

Software Engineering Department

ORT Braude College

Course 61401: Extended Project in Software Engineering

**Formal Verification of Specs of Applications**

Authors

Saeed Namih ID :204582555

Ahmad Mnasra ID :311539647

Supervisors:

Dr. Katerina Korenblat

Dr. Elena Ravve

CONTENTS

[1. INTRODUCTION 3](#_Toc470446028)

[2. THEORY 3](#_Toc470446029)

[2.1. Background 3](#_Toc470446030)

[2.1.1. Formal verification 3](#_Toc470446031)

[2.1.2. Transition system (TS) 4](#_Toc470446032)

[2.1.3. Program graph (PG) 4](#_Toc470446033)

[2.1.4. Linear Temporal Logic(LTL) 5](#_Toc470446034)

[2.1.4.1 Definition Syntax of LTL 5](#_Toc470446035)

[2.1.4.2 Semantics of LTL over Paths and States 6](#_Toc470446036)

[2.1.5. SPIN 7](#_Toc470446037)

[*2.1.5.1 The PROMELA Language* 7](#_Toc470446038)

[ datatypes 7](#_Toc470446039)

[ Processes 8](#_Toc470446040)

[ Atomic Sequences 8](#_Toc470446041)

[ Control Flow 8](#_Toc470446042)

[Case Selection 8](#_Toc470446043)

[Repetition 9](#_Toc470446044)

[Unconditional Jumps 9](#_Toc470446045)

[ Conditional Expressions 10](#_Toc470446046)

[ Active Proctypes 10](#_Toc470446047)

[*2.1.5.2 LTL Syntax in spin* 10](#_Toc470446048)

[2.2. Detailed Description 17](#_Toc470446049)

[2.2.1 Re-producing the spec of "Bopo" 21](#_Toc470446052)

[2.2.1.1 Main activity 21](#_Toc470446053)

[1.2.1.1 Create New Event activity: 22](#_Toc470446054)

## 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 the 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 events, which motivate transitions.

Our application gets a list of Requirements that a user wants to check. Then it uses the machinery of formal verification to verify the spec. The verification results in either a confirmation message or a path where the test failed.

**Why is it not trivial?**

1. As we know, the existing verification methods verify either the **executing code** by searching about a wrong behavior or by analyzing statically. This is the first attempt to propose a method to check the spec when the corresponding code has not been written yet.
2. Breakthrough; nobody thought about the confirmation of the correctness of Specs of cellular applications

**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 the 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 the 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. 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 the state *s. L(s)* intuitively stands for exactly those atomic propositions a ∈ AP which are satisfied on the 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 Logical Temporal properties. LTL can be used to specify important system properties.

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

LTL formulae over the set AP of atomic proposition are formed according to the following gramma

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 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 next-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 Until-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 2 additional temporal operators:**

◊“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

|  |
| --- |
|  |
| *a* has to hold at the current state |
|  |
| *a*  has to hold at the next state |
|  |
| *a* has to hold until b , which holds at the current or a future position |
|  |
| *a*  eventually has to hold (somewhere on the subsequent path); |
|  |
| *a* has to hold on the entire subsequent path |

LTL formulae stands 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).*

### 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.

# *2.1.5.1 The PROMELA Language*

PROMELA is a verification modeling language. It provides a way for making models of distributed systems. The intended use of the Spin analyzer is to verify fractions of process behavior, that are considered suspect. A complete verification is therefore typically performed in a series of steps, with the construction of increasingly detailed PROMELA models. Each model can be verified with Spin under different types of assumptions about the environment. Once the correctness of a model has been established with Spin, that fact can be used in the construction and verification of all subsequent models.

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

## Datatypes

There are five predefined integer data types: bit , bool , byte , short , and int . (There are also constructors for user-defined data types, see [mtype(2)](http://people.cs.ksu.edu/~dwyer/SPINDOC/mtype.html), and [typedef(2)](http://people.cs.ksu.edu/~dwyer/SPINDOC/typedef.html), and a predefined data type for message passing channels, see [chan(2)](http://people.cs.ksu.edu/~dwyer/SPINDOC/chan.html).)

Variables of the predefined types can be declared in a C-like style, with a typename that is followed by a comma-separated list of one or more identifier names, each optionally followed by an initializer field. Each variable can also optionally be declared as an array, rather than as a scalar (for this see [arrays(2)](http://people.cs.ksu.edu/~dwyer/SPINDOC/arrays.html)).

The table below summarizes these definitions.

|  |  |  |  |
| --- | --- | --- | --- |
| **Typename** bit or bool byte short int | **C-equivalent** bit-field uchar short int | **Macro in limits.h** - CHAR\_BIT (width in bits) SHRT\_MIN..SHRT\_MAX INT\_MIN..INT\_MAX | **Typical Range** 0..1 0..255 -2^15 - 1 .. 2^15 - 1 -2^31 - 1 .. 2^31 – 1 |

The default initial value of a variable is zero.

If a value is assigned that lies outside the domain of the variable type, the true value assigned is obtained by truncation of the value to the domain (i.e., by a type cast operation).For instance:  
*byte a, b = 2; short c[3] = 3;*

## Processes

The state of a variable or of a message channel can only be changed or inspected by processes. The behavior of a process is defined by a *proctype* declaration. For example, the following declares a process type A with one variable *state*:

proctype A()

{

byte state;

state = 3;

}

The *proctype* definition only declares process behavior, it does not execute it. Initially, in the PROMELA model, just one process will be executed: a process of type *init*, that must be declared explicitly in every PROMELA specification.

New processes can be spawn using the run statement. It takes as argument the name of a process type and instantiate it. The run operator can be used in the body of the proctype definitions, not only in the initial process. This allows for dynamic creation of processes in PROMELA.

An executing process disappears when it terminates, that is, it reaches the end of the body in the proctype definition, but not before all processes that it started have terminated.

## Atomic Sequences

By prefixing a sequence of statements enclosed in curly braces with the keyword atomic the user can indicate that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. It is a runtime error if any statement, other that the first statement blocks in an atomic sequence. Atomic sequences can be an important tool in reducing the complexity of verification models. Note that atomic sequences restrict the amount of interleaving that is allowed in a distributed system. Intractable models can be made tractable by labeling all manipulations of local variables with atomic sequences.

## Control Flow

### Case Selection

The simplest construct is the selection structure. Using the relative values of two variables a and b, for example we can write:

*if*

*:: (a != b) -> option1*

*:: (a == b) -> option2*

*fi*

The selection structure contains two execution sequences, each preceded by a double colon. One one sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement of a control sequence is called a guard.

In the example above, the guards are mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected non-deterministically. If all guards are unexecutable the process will block until one of them can be selected.

There are two pseudo-statements that can used as guards: the timeout statement and the else statement. The timeout statement models a special condition that allows a process to abort the waiting for a condition that may never become true. The else statement can be used as the initial statement of the last option sequence in a selection or iteration statement. The else is only executable only if all other options in the same selection are not executable.

### Repetition

A logical extension of the selection stucture is the repetition structure. For example:

*do*

*:: count = count + 1*

*:: count = count - 1*

*:: (count == 0) -> break*

*od*

describes a repetition structure in PROMELA. Only one option can be selected at a time. After the option completes, the execution of the structure is repeated. The normal way to terminate the repetition structure is with a break statement. It transfers the control to the instruction that immediately follows the repetition structure.

### Unconditional Jumps

Another way to break a loop is the goto statement. For example, we can modify the example above as follows:

*do*

*:: count = count + 1*

*:: count = count - 1*

*:: (count == 0) -> goto done*

*od*

*done:*

*skip;*

The goto in this example jumps to a label named done. A label can only appear before a statement. If we might want to jump at the end of the program, for example, a dummy statement skip is useful: it is a place holder that is always executable and has no effect.

## Conditional Expressions

Conditional expressions analogous to the C-syntax *expr1 ? expr2 : expr3* are supported in Spin version 2. The syntax is however, different from C:

*(expr1 -> expr2 : expr3)*

The expression has the value of *expr2* when *expr1* evaluates to a non-zero value, and the value of *expr3* otherwise.

## Active Proctypes

In Spin version 2 there is a keyword active that can be prefixed to any proctype definition. If the keyword is present, an instance of that proctype will be active in the initial system state. Multiple instantiations of that proctype can be specified with an optional array suffix of the keyword. Example:

*active proctype A() { ... }*

*active [4] proctype B() { ... }*

# *2.1.5.2 LTL Syntax in spin*

Grammar:

*ltl ::=* ***opd*** *| ( ltl ) | ltl* ***binop*** *ltl |* ***unop*** *ltl*

Operands (**opd**):

true, false, user-defined names starting with a lower-case letter,

or embedded expressions inside curly braces, e.g.,: { a+b>n }.

Unary Operators (**unop**):

[] (the temporal operator *always*)

<> (the temporal operator *eventually*)

! (the boolean operator for *negation*)

Binary Operators (**binop**):

U (the temporal operator *strong until*)

W (the temporal operator *weak until*

V (the dual of U): (p V q) means !(!p U !q))

&& (the boolean operator for *logical and*)

|| (the boolean operator for *logical or*)

/\ (alternative form of &&)

\/ (alternative form of ||)

-> (the boolean operator for *logical implication*)

<-> (the boolean operator for *logical equivalence*)

The easiest way to specify an LTL property is to specify it inline. The formula is specified globally (i.e., outside all proctype or init declarations) with the following syntax:

*ltl [ name ] '{' formula '}'*

The name is optional, but can be useful when specifying multiple formulae. (Each such formula follows the same basic format.) The formula has the grammar outlined above, with some extensions. First, white space (newlines, spaces, tabs) can be used anywhere to separate operands and operators. Second, the names of operators can either be abbreviated with the symbols shown above, or spelled out in full (as always, eventually, until, implies, and equivalent. The alternative operators weakuntil, stronguntil, and release (for the V operator, see above), are also supported.   
This means that the following two are equivalent:

*ltl p1 { []<> p }*

*ltl p2 { always eventually p }*

The properties stated in this way are taken as positive properties that must be satisfied by the model. The model checker will perform an automatic negation of the formula to find counter-examples.

***2.1.5.3 BUILDING AND VERIFYING Spin***

***2.1.5.4 example***

The *PROMELA* represent a program graph of vending machine (fig 1.0)

|  |
| --- |
|  |
| ***Fig 1.0*** |

We added a condition to the PROMELA in order to verify The machine is refilled infinitely many times. We run verification in spin and the result show an error (fig1.1) because there is a path that Contradicts the requirement (fig1.2).

In order to fix this error we added a new condition to the PROMELA (fig 1.3)

|  |
| --- |
|  |
| fig1.1 |

|  |
| --- |
|  |
| fig1.2 |

|  |
| --- |
|  |
| fig1.2 |

|  |
| --- |
|  |
| fig1.3 |



### 2.2. Detailed Description

As it was mentioned above, we will build a tool that allows the graphical definition of specifications of cellular applications. To be more practical, we implement our tool using a real application for cellular phones, called “Bopo” Supervised by Dr. Elena Ravve. We take its spec in order to compose it in a visual form.

In order to add a new screen to the spec of an application, the user should press “add screen”, then she/he sets the screen location, defined the name and the description and then presses “save”.

The user should press on ***"+"*** button to choose an element type from the menu bar, this way we can represent the specs types of screens

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Fig. 1: | Fig. 1.1: | Fig. 1.2: |

the menu bar includes the following options:

* **On-Off** : this type  allows to activate or deactivate some features, in this  element type  we specified a field for name(the element name) and a list for default values.

**Front-end:**

|  |  |
| --- | --- |
|  |  |
| **New element design** | **New GUI design** |

|  |  |
| --- | --- |
| Go:(ack=off)  (ack=off)  {ack=on}  (ack=on)  {ack=off} | **#define max 5;**  **mtype = {Screen1,OFF,ON};**  **mtype ack=OFF;**  **mtype state= Screen1;**  **active proctype vm()**  **{**  **do**  **:: state== Screen1->**  **:: atomic{ ack =ON}**  **:: atomic{ ack =OFF}**  **od**  **}** |
|  |  |

|  |
| --- |
|  |
| Fig. 1: |

**Back-end:**

**If < Screen, defaultval=OFF >**

**< Screen, ON>**

**Else if < screen, ON>**

**<Screen, OFF>**

**Else if < screen, OFF>**

**<Screen, ON >**

* **List**: if the user knows the parameters she/he can add them as a list, so we specified a field for name, values and a list for default values.

|  |  |
| --- | --- |
|  |  |
| **New element design** | **New GUI design** |

|  |  |
| --- | --- |
| Go:(category=study)  {category =study}  {category =eat and drink}  {category =concerts}  {category =sports}  {{category =conventions} | mtype = {study, eat and drink, concerts, sports conventions, screen1};  mtype state=start;  mtype category=study;  active proctype vm()  {  do  : :state== study ->  ::atomic{ category =eat and drink}  ::atomic{ category =study}  ::atomic{ category = concerts }  ::atomic{ category = sports }  ::atomic{ category = conventions }  ::atomic{ category = study }  od  } |
| Fig 2: |  |

**Back-end:**

**< Screen, defaultval=val >**

**N=sizeOfLIst**

**If SelectedVal = 0**

**< Screen, 0>**

**If SelectedVal = 1**

**< Screen, 1>**

**If SelectedVal = 2**

**< Screen, 2>**

**If SelectedVal = 3**

**< Screen, 3>**

**..**

**..**

**If SelectedVal = N-1**

**< Screen, N-1>**

* **standard button** : this type is  used  to enable moving  from screen to another screen  ,in this type we  specified a field for  name ,default value, conditions ,and the next  screen.

|  |  |
| --- | --- |
|  |  |
| Fig 3: | Fig 3.1: |

|  |  |
| --- | --- |
|  | mtype = {screen1, screen2, cond1};  mtype state= screen1;  mtype cond=cond1;  active proctype vm()  {  do  :: state== screen1->  if :: cond==cond1 ->  atomic{ state== screen2}  fi  :: state== screen1->  if :: cond!=cond1 ->  atomic{ state== screen1}  fi  od  } |
|  |  |

* **defined/undefined:** some elements such as a Name, Location, can get a lot of varied values so we defined   this type.

We specified a field for name and a list for default values.

|  |  |
| --- | --- |
|  |  |
| Fig 4: |  |

|  |  |
| --- | --- |
| Go:(description=undefined)  {description= defined } | mtype = {screen1, defined, undefined};  mtype state= screen1;  mtype description = undefined;  active proctype vm()  {  do  : :state== screen1->  ::atomic{ description = defined }  od  } |
|  |  |

### Re-producing the spec of "BoPo"

### Main activity

*"...*

*Moderator + participant:*

*Description: The main screen of the application where the user can*

*choose what to do next (e.g. search, create new event, etc.).*

*Input: The user chooses a desired option.*

*Output: The user redirected to the suitable screen to her/his choice."*

|  |
| --- |
|  |
| Fig. 2: |

As it was mentioned above in the ***Main Activity*** spec of we should add three elements as follow:

* Create New Event - element type : standard button
* Show my events  - element type : standard button
* Notification - element type : standard button

1. The user press on ***"add screen"*** button to add three screens: Create New Event, Show my events and Notification.
2. The user adds an element  by pressing on "**+"** button and choosing a type of element  by pressing on  **“standard button” .** she/he defines the following:
   1. **name** of the button “Create New Event”.
   2. **move to** : "Create New Event" screen
3. user then adds the remaining elements (Show my events and Notification) in the same way.

### Create New Event activity:

*"Moderator + participant:*

*Description: The user creates new event. Upon creating the event the user*

*becomes the moderator of the event.*

*Input: Category, title, description, date and time, Ack is needed (yes/no),*

*more details (optional), maximum number of participants (optional), save*

*the event.*

*Output: If the user didn’t fill one or more of the mandatory fields, a pop*

*up message with a request to correct the suitable field(s) will appear.*

*Otherwise, if the maximum number of participants is less than one, an*

*error message will appear. Upon pressing the save button the user will be*

*directed to the Main screen.”*

the spec of screen ***Create New Event activity***: we should add these elements as follows:

* Category - element type: List
* title - element type: defined/undefined
* description- element type: defined/undefined
* date- element type: defined/undefined
* time- element type: defined/undefined
* Ack  - element type :On-Off
* save- element type:  standard button
* cancel- element type: standard button

*“Input: Category, title, description, date and time, Ack is needed (yes/no), more details (optional), maximum number of participants (optional), save the event.”*

1. The user adds a Category element by pressing on **+** button in  ***Create New Event* screen** and chooses a type of element  by pressing on “***List”*** she/he defines the following:
   1. ***name of element:*** Category
   2. ***values :***  the user defines the values that will be in this list such as : Study, eat and drink ,concerts, sports and convernation .
   3. **defaultVal:** the user chooses default value from this list such as ***“study”.***
2. The user adds elements title, description, date and time by pressing on "**+"** button in  ***Create New Event activity* screen** and chooses the type of element  by pressing on  **“defined/undefined”.** she/he defines for elements as following:
   1. ***Element name* :** the user enters a name such as a ***title*** .
   2. **defaultVal:** the user chooses a default value such as: ***undefined***.
3. The user adds “Ack” element by pressing on **"+"** button in Create ***New Event activity* screen** and choosing the type of element by pressing on  **“on/off”.** she/he defines for elements the following:
   1. **The name of element:** ACK .
   2. **defaultVal:** the user chooses the default value such as an “off”

*”...Upon pressing the save button the user will be directed to the* ***Main screen****.”*

1. The user adds a save element by pressing on **+** button in  ***Create New Event activity* screen** and choose type of element  by pressing on **“standard button” .** she/he defines the following:
   1. ***name of element :*** Save
   2. ***MoveTO:*** she/he ***chooses Main screen to***

*“If the user didn’t fill one or more of the mandatory fields, a pop up message with a request to correct the suitable field(s) will appear. Otherwise, if the maximum number of participants is less than one, an error message will appear.*

c. ***conditions***  in this option the user *insert the element maximum* *participant*, and add a condition that must be greater than 1 , and she/he marks the necessary field;

d. Save: