

Software Engineering Department

ORT Braude College

Course 61401: Extended Project in Software Engineering

Formal Verification of Specs of Applications

In Partial Fulfillment of the Requirements for

Final Project in Software Engineering (Course 61401)

Karmiel – June 2017

Saeed Namih 204582555

Ahmad Mnasra 311539647

Supervisor(s):

Dr. Elena Ravve

Dr. Katerina Korenbla

CONTENTS

[1. INTRODUCTION 3](#_Toc468621096)

[2. THEORY 3](#_Toc468621097)

[2.1. Background 3](#_Toc468621098)

[2.1.1. Formal verification 3](#_Toc468621099)

[2.1.2. Transition system (TS) 4](#_Toc468621100)

[2.1.3. Program graph (PG) 4](#_Toc468621101)

[2.1.4. Linear Temporal Logic(LTL) 5](#_Toc468621102)

[2.1.4.1 Syntax of LTL 5](#_Toc468621103)

[2.1.4.2 Semantics of LTL over Paths and States 5](#_Toc468621104)

[2.1.5. SPIN 6](#_Toc468621105)

## 1. INTRODUCTION

Every program development starts from its specification. Before, one starts implementation, the correctness of the spec must be confirmed. Specs of cellular applications demonstrate very specific character: transfer from one screen to another. We use the specialty of the specs to verify their correctness.

**What are we going to do?**

We will build a tool that allows graphical definition of specifications of cellular applications, that means Represent the specifications as a graph: nodes are the screens associated with the corresponding values of the parameters, edges are the event which motivate transitions.

Our application gets a list of Requirements that user wants to check, then it Uses machinery of formal verification to verify the spec, the verification result either Confirmation message or a path were the test Failed.

**Why is it not trivial?**

1. As we know existing verification methods ether executing code by searching about a wrong behavior or analyzing statically. The first method used only when the code has been written. Moreover, second method can’t detect subtle errors concurrency and algorithm defects. But our method performance verification at the design stage, i.e. before running code, we can detect errors.
2. Breakthrough; nobody thought about confirm correctness of Specs of cellular applications in all transition.

.

**What are the difficulties of the project?**

1. Our tool presents many screens.
2. We should find efficient structures to load and store a lot of nodes and parameters.
3. Building a workspace that allows user to build specifications graph.

## 2. THEORY

# 2.1. Background

It is all about money. We are annoyed when our mobile phone malfunctions, or when our video recorder reacts unexpectedly and wrongly to our issued commands. These software and hardware errors do not threaten our lives, but may have substantial financial consequences for the manufacturer.

### 2.1.1. Formal verification

Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics. This specification prescribes what the system should do and what not, and thus constitutes the basis for any verification activity.

The verification of these systems is done by providing a formal proof on an abstract mathematical model of the system, the correspondence between the mathematical model and the nature of the system being otherwise known by construction.

One approach and formation is model checking refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification. Typically, one has software systems in mind, whereas the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Model checking is a technique for automatically verifying correctness properties of finite-state systems.

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.

### 2.1.2. Transition system (TS)

Transition systems are often used in computer science as models to describe the behavior of systems. They are basically directed graphs where nodes represent *state* and edges model *transitions*, i.e., state changes. A state describes some information about a system at a certain moment of its behavior.

**Definition:**

A transition system TS is a tuple (S, Act,→,I, AP, L) where

* S is a set of states.
* Act is a set of actions,
* → ⊆ S × Act × S is a transition relation,
* I ⊆ S is a set of initial states,
* AP is a set of atomic propositions, and
* L: is a labeling function.

TS is called finite if *S*, *Act*, and *AP* are finite.

We can describe behavior of transition system as follows. The transition system starts in some initial state and evolves according to the transition relation →. That is, if s the current state, then a transition originating from *s* is selected *nondeterministically* and taken, the action *α* is performed and the transition system evolves from state *s* into the state *q*.

This selection procedure is repeated in state q and finishes once a state is encountered that has no outgoing transitions. It is important to realize that in case a state has more than one outgoing transition, the “next” transition is chosen in a purely nondeterministic fashion. That is, the outcome of this selection process is not known a priori, and, hence. Similarly, when the set of initial states consists of more than one state, the start state is selected *nondeterministically*.

The labeling function L relates a set L(s) ∈ of atomic propositions to any state *s. L(s)* intuitively stands for exactly those atomic propositions a ∈ AP which are satisfied by state *s*.

### 2.1.3. Program graph (PG)

program graphs over a set *Var* of typed variables. Essentially, this means that a standardized type (e.g., boolean, integer, or char) is associated with each variable. The type of variable *x* is called the domain *dom(x)* of x. Let *Eval(Var)* denote the set of (variable) evaluations that assign values to variables. *Cond(Var)* is the set of Boolean conditions over Var.

**Definition:**

A program graph PG over set Var of typed variables is a tuple (Loc, Act, Effect, →, , ) where

* Loc is a set of locations
* Act is a set of actions,
* Effect:Act×Eval(Var)→Eval(Var) →Eval(Var) is the effect function,
* →⊆Loc×Cond(Var)×Act×Loc↪⊆Loc×Cond(Var)×Act×Loc is the conditional transition relation,
* ⊆Loc is a set of initial locations,
* ∈Cond(Var) is the initial condition.

### 2.1.4. Linear Temporal Logic(LTL)

Linear temporal logic (LTL), is a logical formalism that is suited for specifying LT properties. LTL can be used to specify important system properties.

Temporal logic is a formalism par excellence for treating correctness depends on the executions. It extends propositional or predicate logic by modalities that permit to referral to the infinite behavior of a system.

The underlying nature of time in temporal logics is *linear*. i.e. at each moment in time there is a single successor moment, several model-checking tools use LTL as a property specification language. The model checker SPIN is a prominent example of such an automated verification tool.

### 2.1.4.1 Syntax of LTL

The basic ingredients of LTL-formulae are atomic propositions (state labels *a ∈ AP*), the Boolean connectors like conjunction , and negation *￢*, and two basic temporal modalities O(pronounced “next”) and U (pronounced “until”).

**The atomic proposition** *a ∈ AP* stands for the state label *a* in a transition system. Typically, the atoms are assertions about the values of control variables (e.g., locations in program graphs) or the values of program variables.

**The O -modality** is a unary prefix operator and requires a single LTL formula as argument. Formula holds at the current moment, if holds in the next “step”.

**The U-modality** is a binary infix operator and requires two LTL formulae as argument. Formula 1 U2 holds at the current moment, if there is some future moment for which 2 holds and 1 holds at all moments until that future moment.

**There are additional temporal operators are :**

“eventually” (eventually in the future)

“always” (now and forever in the future)

By combining the temporal modalities ◊ and □, new temporal modalities are obtained

♦“infinitely often ”

♦“eventually forever ”

### 2.1.4.2 Semantics of LTL over Paths and States

LTL formulae stand for properties of paths (or in fact their trace). This means that a path can either fulfill an LTL-formula or not. To precisely formulate when a path satisfies an LTL formula, we proceed as follows. First, the semantics of LTL formula is defined as a language that contains all infinite words over the alphabet that satisfy . That is, to every LTL formula a single LT property is associated. Then, the semantics is extended to an interpretation over paths and states of a transition system.

Let *TS* = (*S, Act,→, I,AP, L*) be a transition system without terminal states, and let

be an LTL-formula over *AP*.

*•* For infinite path fragment of , the satisfaction relation is defined by

*•* For state *s ∈ S*, the satisfaction relation *|*= is defined by

*• TS* satisfies , denoted *TS |*= , if *Traces*(*TS*) *⊆ Words*(*ϕ*).

From this definition, it immediately follows that

### 2.1.5. SPIN

Spin is a popular verification tool of distributed systems, used by thousands of people worldwide. The tool can be used for the formal verification of multi-threaded software applications.

Spin can perform simulations of the system's execution. It was developed at Bell Labs in the Unix group of the Computing Sciences Research Center, starting in 1980. Spin can perform interactive, guided, or random simulations of the system's execution.