

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. Avi Soffer

Mr. Alex Frid

CONTENTS

[1. INTRODUCTION 3](#_Toc468278399)

[2. THEORY 3](#_Toc468278400)

[2.1. Background 3](#_Toc468278401)

[2.1.1. Formal verification. **Error! Bookmark not defined.**](#_Toc468278402)

[2.1.2. Program graph (PG) **Error! Bookmark not defined.**](#_Toc468278403)

[2.1.3. Transition system(TS) 4](#_Toc468278404)

[2.1.4. Model checking 4](#_Toc468278405)

[2.1.5. Linear Temporal Logic(LTL) 5](#_Toc468278406)

[2.1.6. SPIN 5](#_Toc468278407)

## INTRODUCTION

Each 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 in order to use the machinery of formal verification in order to verify their correctness

**What are we going to do?**

We will build an tool that allows graphical definition of specifications of cellular applications, that's mean Represent the specifications as a graph: nodes are the screens associated with the corresponding values of the parameters.

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

**Why is it not trivial?**

1. Performance verification at the design stage, that mean before running code we can detect

errors.

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

1. our project has many screens
2. We should storage 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

system verification is used to establish that the design or product under consideration possesses certain properties. The properties to be validated can be quite elementary, e.g., a system should never be able to reach a situation in which no progress can be made (a deadlock scenario), and are mostly obtained from the system’s specification. This specification prescribes what the system has to do and what not, and thus constitutes the basis for any verification activity.

### 2.1.2. Program graph (PG)

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.

First of all, the program graph consisting of locations as nodes and conditional transitions as edges is **not** a transition system, since the edges are provided with conditions.

However, each program graph can be translated to a transition system. Particular, the underlying transition system of a program graph results from *unfolding*: a state of the transition system is composed of a location l of the program graph and an evaluation η of the variables.

### 2.1.3. Transition system (TS)

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, no statement can be made about Transition Systems 21 the likelihood with which a certain transition is selected. 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. 1 L(s) intuitively stands for exactly those atomic propositions a ∈ AP which are satisfied by state s.

### 2.1.4. Model checking

Model checking is a verification technique that explores all possible system states in a

brute-force manner. A model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property.

Even the subtle errors that remain undiscovered using emulation, testing and simulation can potentially be revealed using model checking.

Typical properties that can be checked using model checking are of a qualitative nature:

Is the generated result OK?, Can the system reach a deadlock situation,

Model checking requires a precise and unambiguous statement of the properties to be examined.

with making an accurate system model, this step often leads to the discovery of several ambiguities and inconsistencies in the informal documentation.

The model checker examines all relevant system states to check whether they satisfy the desired property. If a state is encountered that violates the property under consideration, the model checker provides a counterexample that indicates how the model could reach the undesired state.

The counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of a simulator, the user can replay the violating scenario, in this way obtaining useful debugging information, and adapt the model (or the property)

### 2.1.5. 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.

### 2.1.6. 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.