (LT) property is a requirement on the traces of a transition system. Such property can be understood as a requirement over all words over AP, and is defined as the set of words (over AP) that are admissible:

Definition 1. LT Property

A linear temporal property over the set of atomic propositions AP is a subset of $(2^{AP})^{w}$.

Here, $(2^{AP})^{w}$ denotes the set of words that arise from the infinite concatenation of words in 2^{AP}. An LT property is thus a language (set) of infinite words over the alphabet 2^{AP}. The fulfillment of an LT property by a transition system is defined as follows.

Let P be an LT property over AP and TS = (S, Act, →, I, AP, L) a transition system. Then, TS = (S, Act, →, AP, L) satisfies P, denoted TS |= P, iff Traces(TS) ⊆ P. State s ∈ S satisfies P, notation s |= P, whenever Traces(s) ⊆ P.

Thus, a transition system satisfies the LT property P if all its traces respect P, i.e., if all its behaviors are admissible. A state satisfies P whenever all traces starting in this state fulfill P. Let's consider an example taken from [11] to see how LT properties are applied to the real world.

Example Traffic Lights

Consider two simplified traffic lights that only have two possible settings: red and green. Let the propositions of interest be

AP = {red_1, $green_1$, red_2, $green_2$}

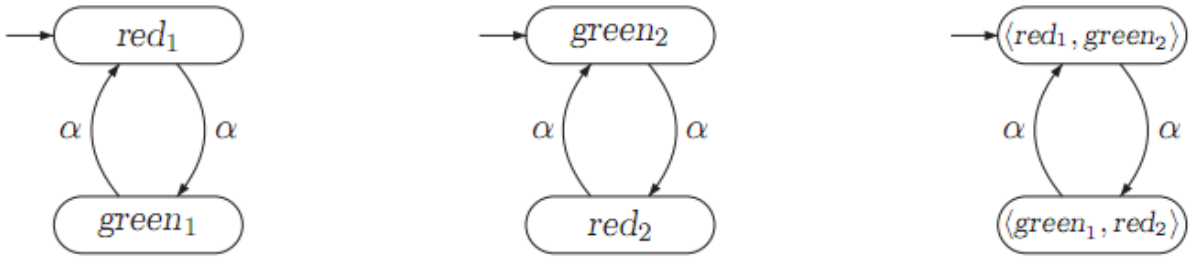

Figure 1: Two fully synchronized traffic lights (left and middle) and their parallel composition (right)

We consider two LT properties of these traffic lights and give some example words that are contained by such properties. First, consider the property P that states:

"The first traffic light is infinitely often green."

This LT property corresponds to the set of infinite words of the form $A_0\ A_1A_2\ldots$ over 2^{AP}, such that $green_1 \in A_i$ holds for infinitely many *i*. For example, P contains the infinitely words

{ red_1, $green_2$ } { $green_1$, red_2 } { red_1, $green_2$ } { $green_1$, red_2 } …,

∅ { $green_1$ } ∅ { $green_1$ } ∅ { $green_1$ } ∅ { $green_1$ } ∅ …

{ red_1, $green_1$ } { red1, $green_1$ }{ red_1, $green_1$ } { red1, $green_1$ }… and

{ $green_1$, $green_2$ } { $green_1$, $green_2$ } { $green_1$, $green_2$ } { $green_1$, $green_2$ } …

The infinite word {red_1, $green_1$ } { red_1, $green_1$ } ∅ ∅ ∅ ∅ … is not in P as it contains only infinitely many occurrences of $green_1$.

As a second LT property, consider P':

"The traffic lights are never both green simultaneously".

This property is formalized by the set of infinitely words of the form $A_0\ A_1\ A_2\ldots$ such that either $green_1 \notin A_i$ or $green_2 \notin A_i$, for all $i \geq 0$. For example, the following infinite words are in P':

{ red_1, $green_2$ } { $green_1$, red_2 } { red_1, $green_2$ } { $green_1$, red_2 } … ,

∅ { $green_1$ } ∅ { $green_1$ } ∅ { $green_1$ } ∅ { $green_1$ } ∅ … and

{ red_1, $green_1$ } { red1, $green_1$ }{ red_1, $green_1$ } { red1, $green_1$ }…,

whereas the infinite word { red_1, $green_2$ } { $green_1$, $green_2$ },… is not in P'.

The traffic lights depicted in Figure 1 are at intersecting roads and their switching is synchronized, i.e., if one switches from red to green, the other switches from green to red. In this way, the lights always have complementary colors. Obviously, these traffic lights satisfy both P and P'. Traffic lights that switch completely autonomously will neither satisfy P – there is no guarantee that the first traffic light is green infinitely often – nor P'.

Often, an LT property does not refer to all atomic propositions occurring in a transition system, but just to a relatively small subset thereof. For a property P over a set of propositions AP' ⊆AP, only the labels in AP' are relevant. Let β be a finite path fragment of TS. We write $trace_{AP'}(\beta)$ to denote the finite trace of β where only the atomic propositions in AP' are considered. Accordingly, $trace_{AP'}(\pi)$ denotes the trace of an infinite path fragment π by focusing on propositions in AP'. Thus, for $\pi = s_0\ s_1\ s_2\ldots$, we have

$$trace_{AP'} = L'(s_0)\ L'(s_1)\ldots = (L(s_0) \cap AP')\ (L(s_1) \cap AP')\ldots$$

Let $Traces_{AP'}(TS)$ denote the set of traces $trace_{AP'}(Paths(TS))$. Whenever the set AP'of atomic propositions is clear from the context the subscript AP' is omitted. In the rest of this thesis, the restriction to a relevant subset of atomic propositions is often implicitly made.

2.2.2.1 Safety properties and Invariants

Safety properties are often understood as "nothing bad should happen". The mutual exclusion property – always at most one process is in its critical section – is a typical safety property. It states that the bad thing (having two or more processes in their critical section simultaneously) never occurs. Another typical safety property is deadlock freedom. For example, the popular dining philosophers problem, such deadlock could be characterized as the situation in which all philosophers are waiting to pick up the second chopstick. This bad (i.e., unwanted) situation should never occur.

In fact, the above safety properties are of a particular kind: they are *invariants*. Invariants are LT properties that are given by a condition Φ for the states and require that Φ holds for all reachable states.

Definition 2. Invariant

An LT property P_{inv} over AP is an *invariant* if there is a propositional logic formula Φ over AP such that

$$P_{inv} = \{\ A_0A_1A_2\ldots \in (2^{AP})^{\omega} \mid \forall j \geq 0.\ A_j \models \Phi\ \}.$$

Φ is called an invariant condition (or state condition) of P_{inv}.

Note that

TS $\models P_{inv}$ iff trace(π) $\in P_{inv}$ for all paths π in TS

iff L(s) $\models \Phi$ for all states s that belong to a path of TS

iff L(s) $\models \Phi$ for all states s $\in$ Reach(TS).

Thus, the notion "invariant" can be explained as follows: the condition Φ has to be fulfilled by all initial states and satisfaction of Φ is invariant under all transitions in the reachable fragment of the given transition system. The latter means that if Φ holds for the source state s of a transition $s \rightarrow s'$, then Φ holds for the target state s' too.

Consider the examples of mutual exclusion and deadlock freedom for the dining philosophers. The mutual exclusion property can be described by an invariant using the proposition logic formula

$$\Phi = \neg crit_1 \vee \neg crit_2.$$

For deadlock freedom of the dining philosophers, the invariant ensures that at least one of the philosophers is not waiting to pick up the chopstick. This can be established using the propositional formula:

$$\Phi = \neg wait_0 \vee \neg wait_1 \vee \neg wait_2 \vee \neg wait_3 \vee \neg wait_4$$

Here, the proposition $wait_i$ characterizes the state(s) of philosopher *i* in which he is waiting for a chopstick.

On the other hands, invariants can be seen as state properties. Some safety properties, however, may impose requirements on finite path fragments, and cannot be verified by considering the reachable states only. To see this, consider the example of a cash dispenser, also known as an automated teller machine (ATM). A natural requirement is that money can only be withdrawn from the dispenser once a correct personal identifier (PIN) has been provided. This property is not an invariant, since it is not a state property. It is, however, considered to be a safety property, as any infinite run violating the requirement has a finite prefix that is "bad", i.e., in which money is withdrawn without issuing a PIN before.

Formally, safety property P is defined as an LT property over AP such that any infinite word σ where P does not hold contains a *bad prefix.* The later means a finite prefix σ where the bad thing has happened, and thus no infinite word that starts with this prefix σ fulfills P.

2.2.2.2. Liveness Properties

Informally, safety properties specify that "something bad never happens". For the mutual exclusion algorithm, the "bad" thing is that more than one process is in its critical section, while for the traffic light the "bad" situation is whenever a red light phase is not preceded by a yellow light phase. An algorithm can easily fulfill a safety property by simply doing nothing as this will never lead to a "bad" situation. As this is usually undesired, safety properties are complemented by properties that require some progress. Such properties are called "liveness" properties (or sometimes "progress" properties). Intuitively, they state that "something good" will happen in the future. Whereas safety properties are violated in finite time, i.e., by a finite system run, liveness properties are violated in infinite time, i.e., by infinite system runs.

A typical example for a liveness property is the requirement that certain events occur infinitely often. In this sense, the "good event" of a liveness property is a condition on the infinite behaviors, while the "bad event" for a safety property occurs in a finite amount of time, if it occurs at all. A liveness property (over AP) is defined as an LT property that does not rule out any prefix. This entails that the set of finite traces of a system are of no use at all to decide whether a liveness property holds or not. Intuitively, it means that any finite prefix can be extended such that the resulting infinite trace satisfies the liveness property under consideration. This is contrast to safety properties where it suffices to have one finite trace (the "bad prefix") to conclude that a safety property is refuted.

Definition 3. Liveness Property

LT property P_{live} over AP is a liveness property whenever $pref(P_{live}) = (2^{AP})^*$.

Thus, a liveness property (over AP) is an LT property P such that each finite word can be extended to an infinite word that satisfies P. Stated differently, P is a liveness property if and only if for all finite words $w \in (2^{AP})^*$ there exists an infinite word $\sigma \in (2^{AP})^{\omega}$ satisfying $w\sigma \in P$.

Example: Repeated Eventually and Starvation Freedom

In the context of mutual exclusion algorithms the natural safety property that is required ensures the mutual exclusion property stating that the processes are never simultaneously in their critical sections. (This is even an invariant.) Typical liveness properties that are desired assert that

- (eventually) each process will eventually enter its critical section;
- (repeated eventually) each process will enter its critical section infinitely often;

- (starvation freedom) each waiting process will eventually enter its critical section.

2.2.2.3. Safety vs Liveness Properties

Any LT property is equivalent to an LT property that is a conjunction of a safety and liveness property. Formally, for any LT property P an equivalent LT property P' does exist which is a combination (i.e., intersection) of a safety and a liveness property. All in all, one could say that the identification of safety and liveness properties thus provides an essential characterization of linear temporal properties.

The first result states that safety and liveness properties are indeed almost disjoint. More precisely, it states that the only property that is both a safety and a liveness property is nonrestrictive, i.e., allows all possible behaviors.

Besides, in term of LT properties, there exist Fairness assumptions that serve to rule out traces that are considered to be unrealistic. They consist of unconditional, strong, and weak fairness constraints, i.e., constraints on the actions that occur along infinite executions. Fairness assumptions are often necessary to establish liveness properties, but they are – provided they are realizable – irrelevant for safety properties.

The above definition about Linear Temporal properties except is taken from the [11]. Hence, the reader is reffered to that documentation for a more formally description about those properties.

2.3. Linear Temporal Logic (LTL)

2.3.1. LTL –standard semantics

LTL is a commonly logic used to present temporal properties of software and hardware systems. Given a set of atomic proposition AP, a well-formed LTL formula is defined inductively using the standard Boolean operators, and the temporal operators **X** (next) and **U** (strong until) as follows:

- Each member of AP is a formula,
- If φ and ψ are formulae, then so are $\neg\varphi$, $\varphi \Lambda \psi$, φ V ψ, X φ, φ U ψ.

An interpretation for an LTL formula is an infinite word $w = x_0x_1x_2\ldots$ over 2^{AP}. In other words, an interpretation maps to each instant of time a set of propositions that hold at that instant. We write w_i for the suffix of w starting at x_i. LTL semantics is then defined inductively as follows [10, 6]:

- $w \models p$ iff $p \in x_0$, for $p \in AP$
- $w \models \neg \varphi$ iff not $w \models \varphi$

- $w \models \varphi \, V \, \psi$ iff ($w \models \varphi$) or ($w \models \psi$)
- $w \models \varphi \, \Lambda \, \psi$ iff ($w \models \varphi$) and ($w \models \psi$)
- $w \models \varphi \, U \, \psi$ iff $\exists \, i \geq 0$, such that:

  $w_i \models \psi$ and $\forall \, 0 \, w_j \models \varphi$
- $w \models X\varphi$ iff $w_1 \models \varphi$

Besides, there are two abbreviations "true ≡ φ V ¬φ" and "false ≡ ¬true". Temporal operators F (eventually) and G (always) typically used in LTL formulae are defined in terms of the main operators as follows: Fφ ≡ true U φ and Gφ ≡ ¬F¬ φ. As usual, both propositions and negated propositions are also referred as *literals*.

However, in the context of this thesis, we are only interested in the next-free variant of LTL, namely LTL-X. This is very typical in model checking, because LTL-X is guaranteed to be insensitive to stuttering [11]. This property is important because it avoids the notion of an absolute next state. The next time operator (X) is misleading, because users naturally tend to assume some level of abstraction on the state of a running program. Additionally, it is not straightforward what the desired meaning of a next time formula would be at the last state of a program trace.

In the rest of this thesis, the next-free invariant of LTL is implied whenever LTL is referred. As a result, the next-free LTL is then defined without X temporal operator as follow:

The propositions and Boolean operators are kept the same with the above definition. For temporal operators, we remove the X (next) operator and add the V operator which is defined as the dual of U, i.e.: $\varphi V \psi = !(!\varphi U \, !\psi)$

2.3.2. LTL –finite trace semantics

A trace execuion goes through a sequence of sates; an infinite execution can therefore be viewed as an LTL interpretation, which assigns to each state variables at a particular time moment the set of atomic propositions that are true at the particular program state. Model checking [11] finds accepting cycles in the state graphs of finite-state systems through infinite traces. This require to store the whole state space of the program, even we have on-the-fly technique that build the product automaton as well as checking at runtime. Runtime monitoring does not store the entire state-space of a program. Rather, it only observes finite program executions, on which we also need to interpret LTL formulae. Because of this difference between finite and infinite traces, we need to modify the semantics of the temporal operators to accommodate this difference.

As discussed in formerly in the background section, every LTL formula can contain either a safety part, or an liveness part (or both). The safety/liveness part requires that something bad never happens or something good eventually happens in an execution. We alter the semantic of safety requirement to mean that, *in the portion of the execution that we have observed,* nothing bad happens. Livenesses are similarly required to be satisfied *in the portion of the execution observed*. Otherwise they will have "not yet" been satisfied. Concerning these changes in the semantics, we define the semantics of the temporal operators accordingly.

Let $w=x_0x_1\ldots x_n$ be a finite interpretation of an LTL formula over 2^{AP}. Then:

Temporal operators:

$W \models \varphi U \psi$ iff there exists $0 \leq i \leq n$ such that $w_I \models \psi$ and for all $0 \leq j < I$, $w_j \models \varphi$.

2.4. Finite automata on finite words

In typical model checking problem, we translate the negated LTL specification into a Buchi automaton then find an accepting cycle in the synchronous product of the model with the Buchi. However, this type of Buchi is not designed to deal with finite execution traces, so we cannot use Buchi to verify finite programs. A study about this has been carried out and presented in [7]. It said that we need to create a new translator which converts an LTL formula to a finite state automaton that monitors finite program traces. In this section, we define that finite automata, and the translation algorithm will be presented in detail later in following sections.

A finite automaton (FA) is defined as a 5-tuple (S, A, Δ, s_0, F), where S is a finite set of states, A is a finite set of labels which we call the alphabet of the automaton, $\Delta \subseteq S \times A \times S$ is a transition function, $s_0 \in S$ is the initial state, and $F \subseteq S$ is a set of accepting states.

An execution of FA is a finite sequence $\sigma = s_0\, a_0\, s_1 \ldots a_{n-1}\, s_n$, such that $(s_i\, a_i\, s_{i+1}) \in \Delta$ for each $0 \leq i \leq n$. An execution σ of FA is accepting if $s_n \in F$. Finally, FA accepts a finite word $w = x_0\ldots x_{n-1}$ over A, if there exists an accepting execution $\sigma=s_0x_0s_1\ldots x_{n-1}\, s_n$ of FA.

2.5. Java Pathfinder (JPF)

2.5.1. JPF top-level architecture

Java Pathfinder (JPF) is an explicit-state explorer for Java bytecode programs is developed at NASA Ames Research Center and was brought to open source community in 2005. JPF consists of many components as it is designed for easy-to-extend goal; however, it centralized around the JPF core which is a backtrackable

Virtual Machine for Java bytecode. It is built on top of the standard Java Virtual Machine. It enables the verification of various properties (deadlocks, uncaught exceptions, etc.), and monitors them while executing a Java program. By default, JPF stores all the explored states and it back-tracks when it hits a previously explored state. Users can customize the search, e.g. by forcing the search to backtrack on user-specified conditions, and can specify what part of the state space (if any) to be stored and used for matching. Users can also add listeners to detect specific conditions (and their violations). However, no facility is provided to perform model checking of temporal formulae in the sense of [11].

JPF have the ability to identify points in a Java program from where the execution could perform in different ways – execution choices – then systematically explore all of them. Theoretically, it means that JPF can explore all paths of a Java program instead of just one path like a normal Java Virtual Machine. Usually, choices are any value in a set of possible values at the running time. They include scheduling sequences or random values, but user can define their own type of choice such as state machine events. JPF allows doing this via a mechanism called ChoiceGenerator that will be briefly describe in following sections.

Nevertheless, executing all paths of a program may lead to state space explosion rapidly. Thus, JPF user several mechanisms to dealing with this problem, and the first one is *state matching:* each time JPF encounters a new state (i.e. picked a choice), it checks if there has been already a similar explored state. In that case, it stops exploring that path, backtracks to a previous choice point. On the other words, JPF can store and restore states during execution.

More specifically, JPF comprises of two main abstraction components: the *JVM*, and the *Search* object. Both of these components will be describe shortly as following:

The JVM is an implementation of the Virtual Machine specification which allows executing Java byte code instructions. In additional, as long as executing instructions, JVM generates program state representations that can then be used for state matching purpose. In particular, JVM can *forward* – generate and advance to a new state, store it to the backtrack stack for restoration later, *backtrack* – restore JVM to a state at the top of the backtrack stack. Even, JVM can restore to an arbitrary state no matter it is in top of the stack or not. The below picture represents the top-level design of JPF.

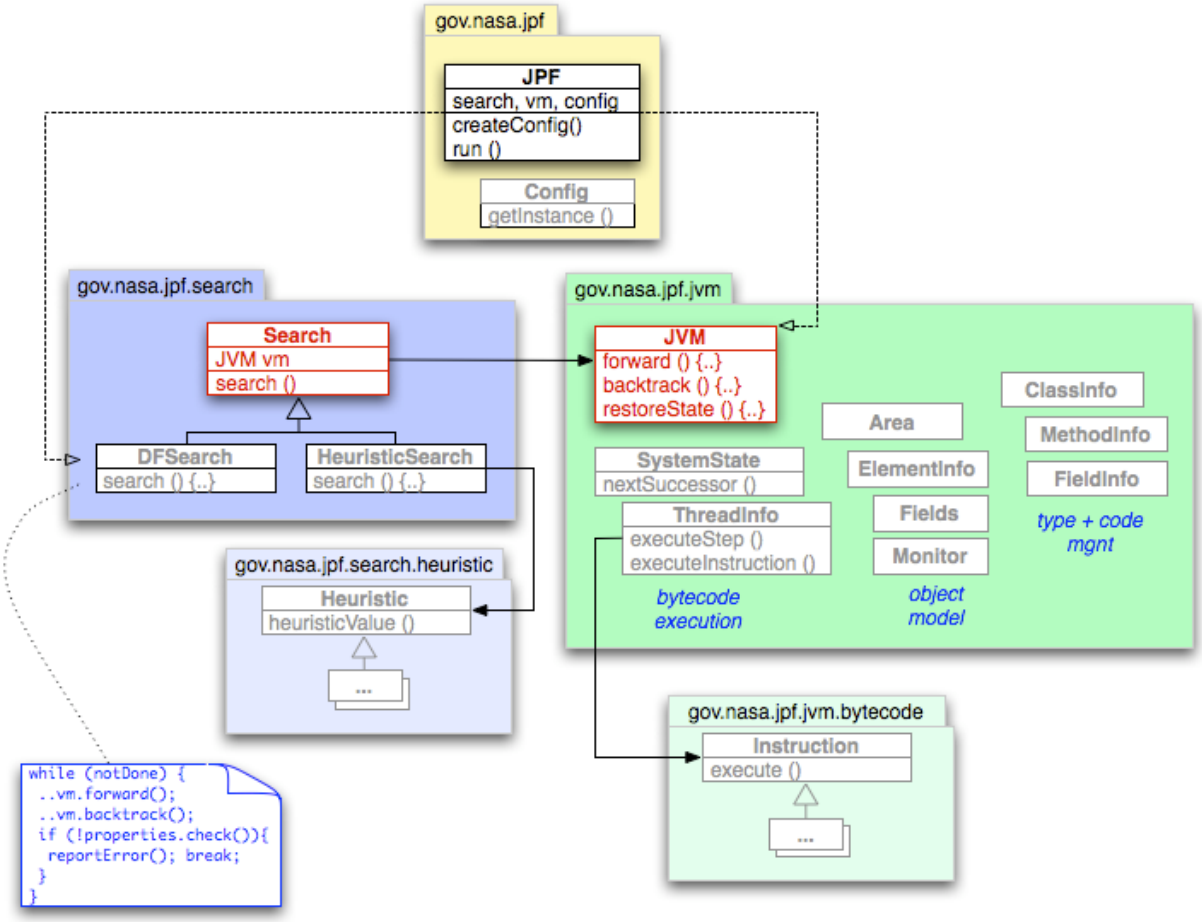

Figure 2: JPF top-level design[2]

The second major abstraction of JPF – the Search object – could be seen as a JVM driver. It follows different search strategies such as Depth First Search by default. The Search objects tells JVM when it should forward to a new state or backtrack to a previous generated one; even, the Search object can select a given stored state and tell JVM restore to that state exactly. The Search object also handles the properties evaluation (e.g. NotDeadlockedProperty). In addition to the default depth-first-search implementation, one could configure JPF to use others search strategy such as breadth-first-search, or heuristic strategies that can be parameterized to do many search types based on choosing the most suitable state from a priority queue of states at a given point. In order to provide a new search strategy for JPF, one could simple provide a single search method in the implementation which includes a main loop through state space until it has been absolutely explored or it detected a property violation.

2.5.2. ChoiceGenerators mechanism

In model checking, the most important thing is making the right choice within the resource constraints of the system and execution environment in order to reach the

[2] http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/design

interesting states. As discussed formerly, JPF determines all possible choice points during the execution and explores all of them. Choices mentioned here can be seen as a set of possible options from which JPF have to choose the right one. For example, a choice can be set of random values in case of random generation or a Thread in case of Thread scheduling points. The underlying mechanism used by JPF to generate choice points and systematically explore the whole state space is *ChoiceGenerators.* This special mechanism can be seen from different perspectives both from the application approach or JPF implementation approach.

If the set of choice values if small such as {true, false} for boolean, we can easily enumerate all of them. However, generating all choices from a large range of a type specific interval becomes difficult already (e.g. 0-10000 integer values), and it becomes impossible if the data type have an infinite values for an intervals (like real numbers type, we cannot generate all float number in the range 0-1). Hence, to overcome this problem we cannot stay in the ideal world of model checking where all possible choices is considered, and return to the real world where we have to make the choice set finite and manageable by using heuristics. Nevertheless, heuristic approach depends on the application under test and the specific domain it applied for. Thus, it may lead to several restrictions for JPF to generate choices:

- Choice mechanism must be separated for different data type (int choice from float choice etc.)
- Choice sets should be encapsulated in type specific objects. Thus, the JVM will only care about the most basic data types; it will use an interface to obtain choices otherwise.
- There should be a mechanism for selecting the classes representing heuristics of ChoiceGenerator instances at runtime.

The figure shown below represents an example of using a “randomly” velocity double value. In this example, we want to test how the system will execute below, at, and above a threshold (a certain specific value of application). As we use a threshold model, the possible choices now shrink from an infinite set to only three “interesting” values.

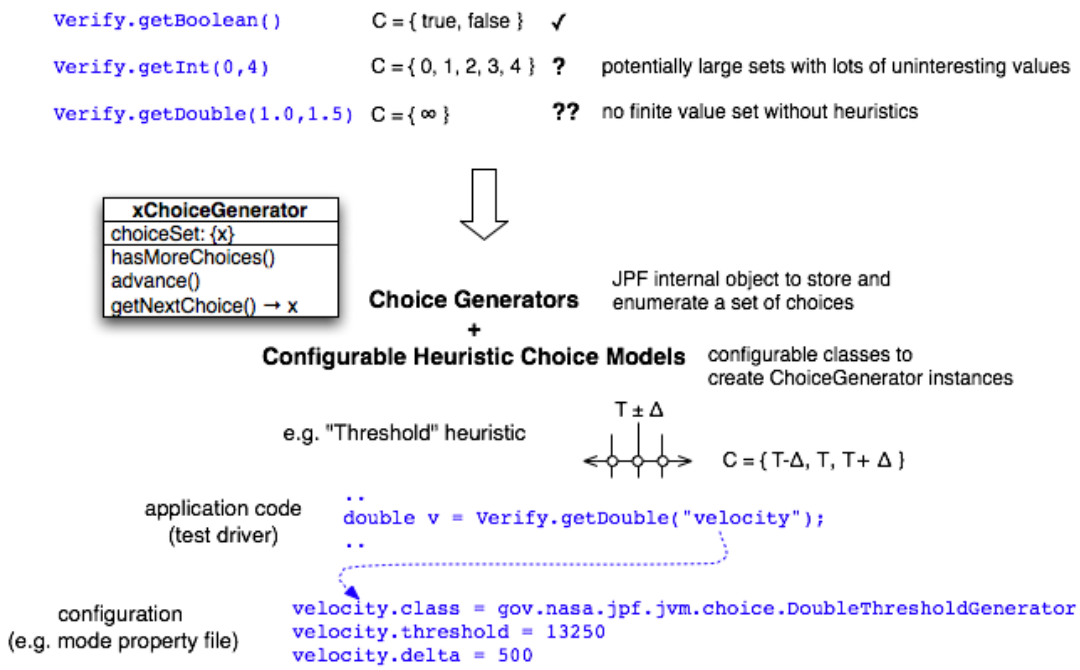

Figure 3. ChoiceGenerator motivation[3]

The sample source code in the above diagram does not mention explicitly the utilization of ChoiceGenerator class, it just give a symbolic name "velocity", which serve as a key for JPF to look up a corresponding class name from its configuration. But keep in mind that this is just a simple case. In fact, other parameterization (e.g. threshold, delta) are also needed for most heuristics, and the ChoiceGenerator constructor will look up those information in the JPF configuration data (e.g. the 'velocity.theshold' property). Each ChoiceGenerator has an appropriate symbolic name that can be used to look up all the parameters it needs.

ChoiceGenerator mechanism allows JPF's user to keep away from test driver modification. However, it would be very useful if we can use this mechanism systematically for scheduling choices in addition to data acquisition choices (i.e. this is often not controlled by the application itself). JPF's ChoiceGenerator is also used to handle this, but, firstly, we have to get familiar with some JPF terminology:

State is an abstracted presentation of the present execution status of the application under verification (almost thread and heap states), plus the execution traces that lead to this state. Every state is assigned with a unique id. State is wrapped in the SystemState object (mostly, some execution history which can only be accessed from the JVM object). This comprises of three main components:

- KernelState – The current execution state of the application (thread, heap)

[3] http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/choicegenerator

- Trail – The last *Transition*
- Current and next ChoiceGenerator – instances that holds the choice enumeration to generate different Transition (but not always to a new state).

Transition is a sequence of instruction which leads from one state to the next successor state. There is no transition that lies in two threads; it is all in one thread (i.e. no context switching here). It is possible to have several out-going transition from one state but not necessary all of them lead to a new state.

A new transition is started when JPF picks a new choice generated by the corresponding ChoiceGenerator. This choice can vary from thread to other data types. In the other hands, a transition is terminated in a choice point (there are possible existences of choices) and started when JPF select a choice from the possible choice set.

2.5.3. Listeners

Listeners are the most important mechanism for building JPF extensions. They allow observe, interact, and extend the execution of JPF at runtime without needed to modify the internal JPF core's structure. Listeners are notified whenever its registered events occurred; hence, you can do almost everything with listeners because they are executed at the same level as JPF.

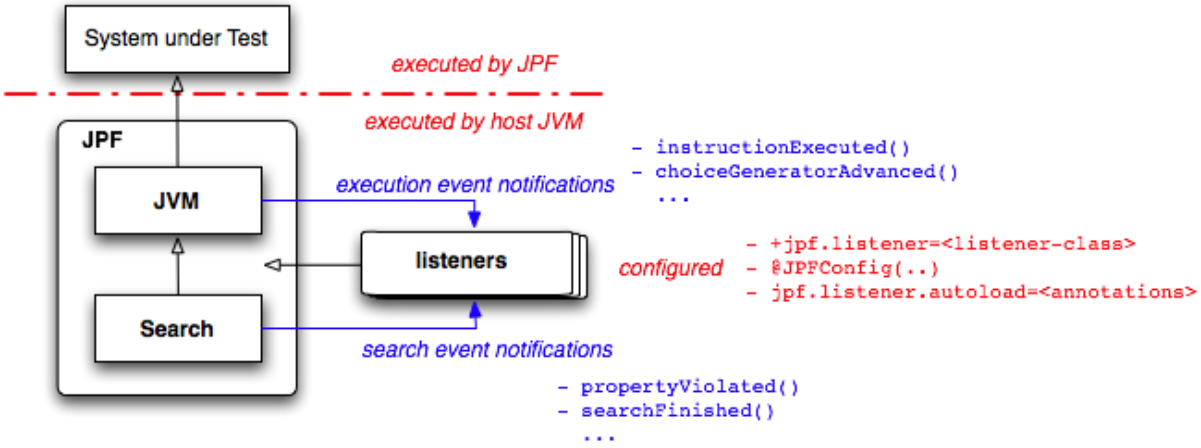

Figure 4. JPF Listeners[4]

Despite its important role, this mechanism is quite easy to understand. Java PathFinder utilizes Observer pattern [12] to notify about certain events of its operation to registered observer instances. Such notifications cover a large set of operation from high level events such as *searchStarted* to low level events such as whenever a

[4] http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/listener

bytecode instruction got executed. When an observer instance is notified of its registered event, it is also provided with the corresponding source, which is either the *JVM* or the *Search* instance, of the notification. From these corresponding sources, the observer can get access to the internal state of JPF to extract the necessary information as well as monitor the JPF's exploration process.

In order to register a listener with JPF, one must specify its existence in JPF configuration data either via command line, or a *.jpf property file. Besides, listeners can automatically load when JPF encounter its associated annotations.

As JPF is designed around two components (i.e. Search object and JVM), there are two types of listeners represented by two interfaces corresponding with each event sources: *SearchListeners* and *VMListeners.* However, both interfaces are quite large and it does not make sense to have a listener that only interests in several events must implement every method in a given interface. This case is similar when a listener wants to be notified of both JVM and Search events. Hence, there are some "adapter" classes that implement all required methods in the two interfaces with empty method bodies. Concrete listener, which extends these adapter classes, only has to override its interested method.

Most of listener implementations use adapter classes, particularly since these classes also implement two other interface that are usually utilized in combination with Search/VMListeners: *Property* (to specify program properties) and *PublisherExtension* (to put the verification output within the JPF reporting system).

ListenerAdapter is the basic adapter class for *SearchListener, VMListener, and PublisherExtension.* This adapter implementation is mostly used to extract the information during the JPF execution.

PropertyListenerAdapter is another adapter class that will be used if we need to define a program property. Figure 5 represent the different types of JPF listener.

From the above figure, we can choose to extend any adapter class or directly implement a specific listener interface as we want. In case we choose the adapter classes, JPF will register our listener automatically; we have to do it ourselves otherwise. Besides, our listener not only wants to be notified but also wants to collect the information from the notification source. Hence, JPF parameterizes each *SearchListener* and *VMListener* method with a corresponding *search*, and *vm* object respectively. The listener can use these objects to query and interact with JPF while it is running.

2.6. Symbolic Execution

The main idea behind symbolic execution [13] is to use symbolic values, instead of actual data, as input values, and to present the values of program variables as symbolic expressions. As a result, the output values computed by a program are expressed as a function of the input symbolic values. The state of a symbolic executed program includes the (symbolic) values of program variables, a *path condition* (PC) and a program counter. The path condition is a (quantifier-free) boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. The program counter defines the next statement to be executed. A symbolic execution tree characterizes the execution paths follow the particular associated path. The program counter defines the next paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

Let's illustrate the difference between concrete and symbolic execution, consider the simple code fragment in Figure below (left) that computes the absolute difference between two input integers x and y. Assume that the values of the input parameters are x=2 and y=1.

Concrete execution will follow only one program path, corresponding to the true branch of the *if* statement at line [1]; this execution does not violate the assertion.

In contrast, symbolic execution starts with symbolic, rather than concrete, values, $x = Sym_x$, $y = Sym_y$, and the initial value of PC is *true.* The corresponding (simplified) execution tree illustrated in the below figure (right). At each branch point, PC is updated with constraint on the inputs in order to choose between alternative paths. For example, after executing line [1] in the code, both alternatives of the *if* statement are possible, and PC is updated accordingly. If the path condition becomes "false", it means that the corresponding path is infeasible (and symbolic execution does not continue for that path).

In this example, symbolic execution explores three different feasible paths, it determines that a fourth path is infeasible and it reports an assertion violation (for Path 3).

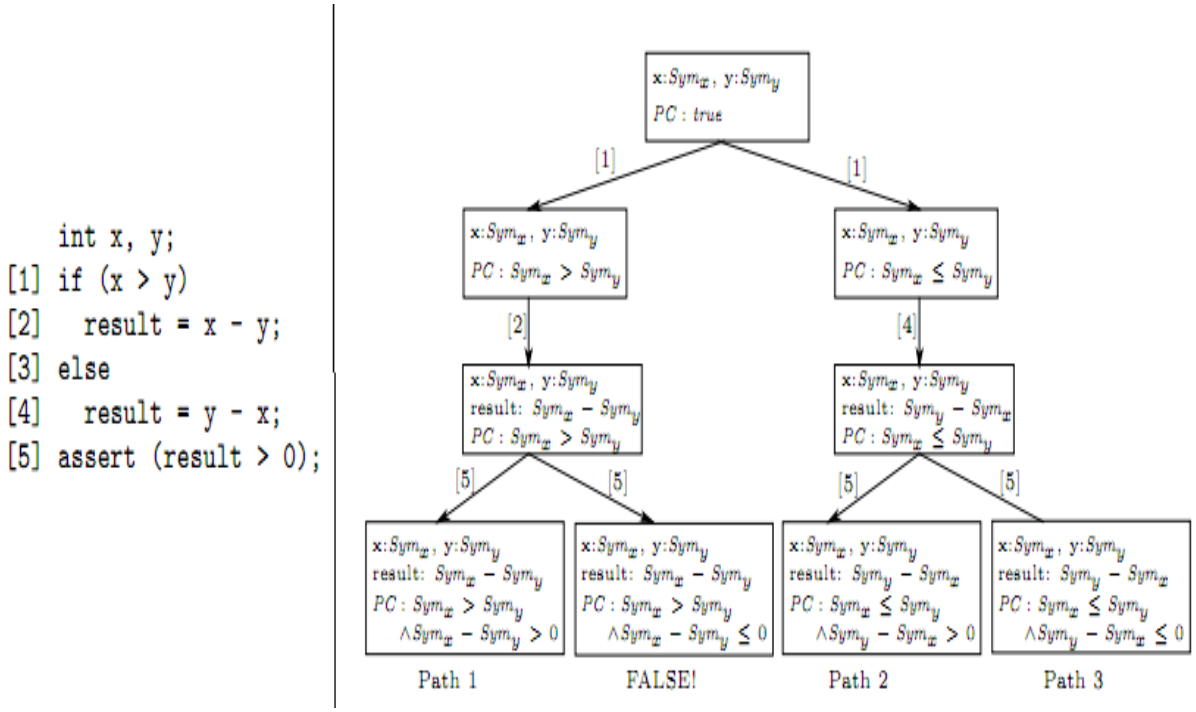

Figure 5. Symbolic execution (left) and corresponding execution tree (right)

2.7. Symbolic Pathfinder

The symbolic execution framework is built as an extension of JPF – the framework performs a non-standard bytecode interpretation uses JPF to systematically generate and execute the symbolic execution tree of the code under analysis. The two key mechanisms that Symbolic Pathfinder used were:

- JPF's bytecode instruction factory and
- Attributes associated with the program state.

The instruction factory allows replacing or extending the standard, concrete execution semantics of bytecodes with a non-standard (symbolic) execution, as desired. The symbolic information is stored in *attributes* associated with program data (fields, stack operands and local variables) and it is propagated as needed, during symbolic execution.

These mechanisms together allow dynamic modification of execution semantics, i.e., changing mid-stream from a system-level, concrete execution semantics to a symbolic execution semantic. In particular, Symbolic Pathfinder used JPF's choice generator discussed in the above sections for handling branching conditions during symbolic execution, and listeners, for printing the results of the symbolic analysis and for enabling dynamic change of execution semantics. It also used native peers, for modeling native libraries, e.g., to capture java.lang.Math library calls and to send them to the constraint solver.

Chapter 3. Implementation

3.1. Construct a LTL formula parser and an atomic proposition parser

We have written LTL grammars for automatically generating lexer and parser by ANTLR. After parsing a LTL formula, we obtain an instance of Formula<String> and extract variables from that formula. Following is the specification of a LTL formula accepted by these parsers.

3.1.1. Supported types of LTL operator

AND	:	/\	&&
OR	:	\/	||
UNTIL		:	U
WEAK_UNTIL		:	W
RELEASE	:	V	
WEAK_RELEASE	:	M	
NOT	:	!	
NEXT	:	X	
ALWAYS	:	[]	
EVENTUALLY		:	<>
IMPLIES	:	->	

3.1.2. Supported types of atomic proposition

Because this tool aim to verify the direct Java source code, it must have the ability to accept any Boolean Java variables (including local and global variables), and any Boolean Java expression. Besides, one could want to check if a function is called at a given time in the execution, so a method signature must be taken into account. The details of these atomic propositions are defined as follows:

3.1.2.1 A method signature

A method signature can be an atom. It must include the full package name (if any), the class name, the method name and types of its parameters (if any). For instance:

packageName.ClassName.methodName(int, float, String, T)

or

ClassName.methodName(T)

However, if the LTL specification and the method are declared in the same class, we can ormit the package name and class name so that we only need to write as methodName(T).

3.1.2.2. A boolean variable

An atom can be a boolean variable, and it must be specify as follow in the LTL formula:

- A field in a class: It must include the full package name (if any), the class name, the field name
  Ex: packageName.ClassName.field
- A local variable in a method: It must include its method signature and its name
  Ex: packageName.ClassName.methodName(T).var
  However, as we did with the method name, the variable name can also ormit the package name and the class name in case it is declared in the same class with the LTL specification.

3.1.2.3. A relation

An atom can be a relation between two Java expressions. For instance:

(x + y) * z – 3.0 > u / 5

Where x, y, z, u are variable that follow the syntax defined in 3.1.2.2

3.1.3. LTL grammar

An LTL formula f might include any atomic proposition as discussed in the previous section, combined with unary or binary, Boolean and/or temporal operators, using the following grammar:

```
f :       binf;

binf :    orf (IMPLIES binf)?;

orf :     andf (OR orf)?;

andf :    untilf (AND andf)?;

untilf :  releasef ((UNTIL | WEAK_UNTIL) untilf)?;

releasef :propf ((RELEASE | WEAK_RELEASE) releasef)?;

propf :   TRUE | FALSE
          | atom
          | unf propf
          | '(' f ')'
          ;

unf :     | NOT | ALWAYS | EVENTUALLY
    ;
/** lexer rules **/
AND             : '/\\';
ROBBYJO_AND     : '&&';
OR              : '\\/';
ROBBYJO_OR      : '||';
UNTIL           : 'U';
WEAK_UNTIL      : 'W';
RELEASE         : 'V';
WEAK_RELEASE    : 'M';
NOT             : '!';
ALWAYS          : '[]';
EVENTUALLY      : '<>';
IMPLIES         : '->';
```

Figure 6. LTL grammar

Where **atom** is any atomic proposition that follows the semantic described in Section 3.1.2. And here is the grammar for atomic proposition:

```
atom
:
| method
| var
| exp ('==' | '!=' | '>=' | '<=' | '>' | '<') exp
;

method
:ID ('.' ID)* (
( '('(type ',' )* type ')')
|
'(' ')'
);

type
: ID ('[]')*;

exp
: mult ( '+' mult | '-' mult )*;

mult
: factor ('*' factor | '/' factor)*;

factor
: '(' exp ')'
| var
| number
;

var
: (
( ID('.' ID)*)
| (method '.' ID)
)
('[' INT ']')* ( '#' INT )?
      ;
number
  :  INT | FLOAT | '+' factor | '-' factor;


ID  : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;

INT : '0'..'9'+ ;

FLOAT
  :   ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
  |   '.' ('0'..'9')+ EXPONENT?
  |   ('0'..'9')+ EXPONENT
  ;


EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;
```

Figure 7. Atomic proposition grammar

3.2. Implement a LTL to FSA translator

In typical model checking problem, we translate the negation of LTL specification into a Buchi automaton then find an accepting cycle in the synchronous product of the system under verification with the Buchi. However, this type of Buchi is designed to deal with the infinite execution traces, it only accepts infinite words. Hence, we can't use this automaton to verify finite programs. This issue has been addressed in the Dimitra's paper [7], it said that we have to create a new translator which converts LTL to finite state automata that check finite program traces. This finite automaton accepts precisely finite words that satisfy an LTL formula. In the following Section 3.2.1, the algorithm presented in that paper will be described in detail and also how it was implemented actually in the tool.

3.2.1. Translation algorithm

This algorithm modifies an efficient tableau-like translation algorithm [9] in order to apply the finite trace semantics of LTL in above sections. Firstly, let's consider the intuition behind this construction.

Before translating the formula into a finite automaton, we need to rewrite it so that it is in negation normal form. On the other words, it means that all negations are pushed inside until the formula reaches a state where all of them precede only atomic propositions. For instance, the formula ![]φ after rewrote, it will become <>!φ.

The core data structure used in this algorithm is a graph node which includes the following fields:

ID: the unique identification of this node

INCOMING: the set of incoming nodes that have edges lead to this node.

NEW: the set of unprocessed LTL formulae but must hold on the current state

OLD: the set of processed LTL formulae. Each formula in NEW will be moved to OLD when it is processed.

NEXT: the set of LTL formulae that required holding in every immediate successor of this node

Assume that we are translating an arbitrary LTL formula. After rewrite the formula, this algorithm will begin by means of initializing a node which contains the formula needed to translate in its NEW field. All other fields of the initial node are empty. The most important part of this algorithm is *expanding a graph node* which is illustrated in the Figure 1. A node is considered to be successfully expanded if there are no formulae left in the NEW field. After the expanding process finished, we

obtained an automaton which is a set of graph nodes. Let's move to the expanding algorithm sketched in Figure 1 by considering two possible cases: 1) there are none, and 2) there are some formula left in NEW field of a node.

```
1  ListOfNodes expand(ListOfNodes automaton) {
2      if (isProcessed()) { //the NEW field is empty
3        //two nodes are equivalent if they have the same NEXT field
4        if (∃ temp ∈ automaton such that equivalent(this, temp)) {
5          //if there already exists an equivalent node,
6          //we modify that one instead of adding a new node
7          temp.modify(this);
8          return automaton;
9        } else { //processed node to be added to the automaton
10         //adds the processed node to the automaton
11         automaton.add(this);
12         //expands the immediate successors of this node
13         Node newNode = getImmediate();
14         return newNode.expand(automaton);
15       }
16     } else { //the NEW field is nontempty, so keep processing
17       //process the formula in NEW one by one
18       f = newFormulae.first();
19       NEW.remove(f);
20       //if there're contradicting formulae in the OLD field
21       if (isContradicting(f))
22         return automaton; //discard this node
23       //process the formula by breaking it down to literals
24       if (f.isLiteral()) {
25         OLD.add(f); //simple move f to old field
26         return expand(automaton);
27       } else {
28         if(f is a "ϕ ∧ ψ" formula) {
29         //both sub formulae must hold to make f true
30         //so both are added to the NEW field
31           addToNew(ϕ);
32           addToNew(ψ);
33           return expand(automaton);
34         }
35         if(f is a OR, UNTIL, WEAK_UNTIL, or RELEASE formula) {
36           //there are alternative ways to make f true
37           //so split this node into two nodes
38           Node temp = split(f);
39           return temp.expand(this.expand(automaton));
40         }
41       }
42     }
43   }
```

Figure 8. Node expansion algorithm

Case 1: There are no formula left in NEW field (lines 2-15)

If the NEW field is empty, it means that this node has been fully processed and can be considered as a node of the automaton. Nevertheless, if we add it to the automaton immediately, there might be an equivalent node which has been already existed in the automaton. In such cases, we will have two duplicate nodes. In order to avoid adding multiple equivalent nodes to the resulting automaton, we must check if there exists an equivalent node there (line 4). Two nodes are considered as equivalent if they have the same OLD and NEXT fields. If we found such an equivalent node, namely *temp*, existed in the automaton, we will modify *temp* by adding (by set union) the INCOMING field of current node to the INCOMING field of *temp*, instead of adding a duplicate node. If no equivalent node found, then we add the current node to the automaton (line 11), and a new node, namely *newNode*, which represents the immediate successors of current node is added to the unprocessed ones (line 13-14). *newNode*'s INCOMING field is the same as the current node and its NEW field is the same as the NEXT field of the current node.

Case 2: There are unprocessed formula left in NEW field (lines 16-42)

All formulae in the NEW field will be transferred to the OLD field one by one in the following manner. Take a formula in the NEW field (lines 18-19); we break it down until we get to the literals (atomic positions or negated propositions) that must hold in order to make the formula true (lines 24-41). For instance, $\varphi \wedge \psi$ will be broken down by means of adding both ψ and φ to the NEW field of the node. However, if it is a $\vee$ or U (Until) formula, there exist alternative ways for that formula becomes true. In that circumstance, the node needs to be split in two new nodes, where each of them presents an alternative way of making the formula true. For instance, assume that we have a node which has the formula $\varphi \vee \psi$ in the NEW field. Hence, we have to split this node into two new nodes (lines 35-40): one which needs to make φ true and another that needs to make ψ true.

In cases of formulae that only consist of logical operators; we just need to split the node into two nodes. However, the sense is different if they contain temporal operators such as U (Until) or V (Release). In those cases, it is necessary to also push obligations to the NEXT field of the original node's immediate successors. The following identities define how obligations are pushed to successors:

- $\varphi \, U \, \psi \equiv \psi \vee (\varphi \wedge X(\varphi \, U \, \psi))$
- $\varphi \, V \, \psi \equiv \psi \wedge (\varphi \vee X(\varphi \, V \, \psi))$

The following table presents the formulae need to be added to different fields of obtained nodes for each type of formula *f* that cause a node to split. The INCOMING and OLD fields of the resulting nodes will be kept the same as the original node, while

the NEW and NEXT fields are the combination of the ones of the original node with the fields shown in the table.

f	NEW1(f)	NEXT1 (f)	NEW2 (f)
$\varphi\ U\ \psi$	$\{\varphi\}$	$\{\varphi\ U\ \psi\}$	$\{\psi\}$
$\varphi\ V\ \psi$	$\{\psi\}$	$\{\varphi\ V\ \psi\}$	$\{\varphi, \psi\}$
$\varphi \vee \psi$	$\{\varphi\}$	$\varnothing$	$\{\psi\}$

Figure 9: Formula expansions utilized in Node splitting

Nevertheless, one node might contain both a formula φ and its negation !φ as they are transferred to the OLD field during processing. If such contradictions are obtained, the node will be discarded (lines 21-22).

After all nodes are processed, we can build an automaton from the obtained nodes as following way. Each node corresponds with a state of the automaton. Each element in the INCOMING field will become an edge of the automaton. The initial node is the one that only has outgoing edges. All incoming edges of a node N are labeled with literals in the OLD field of N (i.e., literals must hold at N). Now, the last thing we have to do is manipulating the accepting condition for each state by applying finite-trace semantics.

3.2.2. Selection of accepting conditions on finite-trace semantics

The most difference between the common LTL to Buchi translation and the algorithm discussed in the previous section is how to select the accepting conditions. Every infinite execution of an automaton which generated by the above algorithm satisfies the safety condition of the original formula *f*. This is guarantee by the construction process of the automaton as presented in [9]. Hence, the selection of accepting conditions must be imposed, to guarantee that eventualities are also satisfied. In particular, the accepting conditions have to be chosen so that whenever a node contains φUψ, some successor node will include ψ.

With the finite-trace semantics mentioned in Section 2.2.2, we have to impose accepting conditions so that any finite execution of the automaton also satisfies all the eventualities. The required eventualities are reflected by the formulae in the NEXT field of the last state of any finite execution. On the other words, a state is considered to satisfy its eventuality requirements if there are no U formulae in its NEXT field.

Recall that all literals in the OLD field are only used to label the edge between states of the automaton. However, in the above algorithm, two nodes are considered to be equivalent if they have the same NEXT and OLD fields. Although comparing two

nodes in this way still generate a correct automaton but it is not efficient because the OLD field do not have any role in the identification of accepting conditions. Hence, we modify the algorithm a bit so that two states are equivalent if they have the same NEXT field only.

Base on the above observation about the OLD field, we can collapse more equivalent states. When collapsing two nodes, the resulting node must record each component's INCOMING and OLD field. However, if the literals in the OLD field of both nodes are the same, the OLD field of the resulting node is the same too, and its INCOMING field is the union of two original nodes.

3.2.3. Proof of correctness

This algorithm is based on the typical tableau-like LTL to Buchi algorithm [9], so the proof of correctness is based on the proof of [9] also. What we need to prove additionally is the selection of accepting conditions discussed above guarantee that any finite sequence if accepted if and only if it satisfies the original LTL claim for which we build the automaton. The translation the algorithm follows ensures this requirement because when a node is expanded; all the possibilities of making its requirements true are considered.

Besides, the algorithm ensures that any U formula that must be satisfied stays in the NEXT field of the node and all of its immediate successors because it pushed as eventual obligations when splitting nodes. Those eventualities remain in the successors until its right-hand side formula becomes true. Thus, it means that there are eventualities that remain to be examined during the path followed to a node which have a U formula in its NEXT field.

Let's consider an LTL property <>(a ∨ b). It states that a or b will eventually become true. On the other hands, a finite trace is accepted iff at least a or b become true during the execution. Following is the automaton generated by the discussed algorithm written in the PROMELA syntax:

```
never {
S0:
      if
      :: (true) -> goto S0
      :: a -> goto accept_S1
      :: b -> goto accept_S1
      fi;

accept_S1:
      if
      :: (true) -> goto accept_S1
      fi;
}
```

Figure 10. PROMELA syntax for the formula <>(a ∨ b)

The generated automaton can also be viewed as the following figure where (1) is the accepting state.

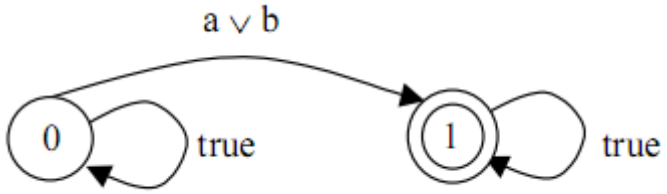

Figure 11. Finite automaton for the formula <>(a ∨ b)

3.3. Implement a listener for JPF to check the satisfiability of finite execution traces

I have implemented an auto load listener for two annotations (@LTLSpec and @LTLSpecFile). Hence, it is initialized whenever JPF encounters the above annotations when classes loaded. As soon as it is initialized, it will get the LTL claim from its corresponding annotations in the main class of the system under verification and then translates the LTL formula into a finite automaton using the translator discussed in the above sections. The algorithm for monitoring the system under verification with the generated automaton is sketched in Figure 4. In the monitoring algorithm, *vm,* and *search* are instance of *JVM* and *Search* object respectively. Path condition represents the constraints over symbolic variables for the current path of the execution.

```
1  instructionExecuted(vm) {
2      Extract variable values
3
4      turns off concrete state matching;
5
6      for (each outgoing transition from the current states) {
7        //guard condition is the conjunction of all the literals that
8        // must hold in the next states
9        combines the guard conditions with the current path condition
10       if (the combined condition is satisfiable) {
11         successors.add(transition.getNext());
12         update the path condition;
13       }
14     }
15
16     if (cannot find a successor state from the current states) {
17       report violation;
18     }
19     else {
20       moves from the current states to the successors;
21
22       if (path condition has changed) {
23         branches the execution of JPF;
24       }
25     }
26   }
27
28 stateAdvanced(search) {
29     if (search.isEndState() && !isAcceptingState()) {
30         report violation;
31     }
32   }
```

Figure 12. Monitoring algorithm

The listener checks the satisfiability of finite run sequences with this automaton in the following way. Whenever an instruction gets executed, our listener does the following things (lines 1-26). Firstly, this task requires extracting and updating the concrete and symbolic value of fields and local variables appeared in the LTL specification after executing every instruction (line 2). A symbolic value is an instance of *Expression* class and is stored in the attribute of the field and local variable. Fields are stored in the static and dynamic area; local variables are stored in the stack frames.

Secondly, it checks which transitions from current states in the automaton are enabled. In order to do this, it examines if the guard labeling these transitions is satisfiable with the current state of the program being checked. We combine the guard constraint with the symbolic execution's constraint (i.e., path condition), then check this conjunction for satisfiability (lines 9-10). For each satisfied transition, we get its next state, and add it to the successor state set (line 11). The details of checking guard

conditions will be discussed in the following section. The set of successors that are reached through enabled transitions are then updated to be the current states of the automaton (line 20). In addition, if the combined constraints differ from the original path condition, we must update it to be the new one (line 12). If successor set is empty (i.e. the automaton cannot make a step), it means that the specific execution trace violated the property. In that circumstance, we stop exploring the program and report the violation (lines 16-17).

As we are monitoring against the automaton, when we move from the current states to others in the automaton, we also have to branch the JPF execution accordingly (line 23). However, if we branch the JPF exploration every time an instruction executed, the state space will rise rapidly, and can lead to state-explosion problem [6]. Hence, we only make new branches in the JPF execution if the path condition has changed. This observation does not affect the semantics of the monitoring semantics, and huge impact in the reduction of the state space because we can ignore every low-level instruction.

Each execution trace corresponds to a path in the program. When the program reach the end of a path, the listener checks if there exist at least one accepting state within the set of current states. If any, that finite run sequence satisfied the LTL property. Otherwise, listener report the property is violated by the specific program trace.

3.4. Checking guard conditions

Every transition in the automata has a guard which is the conjunction of all the literals that must hold in the next state. Therefore, in order to make a transition from the current state, its guard must be satisfiable. Because checking the guard involves checking all of its literals (atomic propositions or its negation), we break it down to checking whether an atom is satisfiable. Recall that there are three type of supported atomic proposition: method signature, Boolean variable, and Java Boolean expression. The three sub section will discuss how to check each type of atomic propositions in detail.

3.4.1. Atom is a method signature

Because JPF represents every bytecode as an appropriate *Instruction* object, we only care about the *InvokeInstruction* instances that represent method invocation instructions. If an atom is a method signature, it is satisfied if and only if the executed instruction is an instance of *InvokeInstruction* and signature of the invoked method is the same as the atom. Therefore, the checking task requires extracting the method's full name including package name, class name, and the type of its parameters from the internal JPF's state.

3.4.2. Atom is a boolean variable

If an atom is a boolean variable, we treat the *"false"* as an integer value: 0 and "*true*" as not equal to 0. So A.x mean that A.x != 0 and !A.x mean that A.x == 0; in the other hand, it's also an expression.

3.4.3. Atom is a Java expression

If there are any field names that do not exist in the program or any local variables that do not exist in the current method or the method in which it declared is never called, the atom is unsatisfiable. Otherwise, we create a parser that receives an atom to parse it and check it for the satisfiability.

Presently, we allow mixed concrete and symbolic checking. An atom is symbolic when there is at least one variable has non-null symbolic value. It's concrete otherwise.

In concrete mode, we use the parser mentioned above to evaluate the value of each expression. Then according to the relation operator, we find out whether atom is satisfied or not by the returned value is *true* or *false.*

In symbolic mode, an atom cannot have the exactly value *true* or *false*, for a symbolic value represents a set of concrete value and we can't take an arbitrary concrete value from that set or all of them. Hence, instead of checking an atom is true or false, we consider it as a symbolic constraint and indicate whether it is satisfiable in the current symbolic state of SUT.

Recall that a relation atom is symbolic if one of its variable is symbolic (the symbolic value is not null), so there might be still some constants or concrete variables. Therefore, we have to convert all of them from concrete to a symbolic constant according to its type. For example, we have the following atomic proposition: x + y < 5 where x is an integer symbolic variable and has the value SymInt_1; y is another integer variable but has the concrete value 3. As a result, we convert y and 5 to symbolic constants: SymConst_3 and SymConst_5. After that, we got the following symbolic constraint: SymInt_1 + SymConst_3 < SymcConst_5. This constraint represents the atom, so the atom is satisfiable if and only if this constraint is satisfiable in the current symbolic state of the program. On the other hands, the symbolic state of system under verification is represented by the path condition which is also a type of symbolic constraint. At last, we add the atom constraint to path condition and check if the new constraint is satisfiability.

3.5. Extend PCChoiceGenerator for the branching purpose

In section 3.4.3 when we check a guard condition in symbolic mode, we have a new constraint which is the conjunction of path condition and the symbolic constraint represents the atom. Each time dealing with a symbolic atom, the obtained symbolic

constraint is added to the path condition. Finally, after checking the guard, we have a new path condition that is conjunction of the original path condition and the atom constraints. In a given state of the program, we suppose that there exists several enabled transitions and the path condition after checking each transition is different. As a consequence, we must create a new branch for the program for each transition.

Hence, we treat every execution step as a branching point. If the path condition is unchanged after checking all guard conditions, the program continues exploring. Otherwise, we have to create new branch for each new path condition with the corresponding successor states. ChoiceGenerator is used to handle the branching purpose.

We create a new ChoiceGenerator – AutomataChocieGenerator - which extends PCChoiceGenerator. It overrides the getCurrentPC() method to return a corresponding path condition for each branch. If we need to branch the program, a new instance of this class is created and registered to the next ChoiceGenerator after the instruction get executed.

Chapter 4. Experiment

This tool is provided with over 20 examples on both concrete and symbolic mode. It can find violation of temporal properties in concurrent programs with multiple threads. Also, it allows specifying more complicated temporal properties by differentiating between instances of a class. We will consider some experimental examples below to have an overview of what this tool can do and how to use it.

4.1. Detect race condition in singleton pattern example

The below example will highlight the problem of singleton pattern which restricts the instantiation of a class to one object. Consider the below implementation of a singleton class. This example illustrates the ability of this tool to detect a race condition of interleaving threads that may not be found by standard tests.

Because of the singleton pattern, the instantiation of Singleton class should be only one object. On the other words, the once the getSingleton() method is called, it will never be called again. This temporal property is expressed as in the implementation [](getSingleton() -> []!getSingleton()). It means that in every execution traces, once the getsingleton() method is called, it must not be called after that.

However, there is a possibility that there are two thread reaches the *if* statement (lines 8-12). After that, one thread calls getSingleton() method, then temporary sleep. At that time, the second thread also does the same, so the claim about singleton pattern is violated.

```
3  @LTLSpec("[](getSingleton() -> []!getSingleton())")
4  public class Singleton extends Thread {
5
6    private static Object mMySingleton;
7
8    public void run() {
9      if (mMySingleton == null) {
10       mMySingleton = getSingleton();
11       yield();
12     }
13   }
14
15   // expected to be called only one time
16   private Object getSingleton() {
17     return new Object();
18   }
19
20   public static void main(final String[] args)
21     throws InterruptedException {
22
23     Thread m1 = new Singleton();
24     Thread m2 = new Singleton();
25
26     m1.start();
27     m2.start();
28
29     Thread.sleep(500);
30   }
31 }
```

Figure 13. Race condition example

And here is the verification result collected after running the tool for this example. From the result, we can easily see that there are two interleaving threads and it violated when the second thread tried to called getSingleton() method while the first thread has already done it.

```
====================================================== error #1
gov.nasa.jpf.ltl.finite.LTLListener
LTL violated: [](getSingleton() -> []!getSingleton())

====================================================== snapshot #1
thread index=1,name=Thread-0,status=RUNNING,this=Singleton@13f,priority=5,lockCount=0,suspendCount=0
  call stack:
        at Singleton.getSingleton(Singleton.java:17)
        at Singleton.run(Singleton.java:10)

thread index=2,name=Thread-1,status=RUNNING,this=Singleton@151,priority=5,lockCount=0,suspendCount=0
  call stack:
        at Singleton.run(Singleton.java:9)
```

Figure 14. Verification result for race condition example

4.2. Verifying in symbolic mode

This example illustrates the benefit of symbolic over concrete verification. In concrete execution, we must provide input (test cases) to verify a given program, and the verification result may be different with different input sets. Unlike the concrete verification, symbolic verification does not require having any concrete value, but still can find violation in all feasible path of the program. Hence, we can, theoretically, find all possible counter-examples in a finite program under symbolic execution.

Consider the figure below, the LTL formula claims that *field* will keep holding until the local variable *z* greater than zero.

```
10 @LTLSpec("field == 0 U myMethod(int).z > 0")
11 public class Until_Sym {
12   @Symbolic("true")
13   static int field = 0;
14
15   public int myMethod(int param) {
16     int z;
17     if (param <= 0 && field > 0)
18       z = field - param + 1;
19     else
20       z = param;
21     return z; // error when param <= 0 && field <= 0,
22               // cannot detect in concrete mode
23   }
24
25   public static void main(String[] args) {
26     Until_Sym until = new Until_Sym();
27     until.myMethod(5);
28   }
29 }
```

Figure 15. Symbolic example

Following is the automata generated for the given LTL formula:

```
never {
S0:
      if
      :: (finite.symbolic.Until_Sym.field == 0) -> goto S0
      :: (finite.symbolic.Until_Sym.myMethod(int).z > 0) -> goto accept_S1
      fi;

accept_S1:
      if
      :: (1) -> goto accept_S1
      fi;

}
```

Figure 16. Generated finite-automata in PROMELA syntax

If we verify this example in concrete mode as specified in the test driver (*param* is set to 5), we will not be able to find any counter-example because there is just one path following the *else* statement. However, when we detect the above example in symbolic mode, we will find out there is a path that violates the LTL formula. And here is the result of verification corresponding with error stack trace:

```
====================================================== error #1
gov.nasa.jpf.ltl.finite.LTLListener
LTL violated: field == 0 U myMethod(int).z > 0

====================================================== snapshot #1
thread index=0,name=main,status=RUNNING,this=java.lang.Thread@2,target=null,priority=5,lockCount=0,suspendCount=0
  call stack:
        at finite.symbolic.Until_Sym.myMethod(Until_Sym.java:17)
        at finite.symbolic.Until_Sym.main(Until_Sym.java:27)
```

Figure 17. Verification result for symbolic example

4.3. Distinguishing between instances of a class

The tool can distinguish between instances of a class, so you can have power to specify complicated temporal properties in a particular object-oriented manner. Currently, you can differentiate one instances from another by adding the #order (such as #1, #2, #3...) suffix at the end of a field in the LTL specification. If a field doesn't end with that suffix, it'll be checked for every instance by default.

Let’s consider the simple example below. The LTL specification state that: the field *k* of the first instance plus 2 will, eventually, equal to the field *k* of the second instance of the same class.

```
 9
10 @LTLSpec("<> k#1 + 2 == k#2")
11 public class MultiInstance {
12   int k;
13   public void foo() {
14     k++;
15   }
16
17   public static void main(String [] args) {
18     new MultiInstance();
19     MultiInstance a = new MultiInstance();
20     a.foo();
21     //a.foo();
22   }
23 }
```

Figure 17. Multi-instance example

The above example will produce a counter-example. However, if we delete the comment in line 21, the program will satisfy the LTL claim.

Chapter 5. Conclusion

The study presented in this thesis is the result of developing an efficient tool to verify finite Java programs against temporal properties written in Linear Temporal Logic. It combines symbolic execution with program monitoring to explore all feasible paths in finite Java programs, and verifies them during the execution. The tool has a parser which can accept standard LTL formulae with the atomic proposition set are almost every Java expression. It presents an algorithm which modifies standard LTL to Buchi automata construction techniques to deal with finite execution traces.

This approach is clearly more efficient than using Buchi automata for the same purpose. A benefit of this approach is that it does not require the detection of accepting cycles in the product of the automaton with the program traces. This is because detecting such cycles often uses the nested-depth-first-search algorithm that requires storing the state space in two stacks, so it is not scalable for many systems. Besides, Buchi automata is not designed to deal with finite execution traces, so using it in finite programs may achieve a wrong verification result. Even adding the self-loop to final states is not suitable in many cases because some program must be terminated.

A very important achievement of this work is that it managed to combine the symbolic execution with the verification procedure. Obviously, this ability allows finding out more counter-examples that maybe lost in concrete verification. Especially, the verification becomes totally automatic – the user does not need to manually create test drivers, just a right click.

References

[1] G. Holzmann. *SPIN model checker, the: primer and reference manual.* Addison-Wesley Professional, 2003.

[2] T. Balland S. K. Rajamani. The slam toolkit. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260-264, London, UK, 2001. Springer-Verlag.

[3] D. Distefano and M. J. Parkinson J. jStar: towards practical verification for Java. In *OOPSLA'08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 213-226, NewYork, NY, USA, 2008. ACM.

[4] Visser, W., Havelund, K., Brat, G., and Park, S. "Model Checking Programs", in *Proc. of the 15th IEEE International Conference on Automated Software Engineering (ASE'2000).* 11-15 September 2000, Grenoble, France. IEEE Computer Society, pp. 3-11. Y. Ledru, P. Alexander, and P. Flener, Eds.

[5] Havelund, K. and Pressburger, T., Model Checking Java Programs Using Java PathFinder. International Journal on Software Tools for Technology Transfer (STTT), Vol. 2(4), April 2000.

[6] Daniele, M., Giunchiglia, F., and Vardi, M.Y. "Improved Automata Generation for Linear Temporal Logic", in *Proc. of the 11th International Conference on Computer Aided Verification (CAV 1999)*. July 1999, Trento, Italy. Springer, LNCS 1633.

[7] Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In ASE 2001, pp. 412–416 (2001)

[8] CorinaS.Pˇ asˇ areanu, PeterC.Mehlitz, DavidH.Bushnell, KarenGundy-Burlet, MichaelLowry, SuzettePerson, MarkPape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages15–26, NewYork, NY, USA, 2008. ACM

[9] Gerth, R., Peled, D., Vardi, M.Y., and Wolper, P. "Simple On-the-fly Automatic Verification of Linear Temporal Logic", in Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95). June 1995, Warsaw, Poland, pp. 3-18.

[10] Somenzi, F. and Bloem, R. "Efficient Buechi automata from LTL Formulae", in *Proc. of the 12th International Conference on Computer Aided Verification (CAV*

2000). July 2000, Chicago, USA. Springer, LNCS 1855. E.A. Emerson and A.P. Sistla, Eds.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[12] R. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[13] J. C. King. Symbolic execution and programtesting. Communications of the ACM, 19(7): 385–394, 1976.