**Formal Verification of Specs of Applications**

Saeed Namih ID:204582555, Ahmad Mnasra ID:311539647

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

**Abstract.** *In our project we*

**Keywords**:

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
3. All the decision that we took were not trivial for example the parameters in every spec may be connect to other parameters in another spec for that if we changes parameter in one screen we should change search about all the parameters that should change.
4. The program graph that we build include all the conditions that the user defined and for every condition an set of parameters we have to check the behavior correctness

**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.
4. We have to save many permeates that connect to many specs
   1. **Organization of the paper**
5. **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 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.

**2.2 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 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, and
* 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 nondeterministically 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 nondeterministically.

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

**2.3 Program Graph (PG)**

Program graphs which 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.

**2.3 Linear Temporal Logic(LTL)**

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

*Linear temporal property* (LT properties) is a temporal logic formula that describes a set of infinite sequences for which it is true.

*LTL* suited for specifying LT 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.

***Syntax:*** 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 (pronounced “next”) and (pronounced “until”).

* **The atomic proposition** stands for the state label in 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  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:**

|  |
| --- |
|  |
| 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

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 . 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

**2.3 Spin**

Spin is a popular verification tool of distributed 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.

**2.3.1 The PROMELA language**

PROMELA (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. 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 this 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 |

\* 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 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() { ... }*

**2.3.2 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 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.

**2.4 Building and Verifying Spin**

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

|  |
| --- |
|  |
| Figure 2.0 : representation a 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).

|  |
| --- |
|  |
| Figure 2.1 The verification result |

|  |
| --- |
|  |
| Figure 2.2 the path that contradicts the condition |

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

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

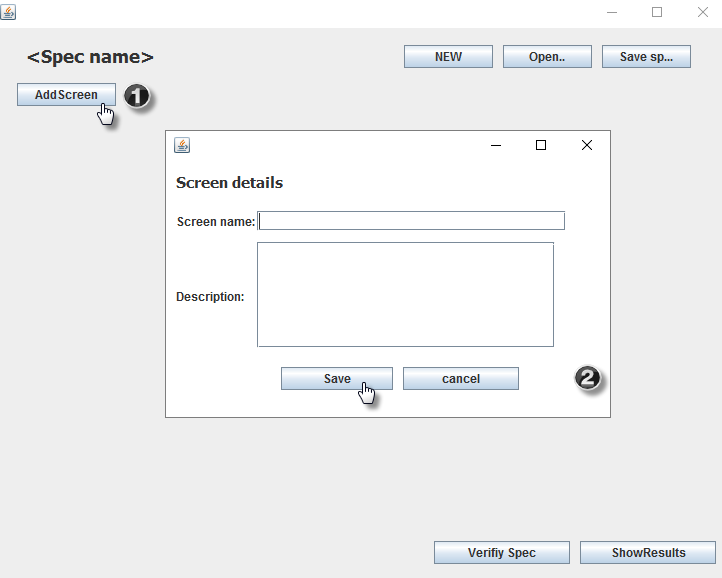
The first vm model allows switching from start state to selection without checking the possibility that the machine is empty, thus the error was occurred, but in the second model we added a condition before the transfer , the condition was number of beer or soda is greater than 0.

1. **DETAILED DESCRIPTION**

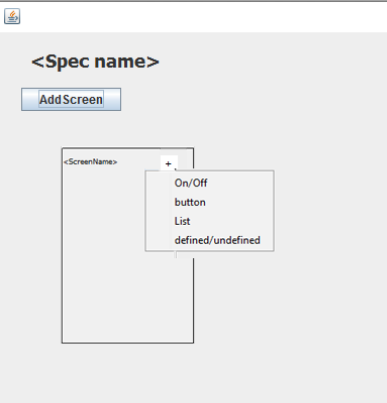
As it was mentioned above, we will build a tool that allows the graphical definition of specifications of cellular applications. To be more practical, we implement our tool using a real application for cellular phones, called “Bopo” Supervised by Dr. Elena Ravve. We take its spec in order to compose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

  
Figure 3.1 Adding a new screen

Backend :

  
Figure 3.2 adding new spec 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 name(the element name), a field for parameter name, and an option button.

**Front-end:**

**User add ack element from type ON/OFF to screen that called Create New Event**

|  |  |
| --- | --- |
|  |  |
| *Fig 4.1 Adding an element called ack to “create New Event” screen .* |  |

**Back-end:**

**The state in program grah take the name of the screen “Create New Event”**

**The parameter ack tack on or off value**

|  |  |
| --- | --- |
| Go:(ack=off)  (ack=off)  {ack=on}  (ack=on)  {ack=off} | **mytype{OFF,ON};**  **mtype = {Screen1}**  **mtype ack=OFF;**  **mtype state= Screen1;**  **active proctype vm()**  **{**  **do**  **:: state== Screen1->**  **:: atomic{ ack =ON}**  **:: atomic{ ack =OFF}**  **od**  **}** |
| *Fig 4.2 PG represent the element ON-OFF in screen “Create New Event”* | *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 name, a field for parameter Name ,values and a default value by choosing from the list.

**User add category element from type List to screen “Create New Event”. Then he entering a parameter name the parameter value can adding values to TextArea by entering value then a new line after each value**

|  |  |
| --- | --- |
|  |  |
| *Fig 4.3 Adding an element called Category to “create New Event” screen .* |  |

|  |  |
| --- | --- |
| Go:(category=study)  {category =study}  {category =eat and drink}  {category =concerts}  {category =sports}  {{category =conventions} | mtype={ CreateNewEvent}  mtype = {undefined, study, eat and drink, concerts, sports conventions};  mtype state= CreateNewEvent;  mtype category=study;  active proctype vm()  {  do  : :state== CreateNewEvent ->  ::atomic{ category =eat and drink}  ::atomic{ category =study}  ::atomic{ category = concerts }  ::atomic{ category = sports }  ::atomic{ category = conventions }  od  } |
| *Fig 4.2 PG represent the element List in screen “Create New Event”* |  |

* **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 = { Bopo-MainSreen  , CreatNewEvent  , cond1};  mtype state= Bopo-MainSreen ;  mtype cond=cond1;  active proctype vm()  {  do  :: state== screen1->  if :: cond==cond1 ->  atomic{ state== screen2}  fi  odod  } |
|  |  |

* **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.

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

|  |  |
| --- | --- |
| Go:(description=undefined)  {description= defined } | Mtype={Screen1}  mtype = {defined, undefined};  mtype state= screen1;  mtype description = undefined;  active proctype vm()  {  do  : :state== screen1->  ::atomic{ description = defined }  od  } |
|  |  |

.

1. **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:

1. **Requirement LTL:**

*<<Introduction for LTL requirement> >*

* There is Always an exit from any screen.
* There is a screen (root), such that each screen is reached from it.
* We can't move from to without changing or defining a parameter.
* Parameter cannot accept value that is not defined in the List of the possible values.
* There is no path to a screen that allows  "Illegal parameters values".

(Illegal i.e. value that does not defined in the list of the parameters values)

* Each list of parameters must be defined before entering a screen.
* Parameters values cannot change unless it was intended to do so in its path.
* If a Parameter changes  in a specific state the change should be updated wherever the parameter is used.
* All parameters always must be consistent.

1. Expected Results

we expect that the tool will help the system architect , now he can enter the spec in a visual from by using our tool instead of writing a document for the spec .

the advantage of out tools that he can check the spec without implement the whole code of an application, what can Reduced time and efforts of error detection

The architect add all the elements that will appear in a real application and he defined conditions and actions for every element after that he choose from the requirement list a set of requirement . after that he run verifying in order to check if the spec design Fulfilling the requirements , and if an error detected the verification result will be shown with the path were the error occurred

In order to prove that our tool is needed we take a spec of real application called Bopo that was written in a traditional way by using word document ,

**4.4. Conceptual UI Design:**

**4.4.1. Prototype Screens:**

**UML**

1. **Testing plan**

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

**5.1 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 didn’t entered Screen Name, and pressed “+” button. | An error message will  be shown: “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 1: Testing plan- add and edit Screen*

**5.2** **Add and edit element (ON/OFF and Defined / Undefined 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. |  |
| Empty ElementName | architect | The user didn’t entered Element Name , and pressed “add Element". | An error message will  be shown: “Mandatory  field, please enter title”. |
| Empty ParametersName | architect | The user didn’t entered Element Name , and pressed “+” save. | An error message will be shown:  “Screen Name must contain only letters”. |

*Table 2: Testing plan- add and edit element (ON/OFF and Defined / Undefined Type)*

**5.3** **Add and edit element (List 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. |  |
| Empty ElementName | architect | The user didn’t entered Element Name , and pressed “add Element". | An error message will  be shown: “Mandatory  field, please enter title”. |
| Empty ParameterName | architect | The user didn’t entered Element Name , and pressed “+” save. | An error message will be shown:  “Screen Name must contain only letters”. |
| Empty List | architect | The user didn’t entered to the list and pressed " save". | An error message will be shown:  “ List Should Contain Values each line contain one value ”. |

*Table 3: Testing plan- add and edit element (List Type)*

**5.4** **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 didn’t entered Element Name , and pressed “add Element". | An error message will  be shown: “Mandatory  field, please enter title”. |
| **NoScreenChosen** | architect | The user didn’t chose Screen, and pressed “add” button. | An error message will be shown: “please choose Screen”. |

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

**5.5 Add action**

|  |  |  |  |
| --- | --- | --- | --- |
| TestID | Initiating actor | Description | Expected results |
| ParameterValueNotSelected | architect | The architect does not chose an specific value from the list | An error message will be shown: “please select parameter value”. |
| ChoseParameterNotSelected | architect | The architect does not chose an specific value from the list | An error message will be shown: “please select parameter Name”. |
| ParameterValueNotChoosed | architect | The architect dos not choose a value from the "change Value" list | An error message will  be shown: “choose a parameter from the list ”. |

*Table 5: Testing plan- add action*

1. **references**

http://www.cs.colostate.edu/~france/CS614/Slides/Ch5-Summary.pdf