**   
Formal Verification of Specs of Applications**

Saeed Namih ID:204582555, Ahmad Mnasra ID:311539647

Supervisor(s): Dr. Katerina Korenblat, Dr. Elena Ravve

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?**

* This is the first attempt to propose a method to check the spec when the corresponding code has not been written yet.
* Breakthrough; nobody thought about the confirmation of the correctness of Specs of cellular applications
* None of the decisions that we took were trivial.
* The program graph that we built include all the conditions that the user defined, and for every condition a set of parameters, we have to check the behavior correctness

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

* Our tool presents many screens.
* We should find efficient structures to load and store a lot of nodes and parameters.
* Building a workspace that allows the user to build specifications graph.
* After the verification process finished we will get a text as a result that may be not clear to the user.
  1. **Organization of the paper**

In Section 2 we define basic concepts and related work. In Section 3 we give detailed description of workspace, and present options that user can do. We describe how to compose the SPEC in Front-end and how to translate it to Back-end (section 3.1). Than we present process of composing the spec by flowchart (Section 3.2), a way how we can re-produce the spec of BoPo (Section 3.3), a list of requirement that must be correct in every application (Section 3.4), and explain the verification process (Section 3.5) . In section 4 we give our expectations regarding the results. Finally, section 5 consists of preliminary Software Engineering documents: Requirements (Section 5.1), class diagram and initial GUI (Section 5.2), concluded by a short test plan section (Section 5.3).

1. **BACKGROUND AND RELATED WORK**

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 Formal Verification**

Formal verification (see, for example [5], [2]) 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 [3] 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, [1] 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.

**2.2 Transition System (TS)**

Transition systems [2] 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 where

* is a set of states.
* is a set of actions,
* is a transition relation,
* is a set of initial states.
* is a set of atomic propositions
* is a labeling function.

TS is called finite if and 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 is selected non-deterministically and taken, the action is performed and the transition system evolves from state *s* into the state q

This selection procedure is repeated in state 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 non-deterministically.

The labeling function relates set of atomic propositions to state *.*  intuitively stands for exactly those atomic propositions which are satisfied on the state .

**2.3 Program Graph (PG)**

Program graphs [2] are defined over a set 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 of . Let denote the set of (variable) evaluations that assign values to variables. *Cond(Var)* is the set of Boolean conditions over .

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

* is a set of locations,
* is a set of actions,
* is the effect function,
* is the conditional transition relation,
* *⊆* is a set of initial locations,

 is the initial condition.

**2.4 Linear Temporal Logic(LTL)**

*Linear Temporal logics (LTL)* [2] is a convenient formalism for specifying and verifying properties of reactive systems. We can say that the modalities in Temporal Logic are Time abstract

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.

***Syntax:*** LTL formulae over the set AP of atomic proposition are formed according to the following grammar

The basic ingredients of LTL-formulae[1] are atomic propositions (state labels *a ∈ AP*), the Boolean connectors like conjunction , and negation , and two basic temporal modalities (pronounced “next”) and (pronounced “until”).

* **The atomic proposition** checks that the given statement or assertion is true on a state. 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  holds at the current moment, if there is some future moment for which holds and 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 ”

**Semantics of LTL over Paths and States [2]:**

|  |
| --- |
|  |
| 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 |

### Fig. 1: Semantics of LTL (Principles of Model Checking by Christel Baier and Joost-Pieter Katoen)

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 *,* whichsatisfy . For 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 be a transition system without terminal states, and let be an LTL-formula over .

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

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

*• TS* satisfies , denoted

**2.5 SPIN**

SPIN [6],[4] is a popular verification for analyzing the logical consistency of asynchronous systems , used by thousands of people worldwide. It was developed at Bell Labs in the UNIX group of the Computing Sciences Research Center, starting in 1980.

The tool can be used for the formal verification of multi-threaded software applications. SPIN can perform simulations of the system's execution. SPIN can perform interactive, guided, or random simulations of the system's execution.

**2.5.1 The PROMELA language**

PROMELA [6] (Process or Protocol Meta Language) is a verification modeling language. The language allows for the dynamic creation of concurrent processes to model, for example, distributed systems. In PROMELA models, communication via message channels can be defined to be synchronous or asynchronous. PROMELA models can be analyzed with the SPIN model checker, to verify that the modeled system produces the desired behavior.

PROMELA programs consist of processes, message channels, and variables[4]. 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.

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

Variables of the predefined types can be declared in a C-like style, with a type name 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 these arrays).

The table below summarizes these definitions:

| Name | Size (bits) | Usage | Range |
| --- | --- | --- | --- |
| bit | 1 | unsigned | 0..1 |
| bool | 1 | unsigned | 0..1 |
| byte | 8 | unsigned | 0..255 |
| mtype | 8 | unsigned | 0..255 |
| short | 16 | signed | −215..215 − 1 |
| int | 32 | signed | –231..231 − 1 |
| *Table 1: Variables types and size* | | | |

\* 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.

***Atomicity***

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 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 un-executable the process will block until one of them can be selected.

There are two pseudo-statements that can be 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 structure 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() { ... }*

**2.5.2 LTL syntax in SPIN**

* Grammar:
* 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*)

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 [5] 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.

**2.6 Verification using SPIN**

**Example:** we performed a verification on a vending machine (VM)[5] model that was written in *PROMELA* language (Fig 2.0).

|  |
| --- |
|  |
| *Fig 2.0 :A representation of program graph and model of VM* |

We added LTL formula to the PROMELA in order to verify the property “The machine is refilled infinitely many times”. We run verification in SPIN and the result show an error (Figure 2.1) because there is a path that contradicts the condition (Figure 2.2).

|  |
| --- |
|  |
| *Fig 2.1 The verification result* |

|  |
| --- |
|  |
| *Fig 2.2 The path that contradicts the condition* |

In order to fix this error we added a new condition to the vm code (Fig 2.3)

|  |
| --- |
|  |
| *Fig 2.3 Second version of VM model and the verification result* |

The first VM model allows switching from start state to selection without checking the possibility that the machine is empty, thus the error was occurred, but in the second model we added a condition before the transfer , the condition was number of beer or soda is greater than 0.

1. **DETAILED DESCRIPTION**

**3.1 Workspace 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 implemented 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 (see Fig. 3.1) 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 press “save”.

The user should press on ***"+"*** button to choose an element type from the menu bar (see Fig. 3.2), this way we can represent the specs types of screens

|  |  |
| --- | --- |
|  |  |
| *Fig 3.1 Adding a new screen* | *Figure 3.2 Adding new element to the screen* |

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 the element name, a field for parameter name, an option for on/off and action button where we can select parameters that must be change if we select a default Value for the parameter .(Fig 3.3 and Fig 3.4)

|  |  |
| --- | --- |
|  |  |
| *Fig 3.3 “On/Off” Element* | *Fig 3.4 “On/Off” Action* |

Below (Fig. 3.5-3.6) one can see an example of the “on/off” element in BoPo application (for more details about BoPo spec see the book Appendix).

***Front-end:***

|  |  |
| --- | --- |
|  |  |
| *Fig 3.5 Adding an element called “ack” to “create New Event” screen .* | *Fig3.6 The “Create New Event “Screen after adding the “Ack”* |

***Back-end:***

In the Back-end we prepare the Program Graph: the state “Create New Event” is the screen Name in "BoPo",and the "ChangeAck" is a state for tracking parameter changes (see Fig 3.7-3.8).

|  |  |
| --- | --- |
| Go:(ack=off)  (ack=on)  {ack=off}  (ack=off)  {ack=on} | #define ON 1  #define OFF 0  mtype = {CreateNewEvent,ChangeAck};  bool ack=OFF;  mtype state=CreateNewEvent;  active proctype vm()  {  do  :: state==CreateNewEvent}->  if :: ack==OFF -> atomic{ack=ON; state=ChangeAck};state=CreateNewEvent  :: ack==ON -> atomic{ack=OFF; state=ChangeAck};state=CreateNewEvent  fi  od  } |
| *Fig 3.7 PG represent the element “On/Off-” in screen “Create New Event”* | *Fig 3.8 The (PG) in a PROMELA language* |

* **List**: if the user knows the parameters, she/he can add them as a list, so we specified a field for Element name, a field for parameter Name, values and a default value by choosing from the list, and action button where we can select parameters that must be change if we select a default Value for the parameter .(see Fig 3.9 and Fig 3.10)

|  |  |
| --- | --- |
|  |  |
| *Fig 3.9 ”List” Element* | *Fig 3.10 “List” Action* |

Below (Fig. 3.11-3.12) one can see an example of the “List” element in BoPo application (for more details about BoPo spec see the book Appendix)

***Front-end:***

User adds category element from type “List” to screen “Create New Event”.

He/she adds: names for the element and the parameter.

Parameters values in TextArea, each value in a separate line.

|  |  |
| --- | --- |
|  |  |
| *Fig 3.11 Adding an element called “Category” to “create New Event” screen .* | *Fig3.12 The “Create New Event “Screen after adding the “Category”* |

***Back-end:***

In the Back-end we prepare the Program Graph: the state “Create New Event” is the screen Name in "BoPo",and the " ChangCategory" is a state for tracking parameter changes(see Fig 3.13-3.14)..

|  |  |
| --- | --- |
| Go:(category=study)  {category =study}  {category =eat and drink}  {category =concerts} {category =sports}  {{category =conventions} | #define None 0  #define study 1  #define eatAndDrink 2  #define concerts 3  #define sports 4  #define conventions 5  mtype={ CreateNewEvent,ChangeCategory}  byte Categoty=None;  mtype state= CreateNewEvent;  active proctype vm()  {  do  : :state== CreateNewEvent ->  ::atomic{ category = study;state=ChangeCategory}; state=CreateNewEvent  ::atomic{ category = eatAndDrink;state=ChangeCategory}; state=CreateNewEvent  ::atomic{ category = concerts;state=ChangeCategory}; state=CreateNewEvent  ::atomic{ category = sports;state=ChangeCategory}; state=CreateNewEvent  ::atomic{ category = conventions;state=ChangeCategory}; state=CreateNewEvent  od  } |
| *Fig 3.13 PG represent the element “List” in screen “Create New Event”* | *Fig 3.14 The (PG) in a PROMELA language* |

* **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(see Fig 13) ,and the next screen(see Fig 13)

|  |  |
| --- | --- |
|  |  |
| *Fig 3.15 “Standard button” Element* | *Fig 3.16 “Standard button” Conditions* |

Below (Fig. 3.17-3.18) one can see an example of the “*Standard button*” element in BoPo application (for more details about BoPo spec see the book Appendix).

***Front-end:***

|  |  |
| --- | --- |
|  |  |
| *Fig 3.17 Adding an element called “Create New Event” to “BoPo-MainScreen” screen .* | *Fig3.18 The “BoPo-MainScreen “Screen after adding the* “*Standard button*” |

***Back-end:***Stander button press at Back-end:(see Fig 3.19-3.20)

|  |  |
| --- | --- |
|  | mtype = { BoPo-MainSreen, CreatNewEventScreen}  mtype state= BoPo-MainSreen ;  active proctype vm()  {  Do  if  :: state== BoPo-MainSreen ->  atomic{ state= CreatNewEventScreen }  :: state== CreatNewEventScreen ->  atomic{ state= BoPo-MainSreen }  fi  od  } |
| *Fig 3.19 PG represent the element* “*Standard button*” *in screen “Create New Event”* | *Fig 3.20 The (PG) in a PROMELA language* |

* **Empty/Not Empty**: some elements such as a Name, Location, can get a lot of varied values so we defined   this type.

We specified a field for element name, field for parameter Name and an option button for Empty or Not Empty (see Fig 3.21 and Fig 3.22).

|  |  |
| --- | --- |
|  |  |
| *Fig 3.21 “Empty/Not Empty ” Element* | *Fig 3.22 “Empty/Not Empty ” action* |

Below (Fig. 3.23-3.24) one can see an example of the “on/off” element in BoPo application (for more details about BoPo spec see the book Appendix).

***Frontend:***

|  |  |
| --- | --- |
|  |  |
| *Fig 3.23 Adding an element called “Description “to “create New Event” screen.* | *Fig3.24 The “Create New Event “Screen after adding the “description”* |

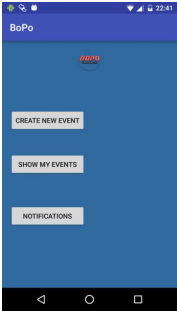
***Back-end:***

Stander button press at Back-end:(see Fig 3.25-3.26)

|  |  |
| --- | --- |
| Go:(description=Empty)  {description=NotEmpty} | #define *NotEmpty* 1  #define *Empty* 0  mtype = {CreateNewEvent,ChangeDescription};  byte description = *Empty*;  mtype state=CreateNewEvent;  active proctype vm()  {  do  :: state==CreateNewEvent}->  ::atomic{ description = *NotEmpty*;state= ChangDescription  }; state= CreateNewEvent  od  } |
| *Fig 3.25 PG represent the “Empty/Not Empty ” element in screen “Create New Event”* | *Fig 3.26 The (PG) in a PROMELA language* |

**3.2** Building spec- Summary:

|  |
| --- |
| C:\Users\hp\Downloads\lastflow2 (1).png |
| *Fig 4: Flowchart: the building process* |

**3.3 Re-producing the spec of “BoPo"**

### Main activity”[7]

*"...*

*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."*

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

|  |
| --- |
| *Fig. 4: "BoPo" -main Screen* |

* 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: Empty/Not Empty
* description- element type: Empty/Not Empty
* date- element type: Empty/Not Empty
* time- element type: Empty/Not Empty
* 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  **“Empty/Not Empty ”.** she/he defines for elements as following:
   1. ***Element name:*** the user enters a name such as a ***"description"*** .
   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(see Fig4.1) .
   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:

* 1. **Requirements in LTL**

In this section, we define a set of general requirements that must be true throughout the application run. In order to translate this requirement to LTL we need to define:

States = {,..,,}

Parameters name - ,values Type=” On/Off”

Parameters name - ,values Type=”Empty/NotEmpty”

Parameters name - ,values Type=” List”

Parameters Values={,,,, – parameters of specific list element }

AP a set of conditions such as (

1. There is always an exit from any screen.
2. Reachability
   1. There is a screen (root), such that each screen is reachable from it:

∨i⋀j

* 1. Each screen is reachable from *screeni*:

∧j

1. We cannot move from to without changing or defining a parameter.

1. Parameter cannot accept value that is not defined in the list of the possible values.
   1. Type "On/Off":

* 1. Type "Empty/NotEmpty":
  2. Type "List ":

1. There is no path to a screen that allows "Illegal parameters values".

(“Illegal” means value that does not defined in the list of the parameter values)

For each parameter Pij:

5.1 if Type("

 [ ⋀j⋀i)]

5.2 if Type("

 [ ⋀j⋀i)]

5.3 if Type("

 [ ⋀j⋀i)]

1. Each list of parameters must be defined before entering a screen

Given button for transition from if button contains subconditions

, the following LTL-formula has to be checked

1. Parameters values cannot be changed unless it was intended to do so in its path.

If a Parameter is changed in a specific state, the change should be updated wherever the parameter is used.

1. All parameters always must be consistent.

For an action { on element X the following property has to be checked :

⋀i (()

**Problems and difficulties in the project:**

1. At part 1 of the project we translated the requirements to ltl formula however we did not check with real spec , but When we Run Spin using BoPo Spec and the formula list , we face some problems :

* requirement 1 : “There is always an exit from any screen ”

this formula gives always true because it Refers to parameter Change as a transition to new state (Fig).

|  |
| --- |
| description== Empty))  {description=NotEmpty}  Go:(description=Empty) |
| Fig: changing parameter value let us move between the screen state and the change state. |

the new formula that we suggested is :

In the second formula we will not refer to the as a moving to new screen

* requirement 4 : “Parameter cannot accept value that is not defined in the list of the possible values."

In our tool we design the GUI So it does not allow parameters to accept values that is not defined in the list of the possible values, for that this requirement check if the translation promela is correct

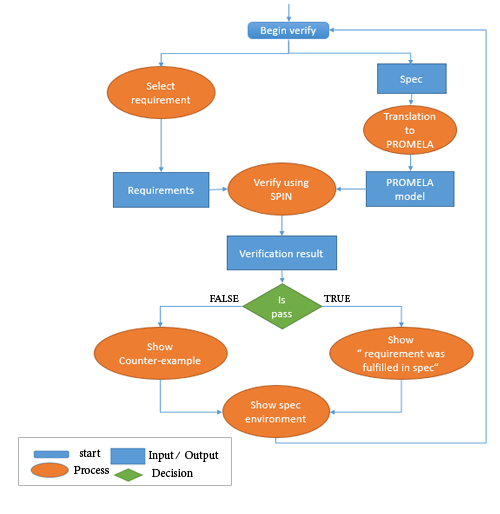
* requirements 7 : Parameters values cannot be changed unless it was intended to do so in its path.

This formula gives always true (fig), we think about new formula

1. Spin have many limitation for example the length of formula should not bee too long
2. We can’t match text field values with database for example: if the username and the password field NotEmpty ,we don’t know if they are correct or not correct, for that we allows the transition to the another screen without check the specific value in username and password fields .
3. we didn't check, if text field should have a specific value for example, number greater than 1, because our project concentrate to proof the concept, Although there are some holes in our tool but if there is continuous project it should handle these cases

We tried to build a model in order to proof a concept ,to add this figure it possible but should makes the project.

Requirement 8 :

* 1. **Verification process**

|  |
| --- |
|  |
| *Fig 5: Flowchart: the verification process* |

After the user built his spec he presses “Verify Spec”, a new window will pop, user chooses from this window a screen as a Root and a list of requirements that must be correct in his spec then he presses “Run” The verification process.

The verification process starts with translating the spec to Program Graph that will be written in PROMELA language, and save it in a file.

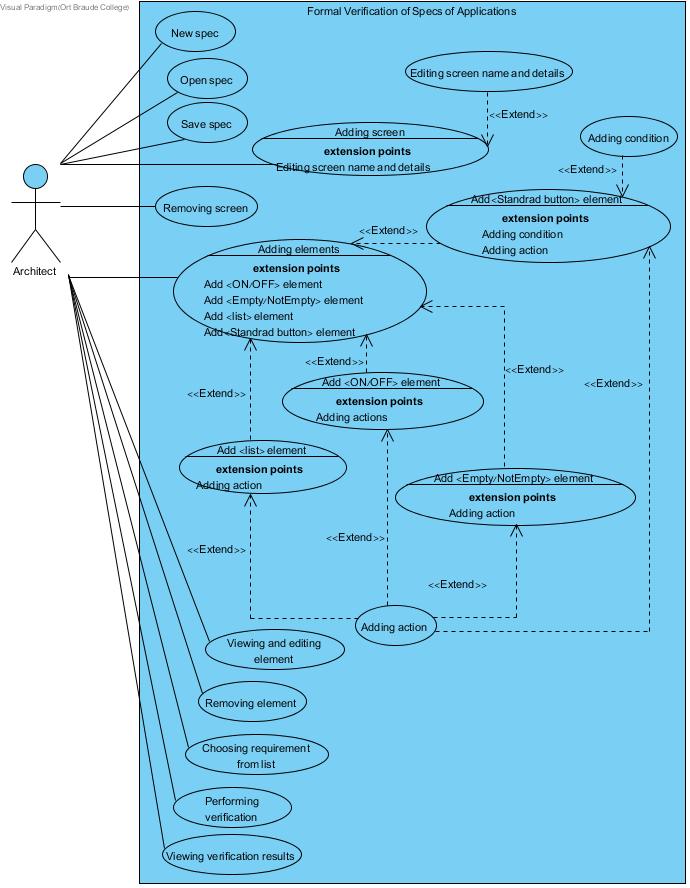
The tool creates a script file that run the PROMELA with every LTL requirement in SPIN and saves the verification results in a new file. Then, the user presses "Show Verification Result" to show the result.

1. **EXPECTED RESULTS**

We expect that our tool will help the system architect: now he can compose SPEC's graphically instead of writing in a traditional way by using word document. We will reproduce spec of real application called "BoPo".

The spec should be sound, and we build "sanity check" model to check consistency of the SPECS. We will try to find inconsistency in BoPo Spec.

The advantage of our tool is that it can check the spec without implementing the whole code of an application, that can reduce time and efforts of error detection

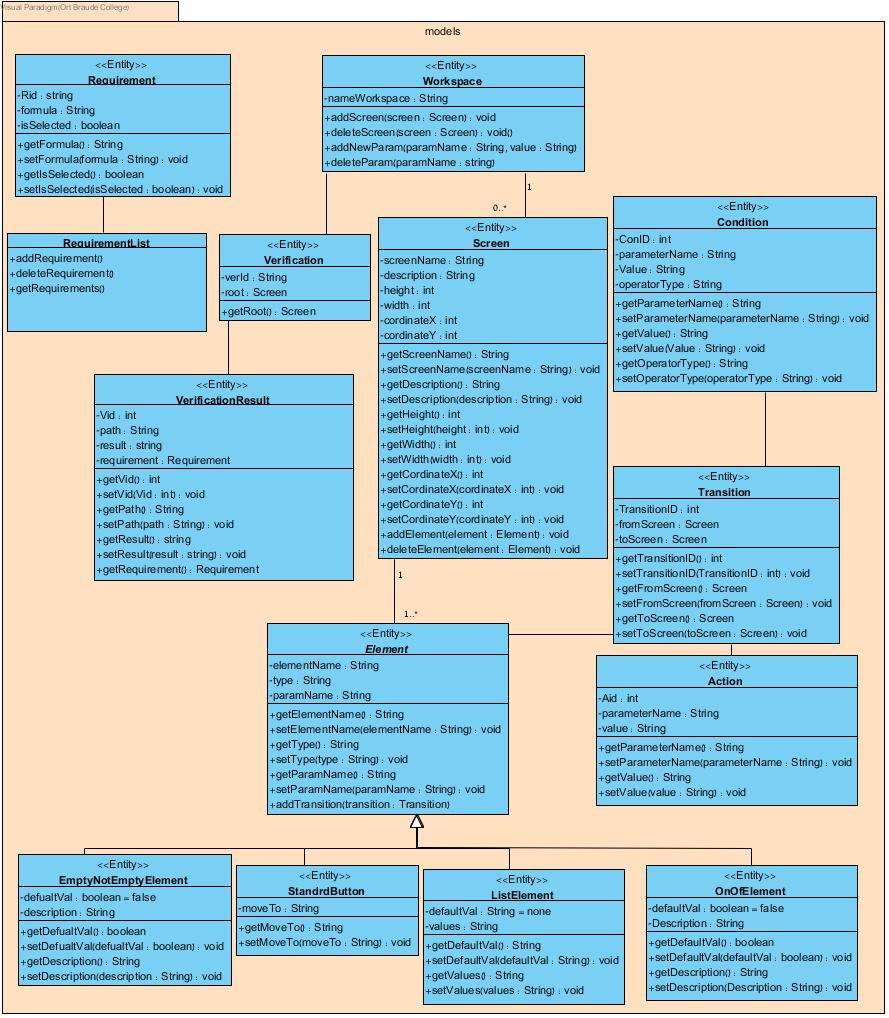
1. **PRELIMINARY SOFTWARE ENGINEERING DOCUMENTS**
   1. **Requirements (Use Case diagram)**

*Fig 6: Flowchart: the verification process*

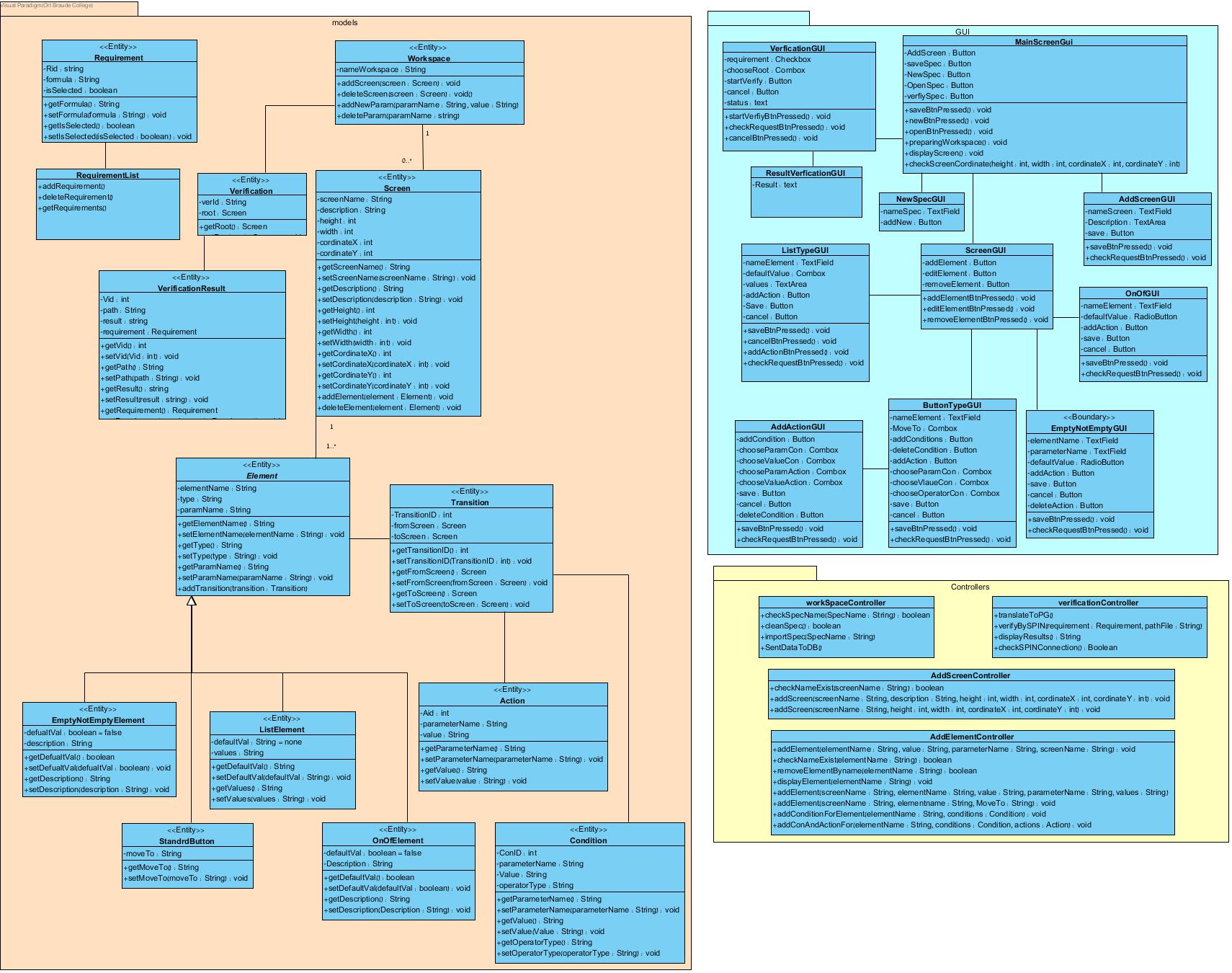
**5.2Design (GUI, UML Diagrams)**

**GUI Design:**

|  |  |
| --- | --- |
| C:\Users\Abu Nawaf\AppData\Local\Microsoft\Windows\INetCacheContent.Word\01mainscreenGUI.PNG |  |
| *Fig. 7.1: workspace for edit spec* | *Fig. 7.2: edit elements* |
| C:\Users\311539647\Downloads\finalproject-master\pic\02addscreen.png | *C:\Users\Abu Nawaf\AppData\Local\Microsoft\Windows\INetCacheContent.Word\01Workspace.png* |
| *Fig.7.3: add/edit screen to spec* | *Fig. 7.4: create new spec in tool* |
| *C:\Users\Abu Nawaf\AppData\Local\Microsoft\Windows\INetCacheContent.Word\04addelementONOFF.PNG* | *C:\Users\311539647\Downloads\finalproject-master\pic\EmptyNotEm;pty.png* |
| *Fig. 7.5: add/edit <ON/OFF> element* | *Fig.7. 6: add/edit <Empty/NotEmpty> element* |
| *C:\Users\Abu Nawaf\AppData\Local\Microsoft\Windows\INetCacheContent.Word\03addelementbutton.png* | *C:\Users\Abu Nawaf\AppData\Local\Microsoft\Windows\INetCacheContent.Word\04addelementList.png* |
| *Fig. 7.7: add/edit <standard button> element* | *Fig. 7.8: add/edit <List> element* |
| C:\Users\Abu Nawaf\AppData\Local\Microsoft\Windows\INetCacheContent.Word\04addactions.png | C:\Users\Abu Nawaf\AppData\Local\Microsoft\Windows\INetCacheContent.Word\06verfiysettingrequirment.png |
| *Fig.7. 9: add action for <ON/OFF> element* | *Fig. 7.10: Requirements list that can be verify* |
| *C:\Users\Abu Nawaf\AppData\Local\Microsoft\Windows\INetCacheContent.Word\07progressverfiy.png* | C:\Users\Abu Nawaf\AppData\Local\Microsoft\Windows\INetCacheContent.Word\08Resultverfiy.png |
| *Fig. 7.11: add/edit <standard button> element type* | *Fig. 7.12: show result verification of spec* |
|  |  |

**Class diagram**

*Fig. 8.1: Class Diagram*

****** *Fig. 8.2: Class Diagram*

1. **Testing plan**

In order to check out the system performance we will run the program on some significant input:

**6.1 Add and Edit Screen**

|  |  |  |  |
| --- | --- | --- | --- |
| **TestID** | **Initiating actor** | **Description** | **Expected results** |
| ScreenNameAlreadyExists | Architect | The architect chose a screen name that is already in use. | An error message will be shown: “screen name is occupied, please choose another”. |
| EmptyScreenName | Architect | The user did not enter screen name, and pressed “+” button. | An error message will  be shown: “A mandatory field; please enter title”. |
| InvalidEdited  ScreenNam | Architect | The user edited the screen name and entered name which contains signs and numbers | An error message will be shown:  “Screen name must contain only letters”. |

*Table 2: Testing plan: add and edit Screen*

**6.2** **Add and edit element (ON/OFF and Empty/NotEmpty Type)**

|  |  |  |  |
| --- | --- | --- | --- |
| TestID | Initiating actor | Description | Expected results |
| ElementNameAlreadyExists | Architect | The architect chose a element name that is already in use. | An error message will be shown: “element name is occupied, please choose another”. |
| ParametersName AlreadyExists | Architect | The architect chose a Parameter name that is already in use. | An error message will be shown: “Parameter Name is occupied, please choose another”. |
| Empty ElementName | Architect | The user didn’t enter Element Name, and pressed “add Element". | An error message will  be shown: “A mandatory  field, please enter title”. |
| Empty ParametersName | Architect | The user didn’t enter element name, and pressed “+” save. | An error message will be shown:  “Screen name must contain only letters”. |

*Table 3: Testing plan- add and edit element (ON/OFF and Empty / NotEmpty Type)*

**6.3** **Add and edit element (List Type)**

|  |  |  |  |
| --- | --- | --- | --- |
| TestID | Initiating actor | Description | Expected results |
| ElementNameAlreadyExists | Architect | The architect chose an element name that is already in use. | An error message will be shown: “element name is occupied, please choose another”. |
| ParametersName AlreadyExists | Architect | The architect chose a parameter name that is already in use. | An error message will be shown: “parameter name is occupied, please choose another”. |
| Empty ElementName | Architect | The user did not enter element name, and pressed “add Element". | An error message will be shown: “A mandatory field; please enter title”. |
| Empty ParameterName | Architect | The user did not enter element name, and pressed “+” save. | An error message will be shown:  “Screen name must contain only letters”. |
| Empty List | Architect | The user did not enter to the list and pressed " save". | An error message will be shown:  “List should Contain Values each line contains one value ”. |

*Table 4: Testing plan- add and edit element (List Type)*

**6.4** **Add and edit element (Button)**

|  |  |  |  |
| --- | --- | --- | --- |
| TestID | Initiating actor | Description | Expected results |
| ElementNameAlreadyExists | Architect | The architect chose a element name that is already in use. | An error message will be shown: “element name is occupied, please choose another”. |
| Empty ElementName | Architect | The user did not enter element name, and pressed “add Element". | An error message will  be shown: “A mandatory  field; please enter title”. |
| NoScreenChosen | Architect | The user did not choose screen, and pressed “add” button. | An error message will be shown: “please choose screen”. |

*Table 5: Testing plan- add and edit element (Button)*

**6.5 Add action**

|  |  |  |  |
| --- | --- | --- | --- |
| TestID | Initiating actor | Description | Expected results |
| ParameterValueNotSelected | Architect | The architect doesn't chose an specific value from the list | An error message will be shown: “please select the parameter value”. |
| ChoseParameterNotSelected | Architect | The architect doesn't chose an specific value from the list | An error message will be shown: “please select the parameter name”. |
| ParameterValueNotChoosed | Architect | The architect doesn't choose a value from the "change value" list | An error message will be shown: “choose a parameter from the list”. |

*Table 6: Testing plan- add action*

**6.6 Add/edit Spec**

|  |  |  |  |
| --- | --- | --- | --- |
| TestID | Initiating actor | Description | Expected results |
| OpenSpecErorr | Architect | The architect cannot open SPEC File. | An error message will be shown: “cannot open this file”. |
| SaveSpecErorr | Architect | The architect cannot save SPEC File. | An error message will be shown: “cannot save this file”. |
| SpecNameAlreadyExists | Architect | The architect chose a SPEC name that is already in use. | An error message will be shown: “SPEC name is occupied, please choose another”. |

*Table 7: Testing plan- spec verification*

**6.7 verify Spec**

|  |  |  |  |
| --- | --- | --- | --- |
| TestID | Initiating actor | Description | Expected results |
| RequirmenttNotSelected | architect | The architect does not select the requirements | An error message will be shown: “Please, select the requirements”. |
| RootNotSelected | architect | The architect does not chose an specific screen from Screen list | An error message will be shown: “Please, select the root”. |
| SPINFail | architect | The architect Press “Run” But SPIN is not installed | An error message will be shown: “Cannot run SPIN; an error occurred”. |
| CreatePromelaFileFaild | architect | The architect Presses “Run” and SPIN gets damaged file | An error message will be shown: “Cannot run SPIN; an error occurred while Creating the File ”. |
| CreateCounterExampleField | architect | The architect Presses “Show Results” and the Results File not found | An error message will be shown: “cannot show counter Example”. |
| OutOfMemmory | architect | The architect adds too screens | An error message will be shown: “SPIN is out of memory”. |

*Table 8: Testing plan- spec verification*

1. **REFERENCES**

[1] Ahmed Amine Jerraya, Wayne Wolf, Katalin Popovici, Frédéric Rousseau, *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, second edition, Springer Science+Business Media ,2010

[2] Christel Baier, Joost-Pieter Katoen ,*Principles of Model Checking*, MIT Press,2008

[3] Doron A. Peled ,*Software Reliability Methods*, 2001

[4] Gerard J. Holzmann ,*The Model Checker SPIN* , second edition, IEEE, 2003

[5] Mordechai Ben-Ari, *Principles of the Spin Model Checker*, second edition, Ohmsha, 2008

[6] SPIN Site-all about spin, "*http://spinroot.com/spin/whatispin.html*" ,2016

[7] Yelena Ladizhensky ,Ofir Rosner ,*BoPo* ,2016

**Appendix[7]:**

Specification of BoPo application.

In the application, there exist three types of users:

The first one is the moderator, who can open new event(s) and manage them.

The second one is the participant, who can join to an existing event(s).

The third one is the administrators, who maintain the users and the events.

* + 1. **Person:**

1. **Login activity:**   
   Description: Login into the application.

Input: User name and password.  
Output: If the person exists, logging into the application. Otherwise, if the user does not exist in DB, an error message will pop up. Otherwise, if username does not match the password, an error message will pop up.

* + 1. **User:**

1. **Register activity:**

***Moderator + participant:***   
Description: The user can register with username and password.   
Input: Name, username, password, date of birth, email, phone number (optional), picture (optional), gender (optional).  
Output: If the username does not exist on DB, the user will be added to the DB. Otherwise, if the username already exists, a pop up message with request to choose another user name will appear.

1. **Register with Facebook activity** **(External App)  
   *Moderator + participant:***Description: The user can register with her/his Facebook username and password.  
   Input: Facebook username and password.  
   Output: If the user exists on Facebook DB, he/she will be directed to the main screen; otherwise, if the user does not exist in Facebook DB, pop up message with an error will appear.

1. **Register with Google+ activity** **(External App)  
   *Moderator + participant:***Description: The user can register with her/his Google+ username and password.  
   Input: Google+ username and password.  
   Output: If the user exists on Google+ DB, he/she will be directed to the main screen; otherwise, if the user does not exist on Google+ DB, pop up message with an error will appear.
2. **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.

1. **Profile activity:**

***Moderator + participant:***   
Description: The user can view or modify her/his personal info.  
Input: The user can choose a desired attribute(s) to change.  
Output: Upon clicking the "save" button, if the entered attribute(s) are legal, the new data will be saved to the DB, and a success message will pop up. Otherwise, if one of the attribute(s) is illegal, a pop up message with a request to correct the suitable field(s) will appear.

1. **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.

1. **Choose Location activity:**

***Moderator + participant:***  
Description: The user can choose location in two ways:

1. By typing a place name.

b) By clicking on a desired location from the map.  
Input: A place name.   
Output: The desired location.

1. **My events** **activity:**

***Moderator + Participant:***  
Description: The user can view and search in her/his events in the list according to her/his role:

1. The moderator can view the events that were created by him.
2. The participant can view the events that he/she was already joined to, or those are waiting for ACK.
3. Both moderator and participant can search for an event.

Input:

1. The user chooses an event.
2. The user can search for an event.

Output:

1. The user will be directed to her/his chosen event.
2. The user views the event search results.
3. **Give a Ride activity:**

***Moderator + Participant:***

Description: The user can offer himself as a driver who can give a ride.

Input: The user inputs the following: starting location, maximum passengers (default: 3) and participants fee (default: 0).

Output: The DB will be updated with the relevant data, and a success message will pop up.

1. **Get a Ride activity:**

***Moderator + Participant:***

Description: The user can ask a ride.

Input: The user inputs the following: starting location and participants' fee (default: 0).

Output: The DB will be updated with the relevant data, and a success message will pop up. If a suitable ride is found then the user will get a notification with the relevant data.

1. **Event activity:  
   *Moderator:***Description: The moderator can see the event he/she created, edit and cancel it. He/She also can approve/disapprove new participants if Ack was required.  
   Input: Select an option:
2. Edit the event.
3. Cancel the event.
4. Approve/Reject user if Ack was required.
5. Preview the location of the event.

Output:

1. Edit the event: Upon entering the relevant new information. If all the fields were filled correctly, the DB will be updated accordingly. All the participants of this event will receive a push notification about the changes. Otherwise, an error message will appear.
2. Cancel the event: If the moderator pressed "confirm,"the event will be removed from the list of the participants' events, as well as from the available events. The relevant data will be sent to the DB. All the participants of this event will receive a push notification. Otherwise, if the moderator pressed "cancel,"he will be redirected back to the event screen.
3. If the user’s request was accepted, he/she will be added to the event, and push notification will be sent to him.
4. The user is directed to a screen, where he/she can see the full maps with all the relevant details.

***Participant:***Description: the participant can view the event details, and can ask to join it.  
Input: select an option:

1. The participant can ask to view list of all participants who are associated with the event.
2. The participant can ask to join to the specified event.

Output:

1. The user is directed to the People screen.
2. If ACK was needed, the user can ask to join the event. Otherwise, if ACK was not needed, the participant simply joins the event.
3. **People activity:  
   *Moderator + participant:***Description: In this screen the user can view the list of users who are associated with a specific event, as well as basic info about the event(Place, date and the time of the event).  
   Input:
4. The user may press a specific user to view her/his full profile.
5. The user may search for a specific user.

Output:

1. The user view the people search results.
2. The user is directed the Person screen.
3. **Person activity:  
   *Moderator + participant:***Description: The user may view specific user info, and call them if they entered a phone number.  
   Input: The user may press call.  
   Output: The user is directed to the Calling screen.
4. **Settings activity:**

***Moderator + participant:***  
Description: The user can manage her/his account settings (e.g. change password, turn on/off notifications Etc.).  
Input: the user can choose a desired attribute(s) to change.  
Output:

1. Delete account: if the user pressed "confirm "he/she will be redirected to the login screen. Otherwise, if he/she pressed “cancel" he/she will be redirected back to the settings screen.
2. Change password: if the user pressed “change password”, he/she will be directed to the change password screen.

1. **Change Password activity:   
   *Moderator + participant:***Description: The user can change her/his password.  
   Input: Old password, new password, confirm new password.

Output: If the old password does not match the one in the DB, an error message will pop up. Otherwise, if the new password does not match the confirmed new password, then an error message will pop up. Otherwise, the new password will be updated in the DB, and a success message will appear.

1. **Search activity:**

***Moderator + participant:***   
Description: The user can search for events with pre-defined filters (optional).  
Input: Search string and after the search results are shown he/she can choose one of the events in the list.  
Output: After choosing an event the user will be directed to the Event screen. If there are results from the DB, they will be shown below the search bar. Otherwise, if there are no results, a message will notify the user that no results found for her/his search.

1. **Request To Create A New Category activity:**

***Moderator + participant:***   
Description: The user may ask the app manager to open a new category with a proposed name.

Input: Category name.  
Output: The app manager may accept or reject the request. The user is notified in any case. If accepted, the name of the category will be added to the list of available categories.

1. **Notification activity:**Description: This screen will present user's notifications:
2. ***Moderator:***
3. A user joined her/his event.
4. A user wants to join her/his event (if ACK was required).
5. ***Participant***
6. An event was cancelled.
7. An event was updated.
8. A request to join an event was approved/declined.
9. ***User***
10. The request to add new category was approved/declined.

Input: the user can choose the notification he/she want to observe.  
Output: the notification will be transferred to the user.

1. **About Us activity:  
   *Moderator + participant:***Description: The user can view the creators of the application.  
   Input: None.  
   Output: None.
2. **Calling activity: (External App)  
   *Moderator + participant:***Description: In this screen the user will do the actual phone call to a specific user.  
   Input: Standard Android phone call screen inputs (e.g. mute the conversation, change the volume, etc.).  
   Output: Standard Android phone call screen output (e.g. conversation muted, the volume has changed, etc.).
   * 1. **Administrator:**
        1. **Manage App activity:**Description: The main screen of the application where the administrator can choose what to do next: add new category or check pending category requests.  
           Input*:* The administrator chooses a desired option.  
           Output: The administrator redirected to the suitable screen to her/his choice.
        2. **Add New Category activity:**Description: The administrator may add a new category name by her/his choice.  
           Input: Category name.  
           Output: If the category is already in the DB, an error message will pop up. Otherwise, the category will be added to the DB.
        3. **Pending Category Request activity:**Description: Any user may ask the app administrator to open a new category with a proposed name. The app administrator may accept or reject the request.  
           Input: The administrator chooses a pending request and approves or rejects it.   
           Output: The request is deleted from the list and the user is notified in any case. If accepted, the name of the category is joined to the list of available categories.

**Screen Diagram[7]:**

