ParanoYa

Lóránt Boráros, Filip Budáč, Martin Cehelský, Silvia Holecová

October 2019

1 Analysis

- **The current status of the application**

  Application named ParanoYa is used for statistical testing pseudone random sequences. In this application, are implemented various test sets like NIST, FIPS, Diehard. With this application it is also possible to evaluate each testing sequence. Using the application it is also possible to evaluate individual tested sequences based on two methodologies. Output of the application is processed in Microsoft Excel document. With this document we can evaluate the achieved results. Application was created with frameworku Qt and used test suites are implemented in C.

- **Methods used in evaluate tests**

  porovnanie s odporucanymi/ interval odporucanych (pouzivnie pre ucely kryptografie na predmete) diehard test sa nedaju pouzit 10 MB

- **Used test sets**

  Testing is a process when is executed one or more test cases based on specified conditions. During this process is compared current and expected behavior. In the application are implemented different sets of tests, for example NIST, FIPS a Diehard.

  1. **NIST**

     NIST is statistical package of tests which is used to testing randomness of arbitrarily long binary sequences. This sequences are generated using a random or pseudo-random sequence generator. This package is consists of the following 15 test:

     (a) **The Frequency (Monobit) Test**
     The aim of this test is determine whether the ratio of zeros and units in a given sequence corresponds to the expected ratio for a random sequence. The number of units and zeros in the sequence should be approximately equal, which is also examined by the test.

(b) **Frequency Test within a Block**
This test considers the ratio of zeros and units in M-bit blocks. The aim of the test is to determine whether frequency of M-bit block is approximately M/2.

(c) **The Runs Test**
In this test is important the total number of zeros and units in runs in whole sequence, where the run represents a continuous sequence of equal bits. A run of length k means that it consists of k identical bits and is bounded before and after with a bit having the opposite value. The purpose of this test is to determine whether the number of runs of units and zeros of varying length is as expected for random sequences. This test is mainly used to assess whether the variation between such substrings is too slow or too fast.

(d) **Tests for the Longest-Run-of-Ones in a Block**
This test focuses on the longest run units within M-bit blocks. Its purpose is to determine whether the length of the longest run units in the test sequence is consistent with the length of the longest run units expected in random sequences. Irregularity in the expected length of the longest run of units means that there exists an irregularity in the expected length of the longest run of zeroes. Long zero runs are not evaluated separately because of concerns about statistical independence between tests.

(e) **The Binary Matrix Rank Test**
The test is aimed at the discontinuous order of the submatrices in the whole sequence. The purpose of this test is checking the linear dependence in the fixed length of the substrings of the original sequence.

(f) **The Discrete Fourier Transform (Spectral) Test**
The focus of this test are the heights of the peaks in the Fourier transform. The purpose of this test is detect periodic functions (for example, repeating patterns that are close together) in a test sequence that would indicate a deviation from the assumption of randomness.

(g) **The Non-overlapping Template Matching Test**
The random number sequence is divided into independent substrings of length M and the number of occurrences of template B, which represents the m-bit run units in each of the substrings. IfP-value chi-square of statistic is less than the significance level, the test concludes that the test sequence appears random. Otherwise, the test concludes that the retest appears to be random. The throughput is defined by the ratio of the sequences that passed the test.

(h) **The Overlapping Template Matching Test**
This test detects the number of occurrences in pre-specified tar-

get strings. The test uses an m-bit window to search for a specific m-bit pattern. If the pattern is not found, the window moves about one bit position. If the searched pattern is found, the window moves only one bit before resuming the search.

(i) **Maurer's "Universal Statistical" Test**
The purpose of this test is determine whether the sequence can be significantly compressed without losing information or not. A too compressed sequence is considered as non-random.

(j) **The Linear Complexity Test**
The purpose of this test is determine whether the sequence is sufficiently complex to be considered as random.

(k) **The Serial Test**
The purpose of this test is determine whether the number of occurrences of overlapping m-bit patterns is approximately the same as would be expected in a random sequence.

(l) **The Approximate Entropy Test**
The test focuses on the frequency of any possible overlap of m-bit patterns in the whole sequence. The purpose of this test is to compare the frequency of the overlapping blocks of two consecutive or adjacent lengths (m and m + 1) with the expected result for a random sequence.

(m) **The Cumulative Sums (Cusums) Test**
This test focuses on the maximum deviation (from zero) of the random walk (defined by the cumulative sum of the adjusted (-1, +1) digits in sequence). The aim of the test is determine whether the cumulative sum of the partial sequences occurring in the test sequence is too large or too small relative to the expected behavior of this cumulative sum for the random sequences. This cumulative sum can be considered as a random walk. The random walk deviation should be near zero for a random sequence. For certain types of random sequences, the deviations of this random walk will be greater than zero.

(n) **The Random Excursions Test**
The test is focused on the number of cycles that have exactly K occurrences in the cumulative sum of random steps. The cumulative sum can be found if the subtotals (0, 1) of the sequence are adjusted to (-1, +1). The random deviation of the random steps consists of a sequence of n steps of unit length. The purpose of the test is determine whether the number of occurrences of the state with random-step exceeds what is expected of the random sequence.

(o) **The Random Excursions Variant Test**
This test examines how many times is occurred specific status in a cumulative sum of random steps. The goal is detect deviations from the expected number of occurrences of different states in

random steps.

These tests deal with the different types of randomness that might arise in sequence. Some of the tests could be broken down into different subtests. The order in which the tests are run is arbitrary, but it is recommended that the Frequency test be run first, because if this test fails, the probability of failing further tests is very high.

2. **FIPS** nist sp-822,fips 140-2 Test Federal Information Processing is the US government security standard used to validate cryptographic modules. FIPS provides different types of security based on a defined level of security. There are four such levels:
   (a) **Level 1** - the lowest security level that does not require specific physical security mechanisms but requires the use of at least one approved security algorithm or function
   (b) **Level 2** - this level requires role-based access control, as well as physical security
   (c) **Level 3** - in this level is provided identity-based authentication and physical security. It should include an attack detection mechanism. If the system were hacked, the system should be able to delete critical security parameters
   (d) **Level 4** - it is the highest level of security. In addition to the above-mentioned requirements for the system, the requirements of physical security are tighten, it is especially advantageous for working in a physically unprotected environment

   FIPS validation involves intensive testing to identify specific deficiencies and weaknesses. For the system to meet FIPS validation, it needs to include cryptographic algorithms and hash functions. The three best known examples are AES, Triple DES, and HMAC SHA-1.

3. **Diehard**
   Diehard tests are statistical tests used to evaluate the quality of the random number generator. The Diehard test battery consists of various, independent statistical tests. The results of these assays are referred to as p-values. Diehard's tests include:
   (a) **The Birthday spacings test**
   This test first selects m birthdays in a year with n days, then it is a list of birthday gaps between birthdays. Finally, the Poisson asymptotically distribution of j value is assessed. The j value is the number of values that are in the list of spaces. If it is multiple times in the list, then j is asymptotically Poisson divided with diameter $m^3/(4n)$. n must be large enough to compare the results with the Poisson distribution.
   (b) **Overlapping permutations**

This test follows a sequence of one million 32-bit random integers. Each set of five consecutive integers can be in one of 120 states for 5! possible arrangement of five numbers.

(c) **Ranks of matrices**
This test is performed by selecting a number of bits from a number of random numbers to form a matrix above [0,1] and then is determining the matrix order. The number of rows should follow a certain distribution.

(d) **Monkey test**
Also called as bitstream test. This test has its name from an endless "monkey theorem". It is best achieved by processing sequences of a certain number of bits as "words" and counting the overlapping words in the steam. The number of "words" that do not appear should follow the known distribution.

(e) **Count the 1s**
The test is done through counting the 1 bits in each of either successive or chosen bytes and converting the counts to "letters", and counting the occurrences of five-letter "words".

(f) **Parking lot test**
Randomly place unit circles in a 100 x 100 square. If the circle overlaps an existing one, try again. After 12,000 tries, the number of successfully "packed" circles should follow a certain normal distribution.

(g) **Minimum distance test**
Randomly place 8000 points in a 10,000 x 10,000 square and then find the minimum distance between the pairs. The square of this distance should be exponentially distributed with a certain mean.

(h) **Random spheres test**
Randomly choose 4000 points in a cube of edge 1000. Center a sphere on each point, whose radius is the minimum distance to another point. The smallest sphere s volume should be exponentially distributed with certain mean.

(i) **The squeeze test**
Multiply 231 by float random integers on [0,1) until you reach 1. Repeat this 100,000 times. The number of floats needed to reach 1 should follow a certain distribution.

(j) **Overlapping sums test**
Generate a long sequence of random floats on [0,1). Add sequence of 100 consecutive floats. The sums should be normally distributed with characteristic mean and sigma.

(k) **Runs test**
Generate a long sequence of random floats on [0,1). Count ascending and descending runs. The counts should follow a certain distribution.

(l) **The craps test**
Play 200,000 games of craps, counting the wins and the number of throws per game. Each count should follow a certain distribution.

1.1 Solution methods

The core of the application, test sets, are represented as C libraries. User interface is created using Qt framework and application output is currently presented within an *.xls* file, readable in spreadsheet editors. The solution design is divided into several steps:

1. **Project actualisation compatible with current design environments** We decided to use Python for developing user iterface. Using Cython library we created an python-c interface. A shared object file "*.so*" was created from the C libraries. The shared object enables us to dynamically connect library with different programs.

1.2 Functional requirements

- **Statistic pseudorandom sequence testing -** user will be able to statistically test pseudorandom sequences using implemented test sets
- **Adjusting of tests -** user will be able to adjust and edit tests by their criteria
- **Evaluation of testing sequences -** after sequence testing, user will be able to view test evaluation based on selected methodology

2 Existing programs

1. **Ent** je konzolová aplikácia, ktorá slúži na vyhodnocovanie pseudnáhodných postupností pre štatistické vzorkovacie aplikácie, šifrovacie a kompresné algoritmy. Ent vykonáva nasledujúce testy:

   - **Entropia**
   - **Chi-square Test**
   - **Arithmetic mean**
   - **Monte Carlo Value of Pi**
   - **Serial Correlational Coefficient**

   Ent poskyutje viacerých možností na modifikovanie spracovania údajov ako aj formát výstupu:

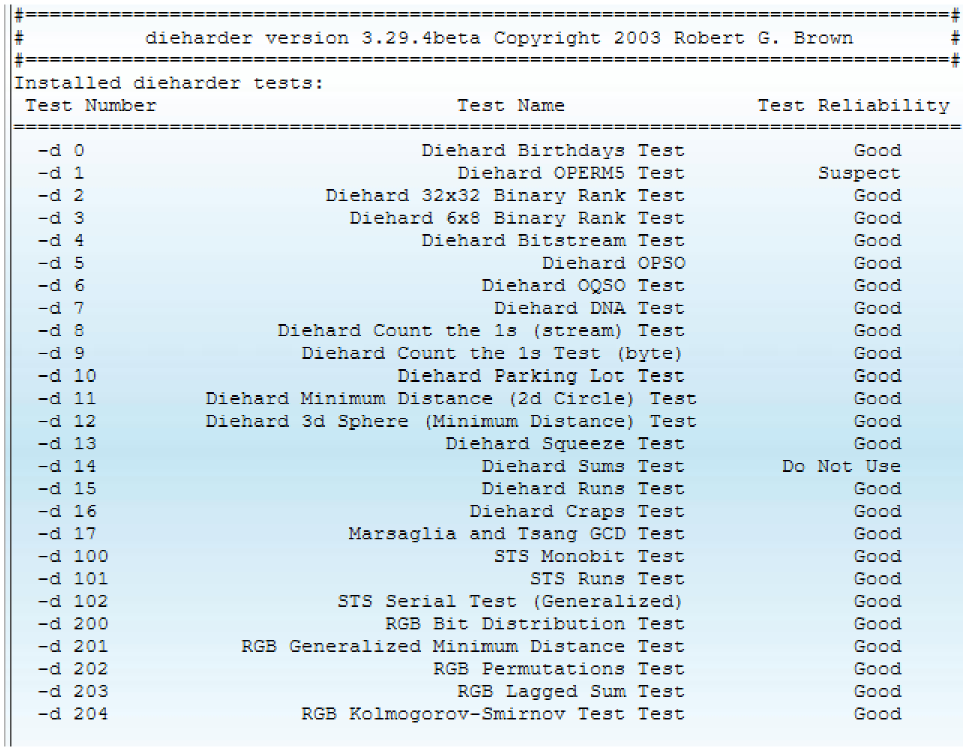
   - **-b**
     Vstupné dáta s spracované ako bity namiesto bajtov.

- **-c**
  Na štandarný výstup vytlačí tabuľku vyskytnutých znakov, ktoré sú zobrazené s ich desatinnými bajtovými hodnotami spolu s prislúchajúcim znakom zo sady ISO 8859-1 Latin-1.
- **-f**
  Veľké písmená sú zmenené na malé pred vykonaním testov.
- **-t**
  Výstup je vo forme tzv. *terse mode*, v ktorom výstupy sú oddelené čiarkou(CSV formát).
- **-u**
  Výpis informácií

2. **Dieharder** je vylepšená verzia programu *Diehard battery of tests* s vyčisteným zdrojovým kódom, implementované v jazyku C. Najdôležitejšie vylepšenia sú jeho rýchlosť a vďaka architektúry rozšíriteľnosť sady testov. Je open source projektom, program je voľne dostupný a stiahnuteľný na jeho webstránke. Dieharder umožňuje otestovanie generátorov priamo, vstupom môže byť aj neobmedzený tok náhodných čísel.

3. **Practically random** je knižnica implementované v programovacom jazyku C++ a slúži na testovanie generátorov náhodných postupností-*RNG*

4. **TestU01** je knižnica implementované v programovacom jazyku ANSI C. Obsahuje funkcie na empirické štatistické testovanie generátorov náhodných postupností. Aplikácia



```
#=============================================================================#
#          dieharder version 3.29.4beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
Installed dieharder tests:
 Test Number                         Test Name                Test Reliability
===============================================================================
  -d 0                            Diehard Birthdays Test              Good
  -d 1                               Diehard OPERM5 Test           Suspect
  -d 2                    Diehard 32x32 Binary Rank Test              Good
  -d 3                      Diehard 6x8 Binary Rank Test              Good
  -d 4                            Diehard Bitstream Test              Good
  -d 5                                      Diehard OPSO              Good
  -d 6                                 Diehard OQSO Test              Good
  -d 7                                  Diehard DNA Test              Good
  -d 8                Diehard Count the 1s (stream) Test              Good
  -d 9                  Diehard Count the 1s Test (byte)              Good
  -d 10                         Diehard Parking Lot Test              Good
  -d 11         Diehard Minimum Distance (2d Circle) Test             Good
  -d 12         Diehard 3d Sphere (Minimum Distance) Test             Good
  -d 13                            Diehard Squeeze Test               Good
  -d 14                               Diehard Sums Test         Do Not Use
  -d 15                               Diehard Runs Test               Good
  -d 16                              Diehard Craps Test               Good
  -d 17                    Marsaglia and Tsang GCD Test               Good
  -d 100                               STS Monobit Test               Good
  -d 101                                  STS Runs Test               Good
  -d 102                 STS Serial Test (Generalized)                Good
  -d 200                     RGB Bit Distribution Test                Good
  -d 201         RGB Generalized Minimum Distance Test                Good
  -d 202                         RGB Permutations Test                Good
  -d 203                           RGB Lagged Sum Test                Good
  -d 204              RGB Kolmogorov-Smirnov Test Test                Good
```

Figure 1: Dieharder test suite

3 GUI - Graphical user interface

3.1 Development environment Figma

3.2 Opening window

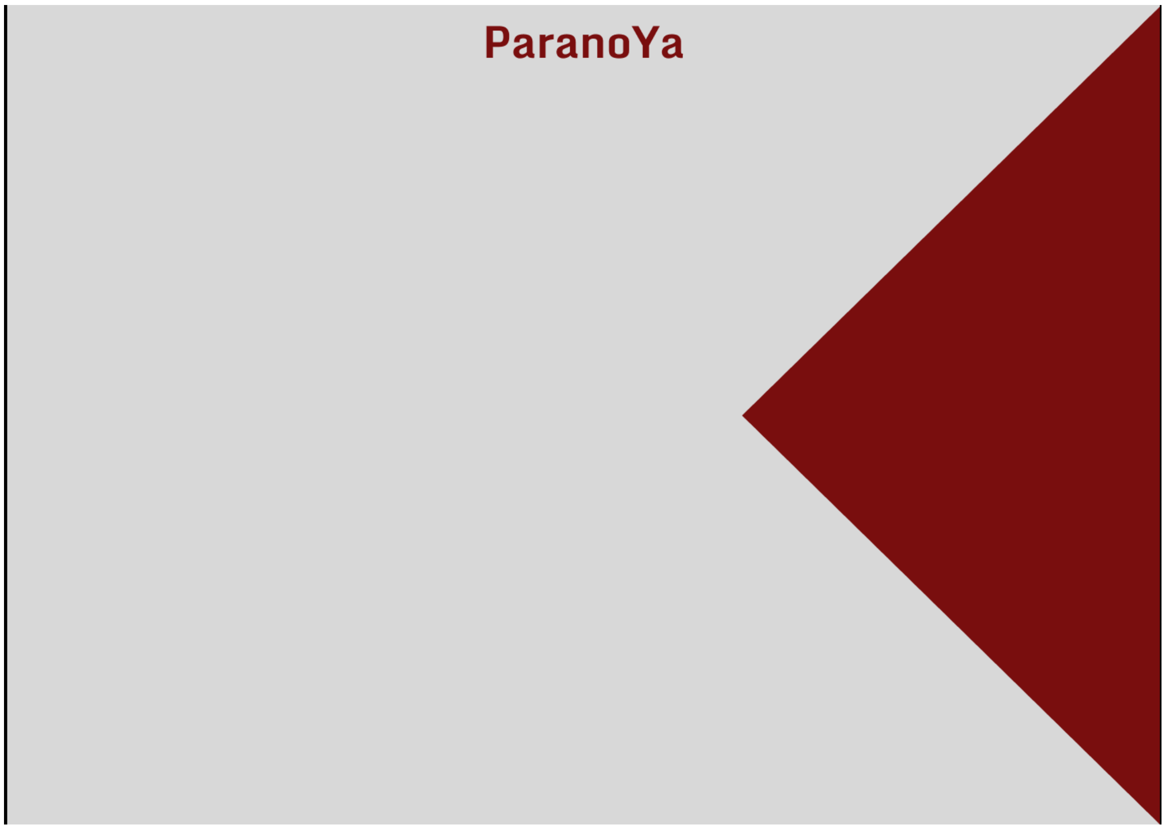


Figure 2: Opening window

Purpose:	Po spustení aplikácie užívateľ je uvítaný týmto oknom.
Navigation and User Interaction:	Po inicializácií aplikácia bude pripravený na používanie, program automaticky presmeruje na nasledujúce okno.

3.3 Main menu

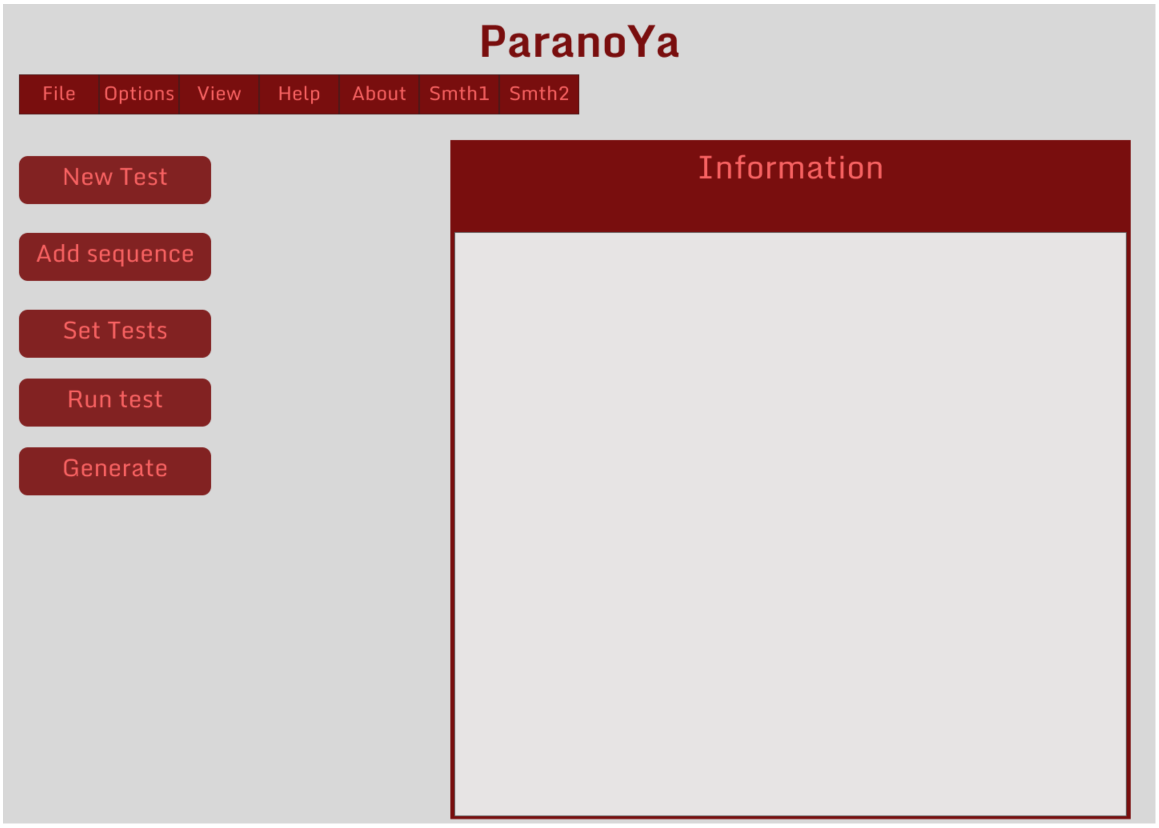


Figure 3: Main menu

Purpose:	Z tohto okna sú dostupné všetky funkcionality programu: • **Nastavenie testov** • **Pridanie postupností** • **Generovanie postupností** • **Spustenie testov**
Navigation and User Interaction:	Užívateĺ tlačítkom zvolí akciu, ktorú chce vykonať. Aplikácia naviguje na príslušné okno.

3.4 New Test

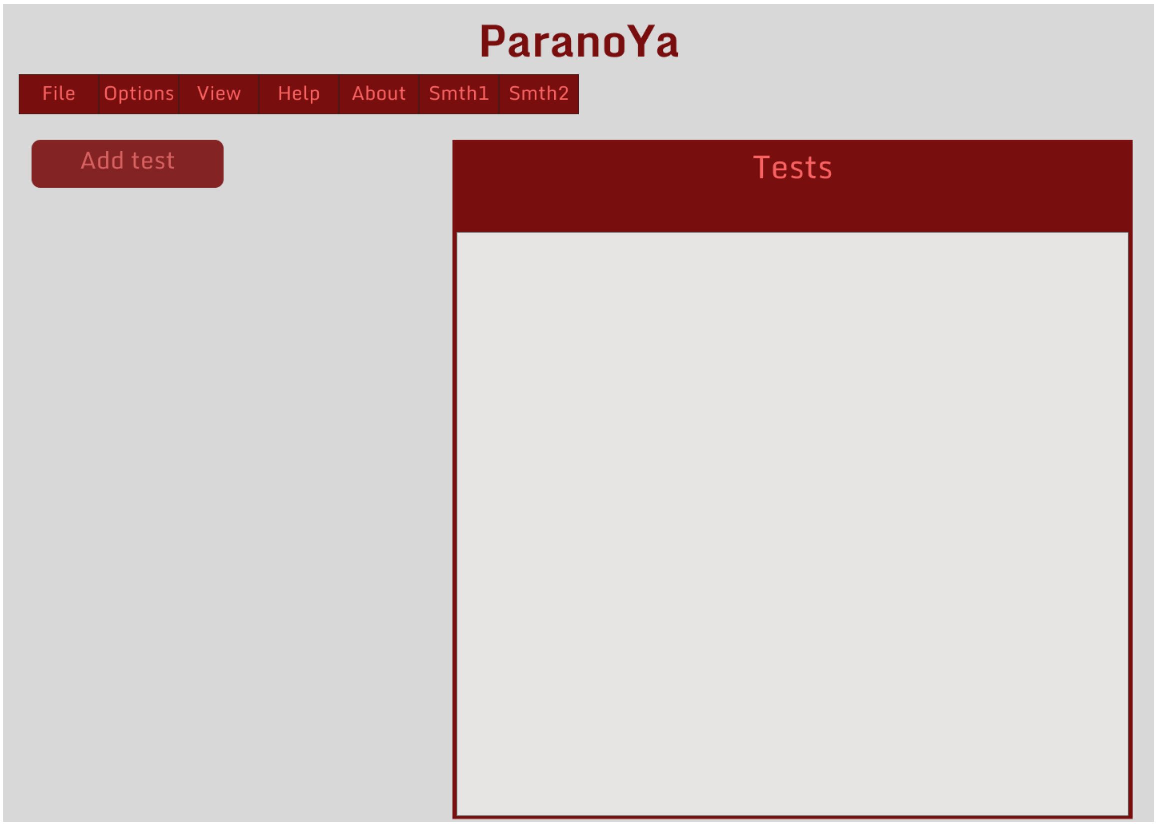


Figure 4: New Test

Purpose:	Aplikácia umožní užívateľovi zostrojiť novú sadu testov a uložiť ich, alebo načítať a použiť už existujúce sady testov
Navigation and User Interaction:	Užívateľ s tlačítkom **Add** môže pridať testy, ktorých vyberá zo zoznamu všetkých testov. Nastavenie parametrov jednotlivých testov sa robí vo vzláštnom okne. S tlačítkom **Save** je možné uložiť zostrojenú sadu testov. S tlačítkom **Load** je možné načítať vopred vytvorených sád testov.

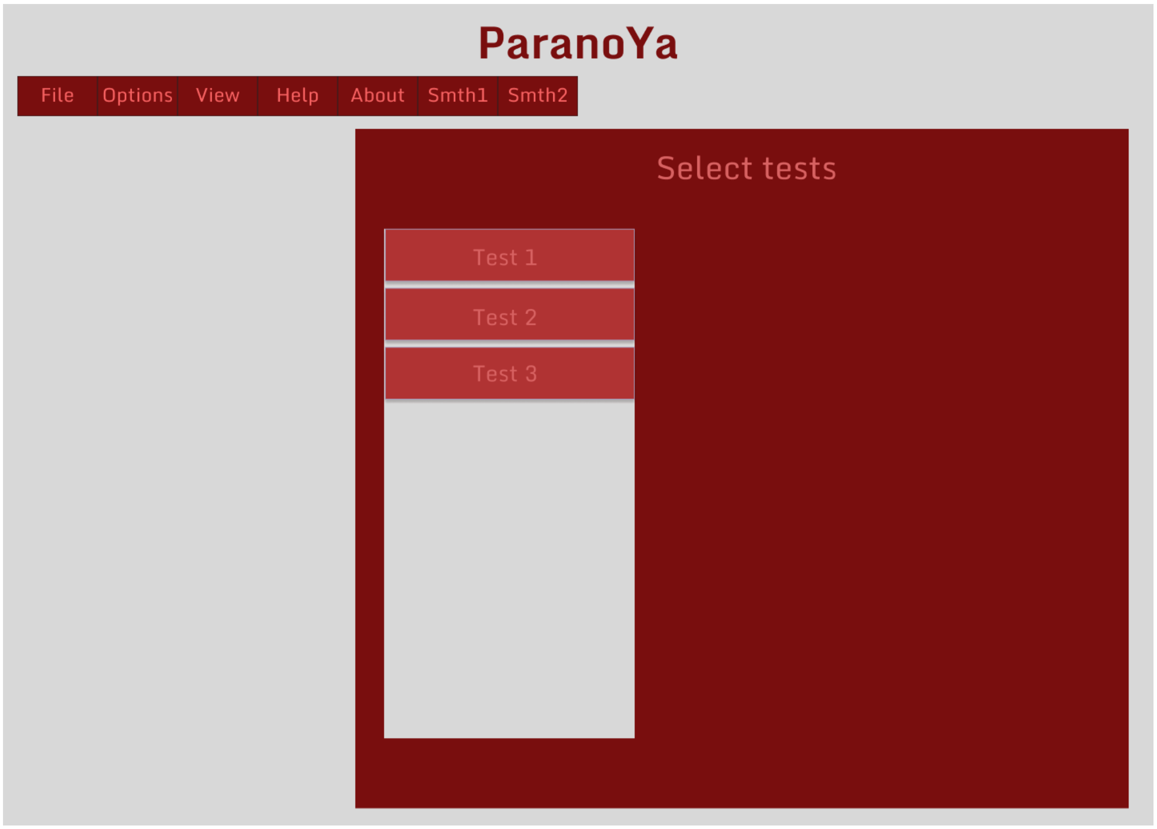


Figure 5: Add test

Purpose:	...
Navigation and User Interaction:	...

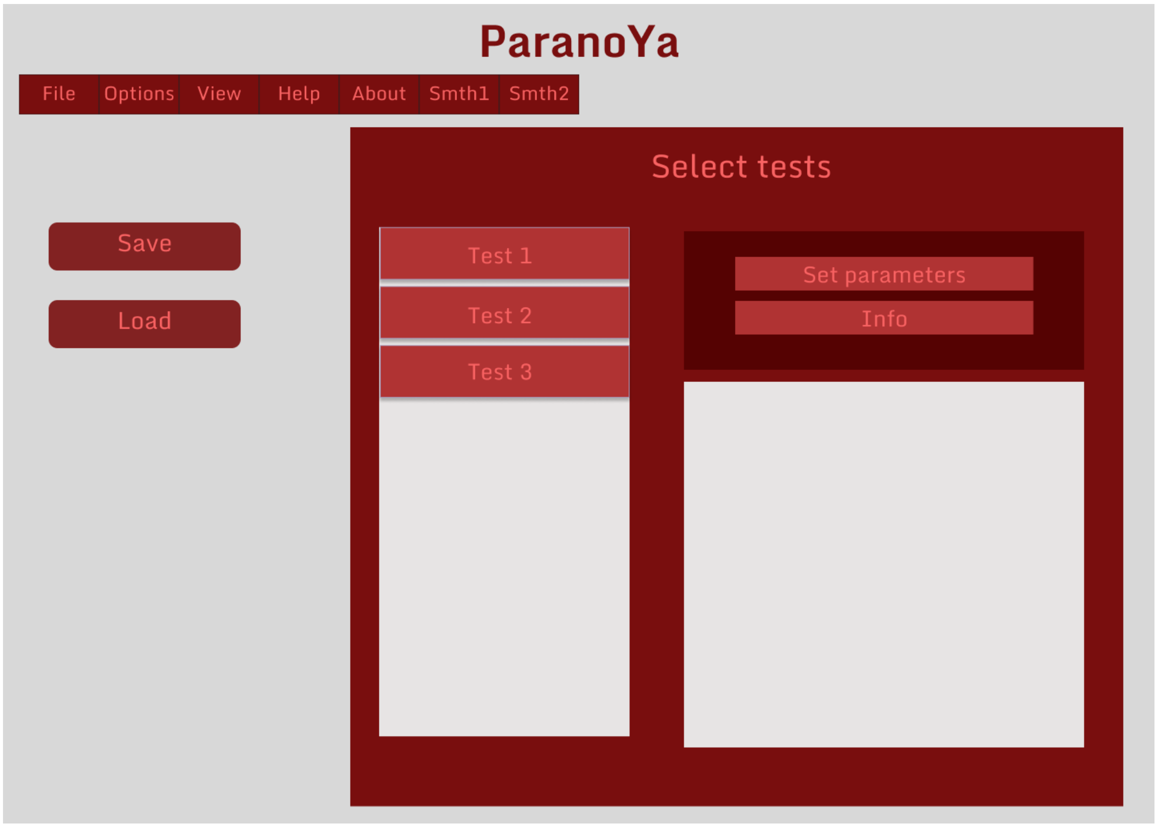


Figure 6: Add test

Purpose:	...
Navigation and User Interaction:	...

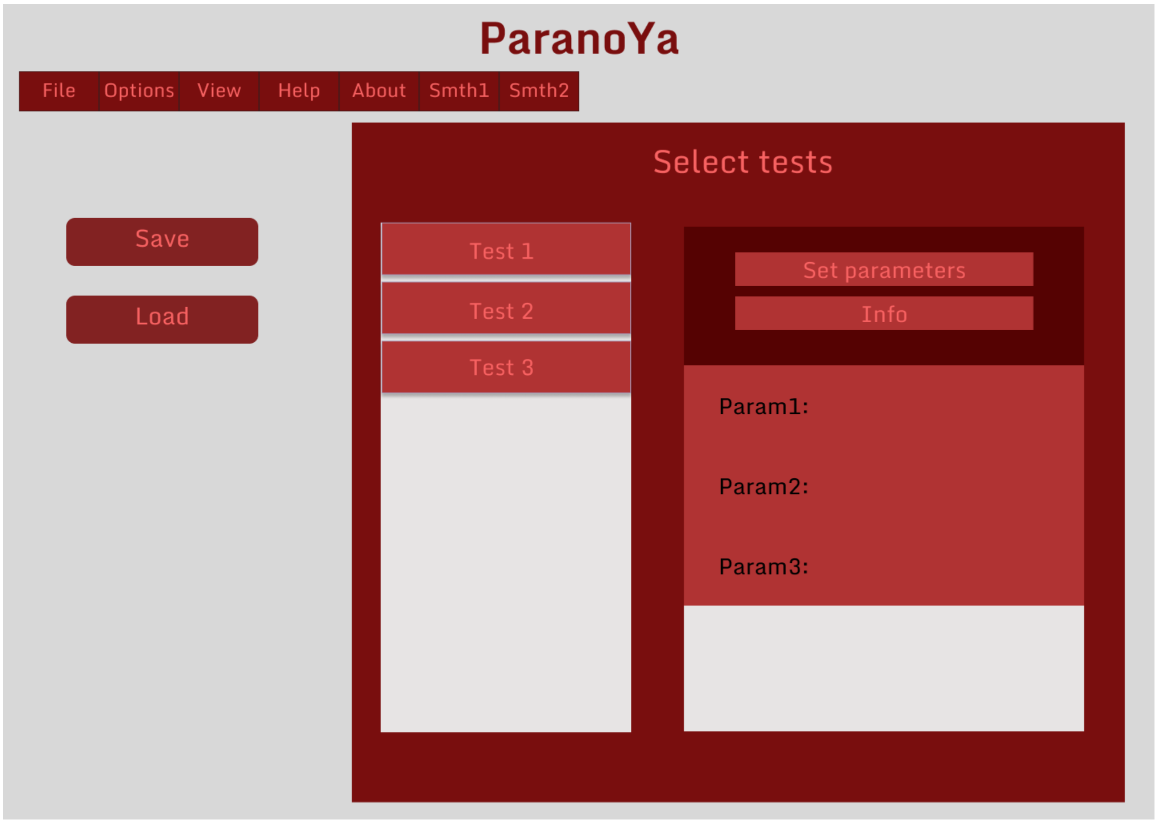


Figure 7: Set test Parameters

Purpose:	...
Navigation and User Interaction:	...

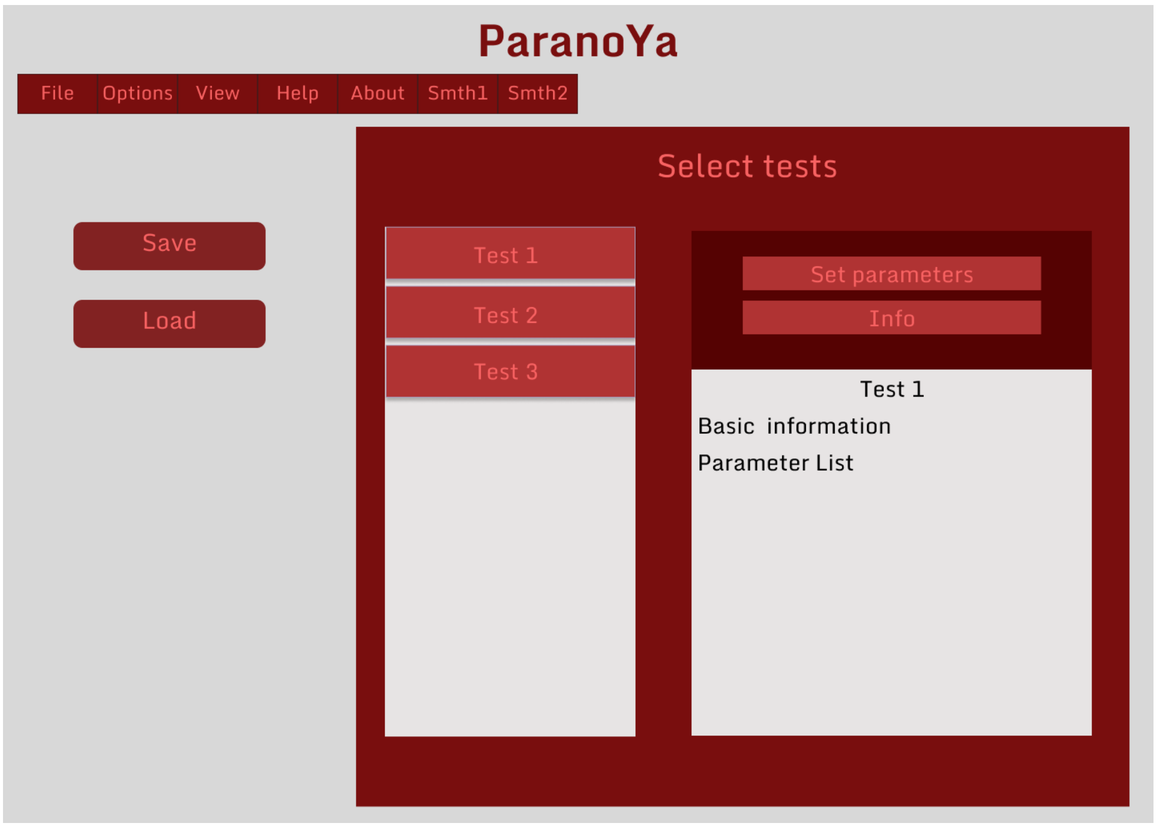


Figure 8: Show Test Informaion

Purpose:	...
Navigation and User Interaction:	...

3.5 Sequences

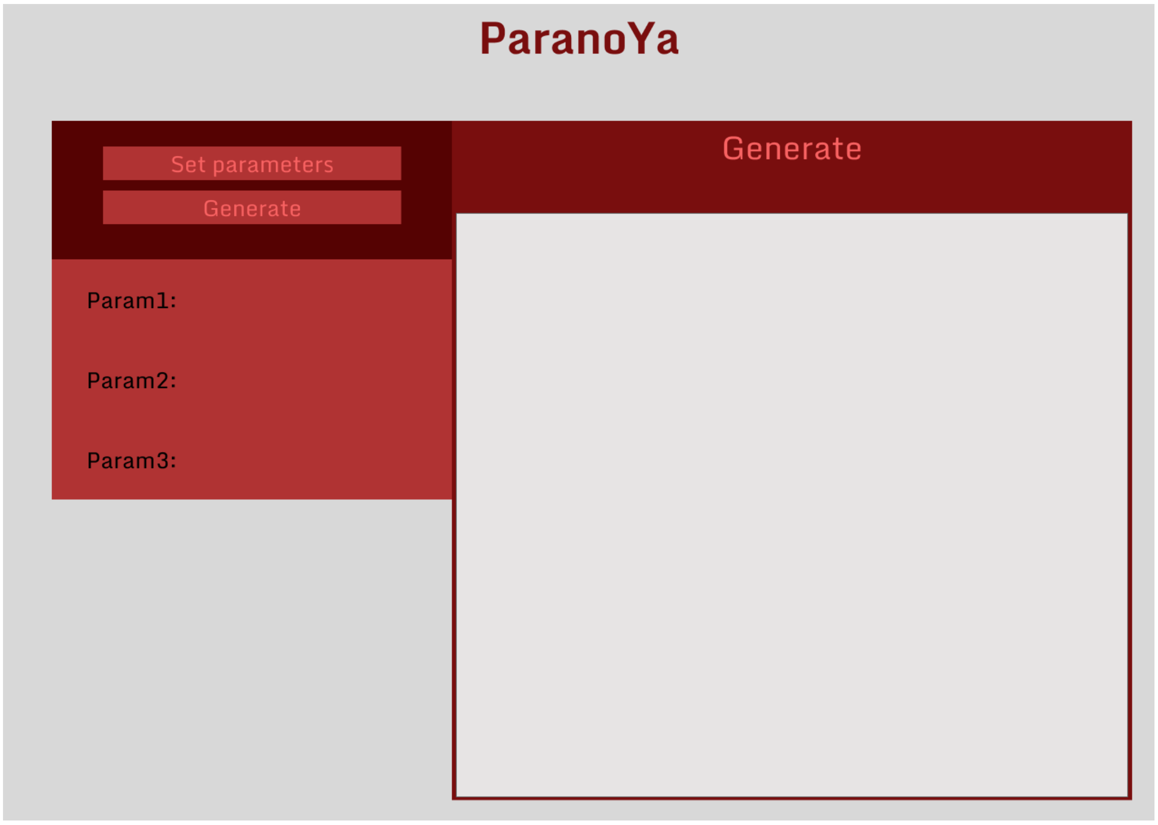
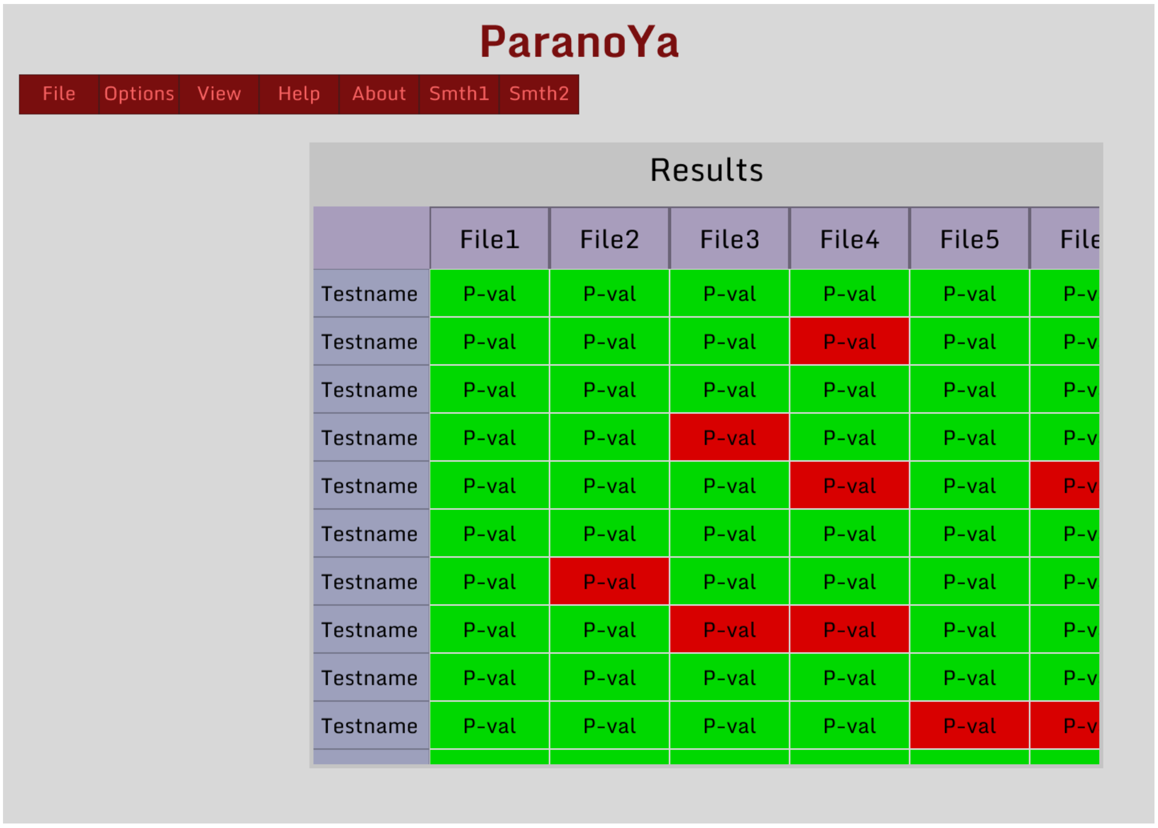


Figure 9: Generate

Purpose:	...
Navigation and User Interaction:	...

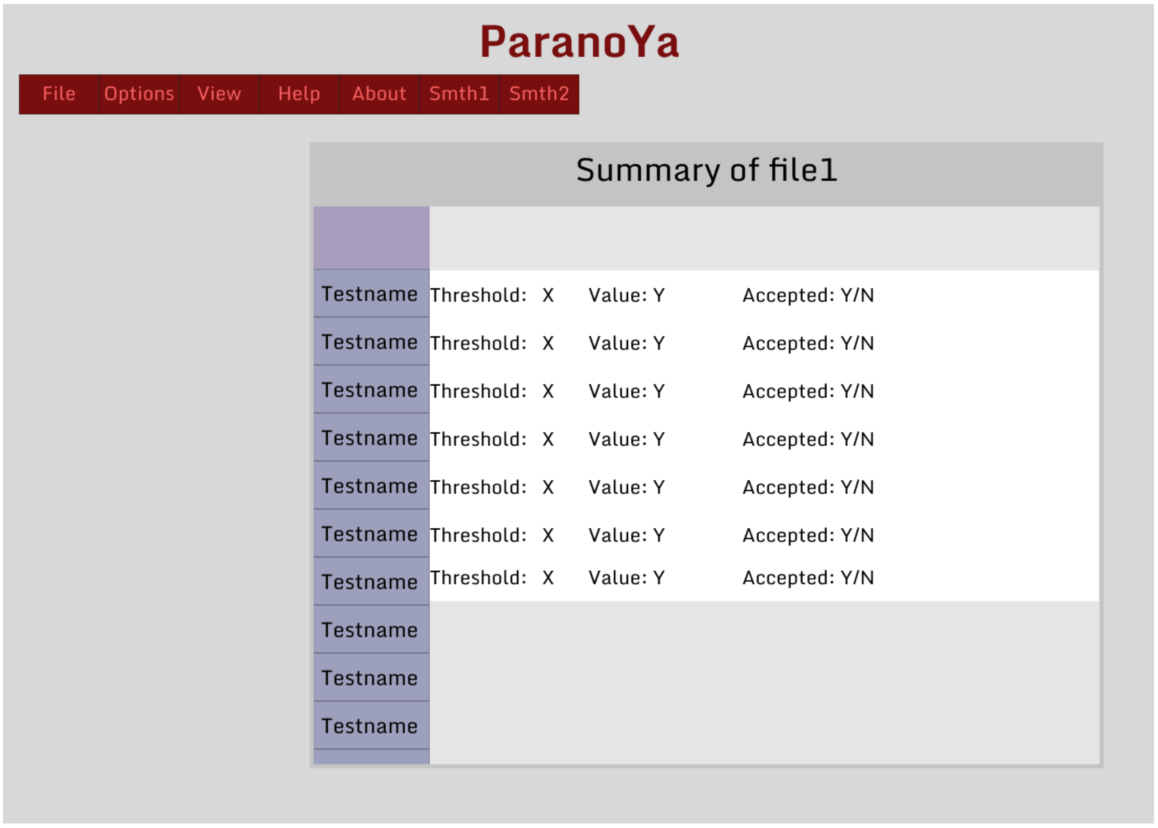
3.6 Results



	File1	File2	File3	File4	File5	File
Testname	P-val	P-val	P-val	P-val	P-val	P-v
Testname	P-val	P-val	P-val	P-val	P-val	P-v
Testname	P-val	P-val	P-val	P-val	P-val	P-v
Testname	P-val	P-val	P-val	P-val	P-val	P-v
Testname	P-val	P-val	P-val	P-val	P-val	P-v
Testname	P-val	P-val	P-val	P-val	P-val	P-v
Testname	P-val	P-val	P-val	P-val	P-val	P-v
Testname	P-val	P-val	P-val	P-val	P-val	P-v
Testname	P-val	P-val	P-val	P-val	P-val	P-v
Testname	P-val	P-val	P-val	P-val	P-val	P-v

Figure 10: Results

Purpose:	...
Navigation and User Interaction:	...



Testname	Threshold: X	Value: Y	Accepted: Y/N
Testname	Threshold: X	Value: Y	Accepted: Y/N
Testname	Threshold: X	Value: Y	Accepted: Y/N
Testname	Threshold: X	Value: Y	Accepted: Y/N
Testname	Threshold: X	Value: Y	Accepted: Y/N
Testname	Threshold: X	Value: Y	Accepted: Y/N
Testname	Threshold: X	Value: Y	Accepted: Y/N
Testname			
Testname			
Testname			

Figure 11: Detailed results

Purpose:	...
Navigation and User Interaction:	...

4 Implementation

4.1 UML Diagrams

4.1.1 Use Case Diagram

Each Use case describes a sequence of actions that provide something of measurable value to an Actor and is drawn as a horizontal ellipse. In our diagram are described actions, which are offered to the Actor operating with an app. Actor in our case is capable of several actions, to name a few, *File options*, *Selects tests*, *Tasks*, *Tests evaluation* etc. Each action has its respective Action and Sequence diagram, describing action more detaily in pages below.

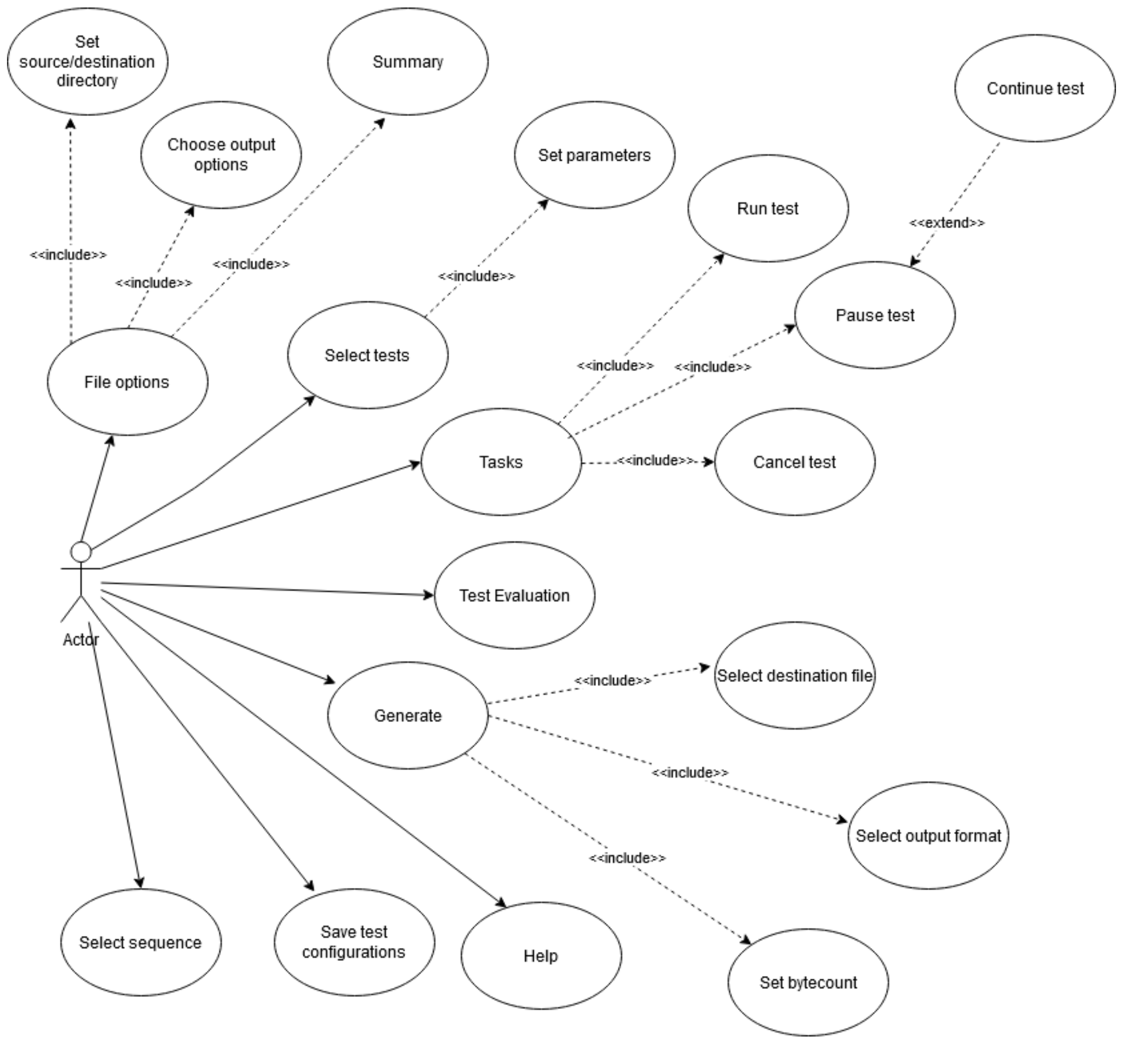


Figure 12: paranoYa - Use case diagram

4.1.2 Sequence Diagrams

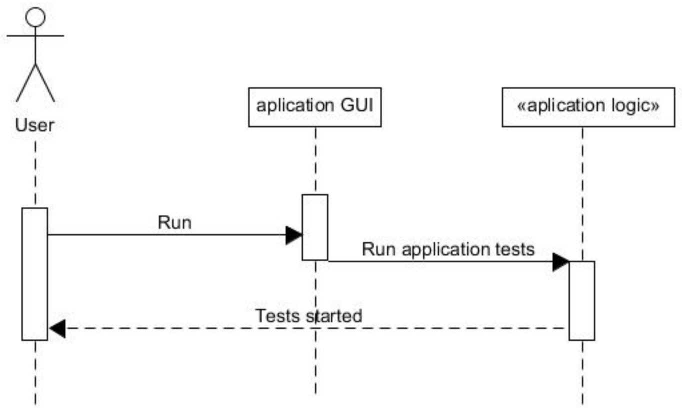


Figure 13: paranoYa - Sequence diagram (Run)

The user interacts with the app's graphical interface. In the *Tasks* tab in the application navigation bar, selects *Run*. Pseudo-random sequence testing starts. Start-up is preceded by loading a sequence, selecting a methodology.

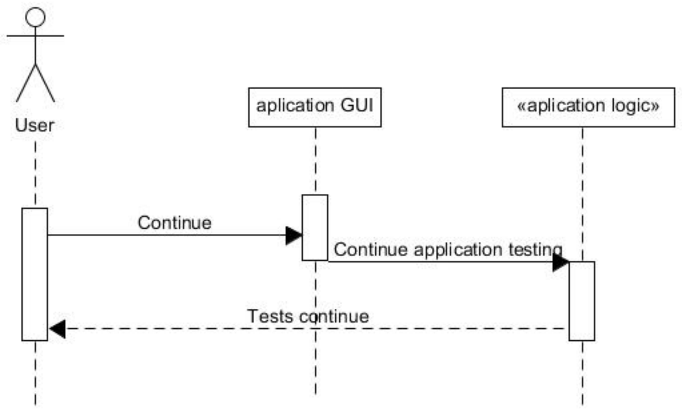


Figure 14: paranoYa - Sequence diagram (Continue)

The user interacts with the app's graphical interface. In the *Tasks* tab in the application navigation bar, selects *Continue*. Pseudo-random sequence testing continues. Actions needed before that *Run* and *Pause* the testing.

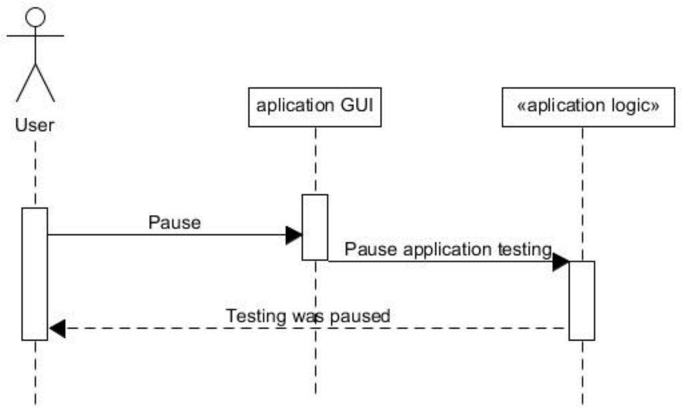


Figure 15: paranoYa - Sequence diagram (Pause)

The user interacts with the app's graphical interface. In the *Tasks* tab in the application navigation bar, selects *Pause*. The pseudo-random sequence testing is discontinued. The interrupt is preceded by *Run* testing.

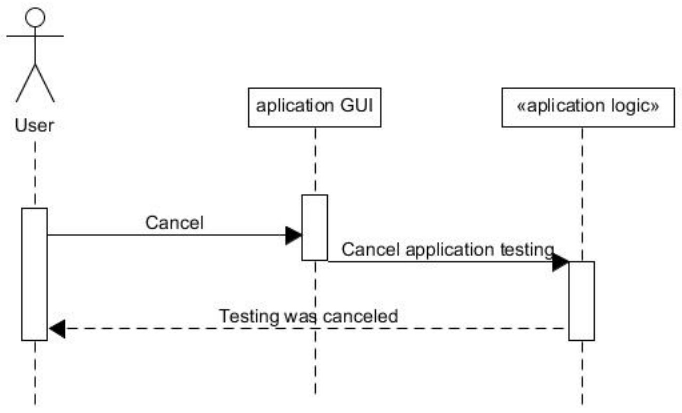


Figure 16: paranoYa - Sequence diagram (Cancel)

The user interacts with the app's graphical interface. In the *Tasks* tab in the application navigation bar, selects *Cancel*. The pseudo-random sequence testing stops. Stopping is preceded by *Run* the testing.

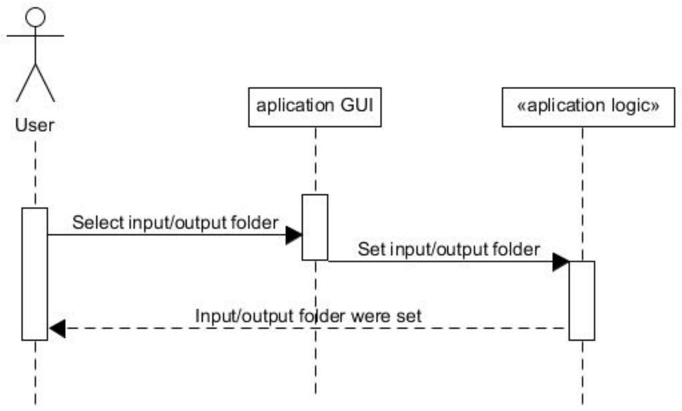


Figure 17: paranoYa - Sequence diagram (Set input/output folder)

The user interacts with the app's graphical interface. In the *File* tab in the application navigation bar, selects *Batch process...*. Next window is shown. This window belongs Source directory, Destination directory, Output options and Summary. After clicking on the button *Set...*, the user selects Source directory in the option Source directory and then he clicks button *OK*. This directory is also set as Destination directory by default. If user would like to change destination directory, he sets it in a similar way like Source directory.

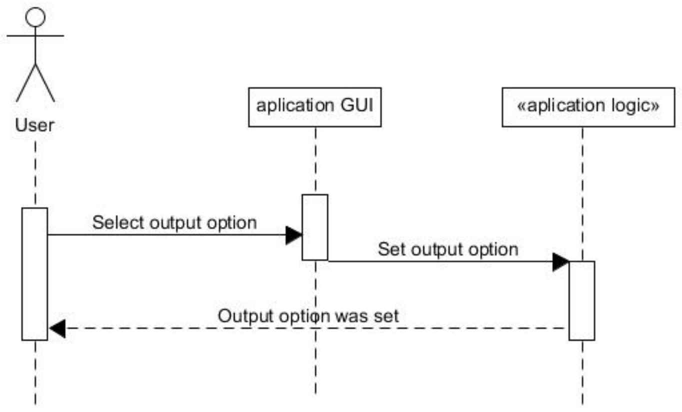


Figure 18: paranoYa - Sequence diagram (Select output option)

The user interacts with the app's graphical interface. In the *File* tab in the application navigation bar, selects *Batch process...*. Next window is shown. This window belongs Source directory, Destination directory, Output options and Summary. In the part Output options, the user selects one of the following options: XML, HTML, XML + HTML.

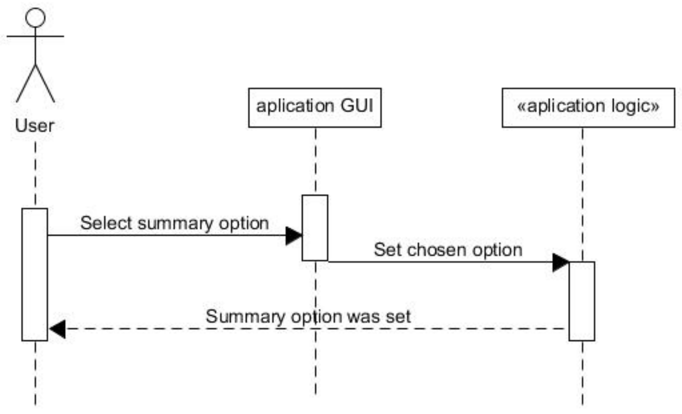


Figure 19: paranoYa - Sequence diagram (Select summary option)

The user interacts with the app's graphical interface. In the *File* tab in the application navigation bar, selects *Batch process...*. Next window is shown. This window belongs Source directory, Destination directory, Output options and Summary. In the part Summary, the user selects none, one or both of the following options: Generate summary HTML file, Prase P-values from all sequences..

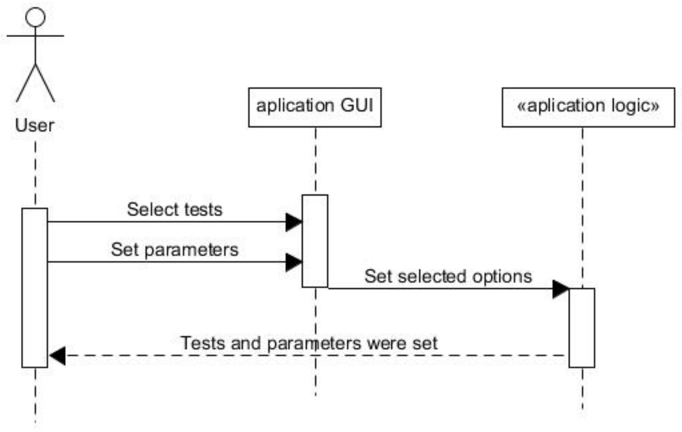


Figure 20: paranoYa - Sequence diagram (Set parameters for selected tests)

The user interacts with the app's graphical interface. In the main menu selects test, which would like to run. Clicks button *Add new* and test is inserted. User can set parameters for inserted test by inscribing values, for example. N - length of input string or M - length in bits of each block.

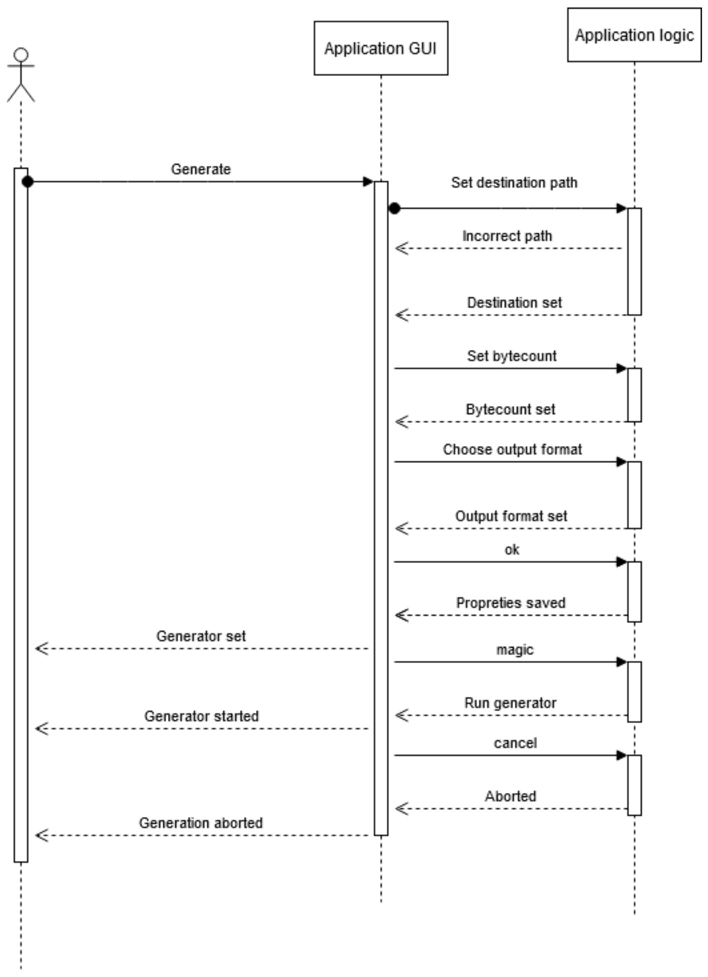


Figure 21: paranoYa - Sequence diagram (Generator)

User interacts with the application GUI. In the main menu clicks Generate. A new window opens with three tasks. First, the User selects the path to a Destination file, where the generated sequence is going to be saved. If the User enters an incorrect path, he will be notified until a valid path is given. Second, the User has to choose an ouptut format from a predefined list of available formats with a radio button. Third, the User has to provide a byteCount, which will be a number written to a text field.

Now the User has three options. Clicking the OK button will save the parameters of the Generator. Clicking the Magic button will start the generating process. Clicking Cancel will abort the the whole process and will close the Generator window.

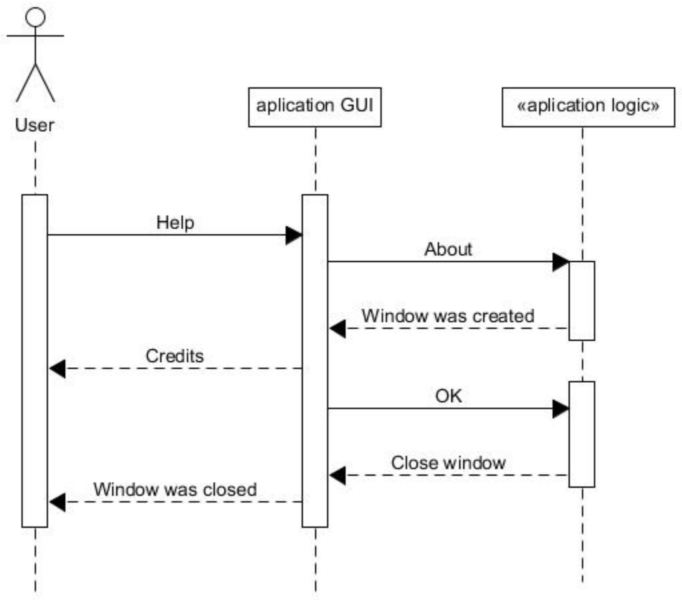


Figure 22: paranoYa - Sequence diagram (Help)

User interacts with the application GUI. In the main menu clicks Help. a submenu appears with one element named About... Clicking the About... button will open the Credits window. The Credit window can be closed with the ok button.

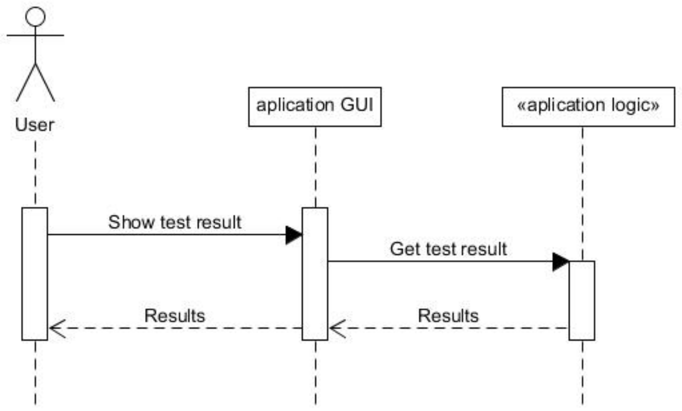


Figure 23: paranoYa - Sequence diagram (Show test results)

After testing has ended, user has an option to show test results. When selected, it retrieves results from application logics and displays it to user via application GUI.

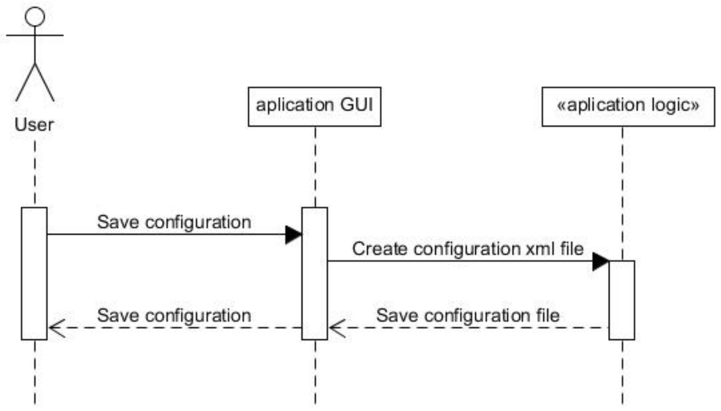


Figure 24: paranoYa - Sequence diagram (Save configuration)

User has an option to save current configuration in an XML file. The configuration is exported by application logics to an XML file which is sent back to user.

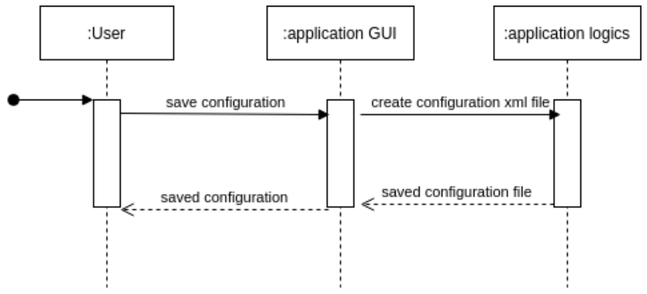


Figure 25: paranoYa - Sequence diagram (Load sequence)

By selecting File -¿ Load Sequence, user is able to load a sequence into the program.

4.1.3 Activity Diagrams

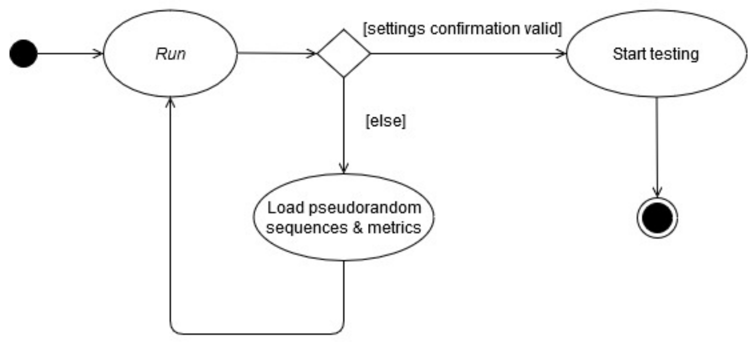

Run
[settings confirmation valid]
Start testing
[else]
Load pseudorandom sequences & metrics

Figure 26: paranoYa - Activity diagram (Run)

Algorithm 1: Start testing. Function triggered after user click event.

```
1 Function StartTests(event):
2     if settings_valid then
3         start_testing()
4     else
5         load_sequence()
6         run()
7     end
```

Action *Start testing*. Before app's logic can proceed with running the tests, it has need to check whether settings, inputed by user are valid. If settings validation failed, user is prompt to load a sequence and then proceed to run the testing.

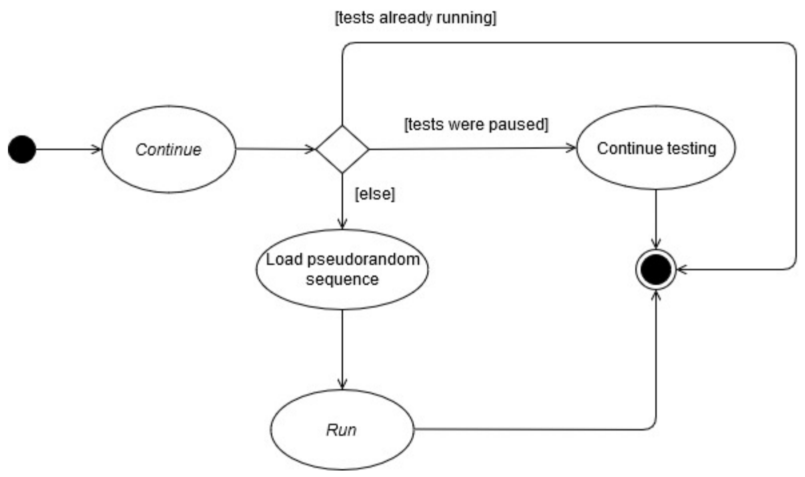


Figure 27: paranoYa - Activity diagram (Continue)

Algorithm 2: Continue testing. Function triggered after user click event.

```
Function ContinueTests(event):
    if tests_running then
        return
    end
    if tests_paused then
        continue_testing()
    else
        load_sequence()
        run()
    end
```

Action *Continue testing*. If tests are running *Continue* is disallowed. If tests being paused then testing will continue. In case that both checks failed user is prompted to load a sequence.

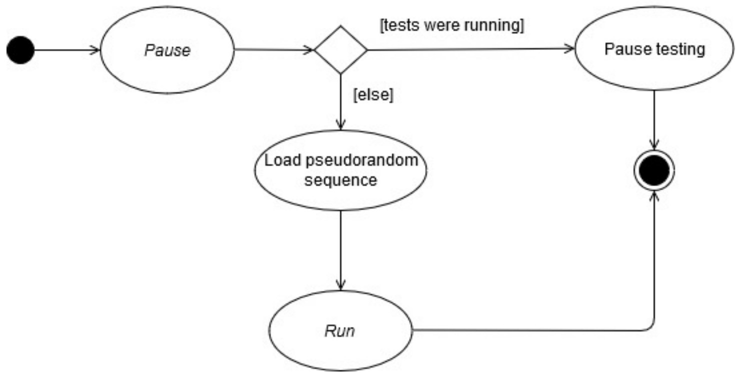


Figure 28: paranoYa - Activity diagram (Pause)

Algorithm 3: Pause testing. Function triggered after user click event.

```
1 Function PauseTests(event):
2     if tests_running then
3         pause_testing()
4     else
5         load_sequence()
6         run()
7     end
```

Action *Pause testing*. If tests are running then *Pause* testing. If tests are not running prompt user for loading a sequence.

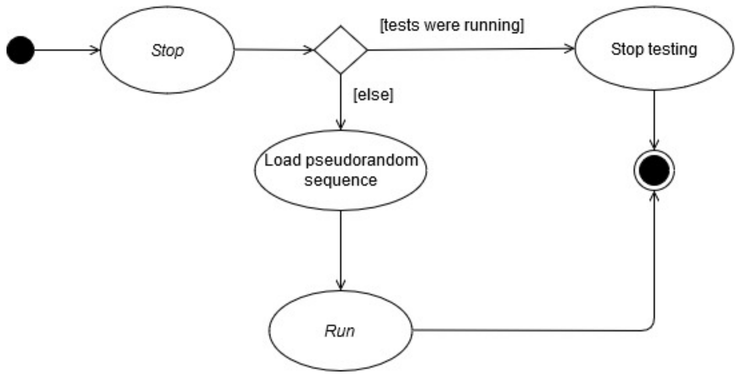


Figure 29: paranoYa - Activity diagram (Stop)

Algorithm 4: Stop testing. Function triggered after user click event.

```
Function StopTests(event):
    if tests_paused then
        return
    end
    if tests_running then
        stop_testing()
    else
        load_sequence()
        run()
    end
```

Action *Stop testing.* If tests are paused, do nothing. If tests are running *Continue* then *Stop* testing. If tests are not running prompt user for loading a sequence.

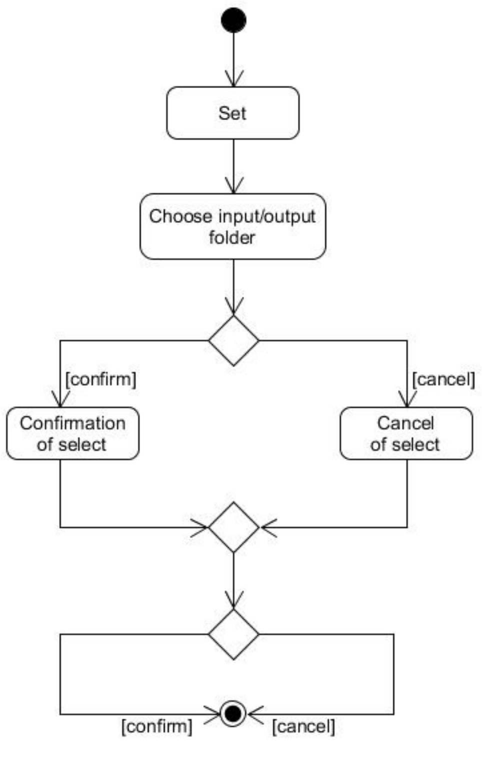


Figure 30: paranoYa - Activity diagram (Set input/output folder)

Algorithm 5: Set input/output folder. Function triggered after user click event.

```
1 Function SetInputOutputFolder($event$):
2     $chosenFolder$ ¡- choose_folder();
3     if $ok$ then
4         $sourceDirectory$ ¡ - $chosenFolder$;
5     end
6     if $cancel$ then
7     end
```

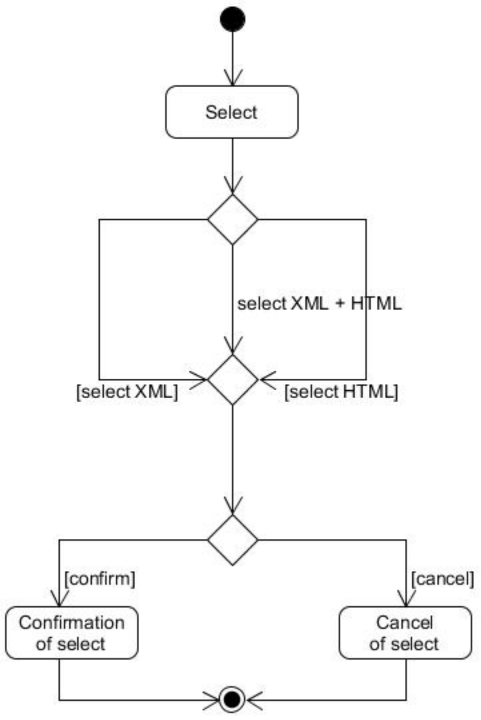


Figure 31: paranoYa - Activity diagram (Select output option)

Algorithm 6: Select output option. Function triggered after user click event.

```
1 Function SelectOutputOption(event):
2     if $XMLselected$ then
3         $outputOption$ ¡ - $XML$;
4     end
5     if $HTMLselected$ then
6         $outputOption$ ¡ - $HTML$;
7     else
8         $outputOption$ ¡ - $XMLHTML$;
9     end
```

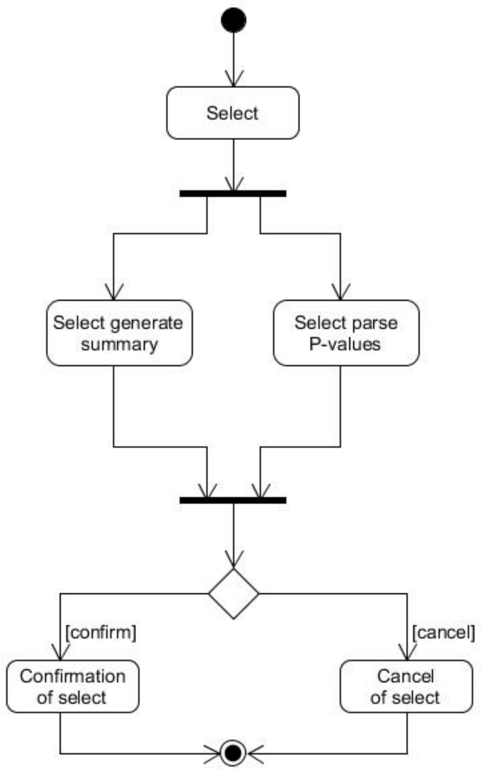


Figure 32: paranoYa - Activity diagram (Select summary option)

Algorithm 7: Select summary option. Function triggered after user click event.

```
Function SelectSummary(event):
    if isClickedGeneratet then
        generateSum ¡ - true;
    else
        generateSum ¡ - false;
    end
    if isClickedParse then
        parsePvalues ¡ - true;
    else
        parsePvalues ¡ - false;
    end
```

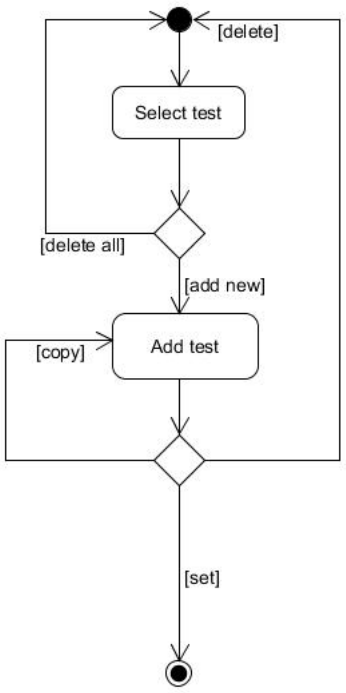


Figure 33: paranoYa - Activity diagram (Set parameters for selected tests)

Algorithm 8: Set parameters for selected tests. Function triggered after user click event.

```
Function SetParameters(event):
    select_test();
    if add then
        add_test();
        if copy then
            copy_test();
        end
        if delete then
            delete_test();
        end
    end
    if deleteAll then
        delete_all_tests();
    end
```

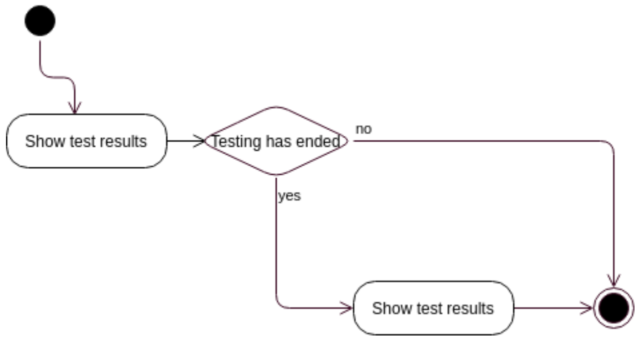


Figure 34: paranoYa - Activity diagram (Show test results)

Algorithm 9: Show test results, after testing has ended, triggered after user click

```
Function ShowTestResults(event):
    if testing_has_ended then
        show_test_results()
    end
```

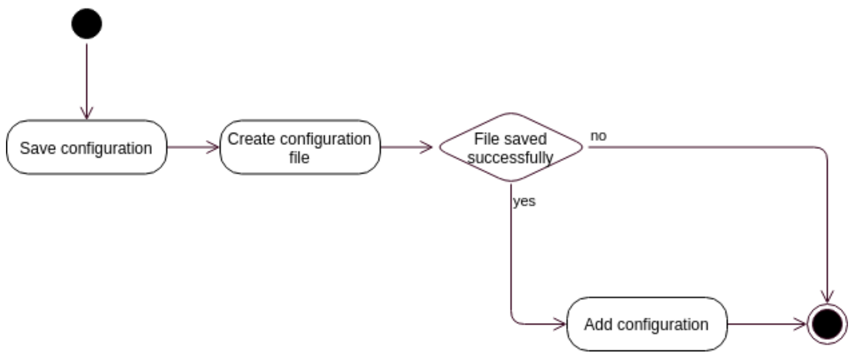


Figure 35: paranoYa - Activity diagram (Save configuration)

Algorithm 10: Save configuration, triggered by selecting the option

```
Function SaveConfiguration(event):
    file ¡ - create_configuration_file()
    if file then
        add_configuration()
        save_configuration()
    end
```

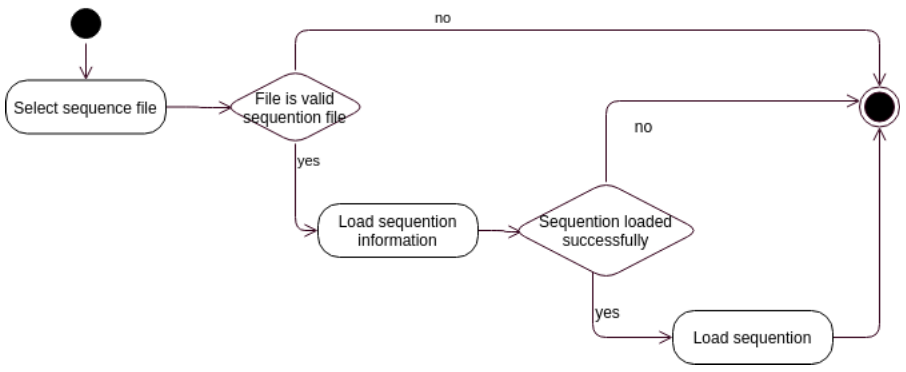

no
Select sequence file
File is valid sequention file
yes
no
Load sequention information
Sequention loaded successfully
yes
Load sequention

Figure 36: paranoYa - Activity diagram (Load sequence)

Algorithm 11: Load sequence, triggered by selecting the option

```
Function LoadSequence(file):
    if is_valid_sequention_file(file) then
        sequention ¡- import_sequention(file)
        if sequention then
            load_sequention(sequention)
        end
    end
```

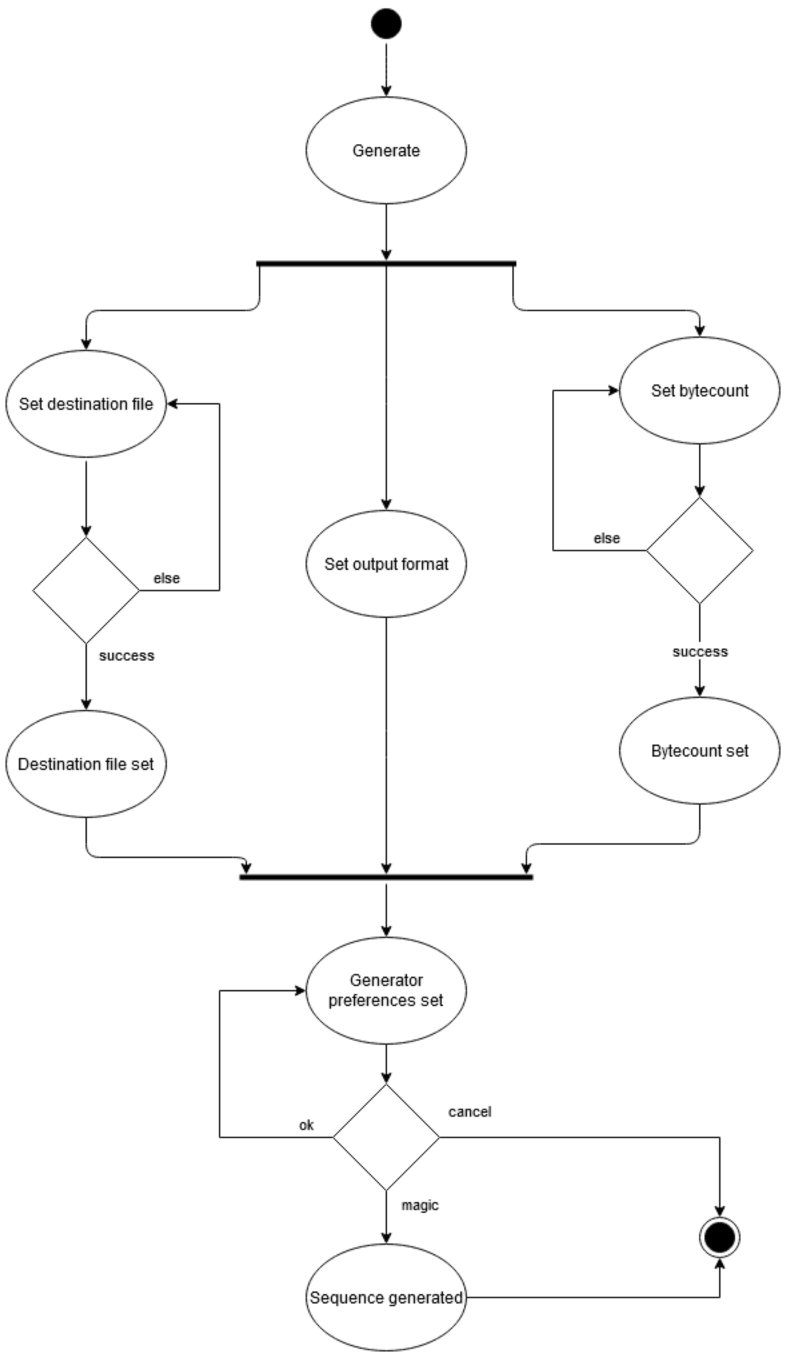


Figure 37: paranoYa - Activity diagram (Generate sequence)

Algorithm 12: Generate sequence into file.

```
Function Generate(event):
    generate();
    destinationFile_select();
    if success then
        destinationFile_set() ;
    else
        destinationFile_select() ;
    end
    OutputFormat_set();
    byteCount_select();
    if success then
        byteCount_set() ;
    else
        byteCount_select() ;
    end
    generatorPreferences_select();
    if Ok then
        generatorPreferences_select();
    end
    if Magic then
        Sequence_generate();
    end
    if Cancel then
        Generate_quit();
    end
```

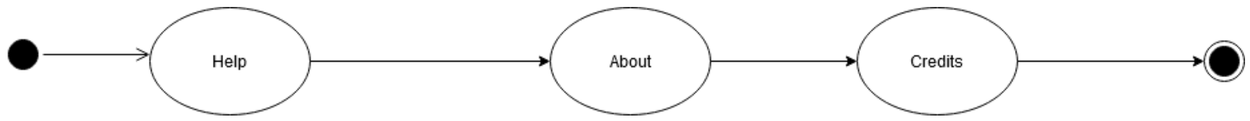


Figure 38: paranoYa - Activity diagram (Help)

Algorithm 13: Open help for inforamtion

```
Function Help(event):
    help();
    About_click();
    Credits.show();
```

4.2 Acceptance tests

ID	1	**Name**	Show test results
Interface	Client / application GUI / application logics		
Input	Successfully ended testing		
Output	Test results are displayed to user in application GUI		
Step	**Action**	**Expected reaction**	
1	Testing ended	Application GUI shows an option to display test results	
2	Users selects to show test results	Test results are displayed to user	

ID	2	**Name**	Save configuration
Interface	Client / application GUI / application logics		
Input	-		
Output	Configuration is saved in a XML file		
Step	**Action**	**Expected reaction**	
1	User makes a change in a configuration	Application saves change for configuration	
2	Users selects to save configuration	Configuration is saved in a XML file	

ID	3	**Name**	Load sequence
Interface	Client / application GUI / application logics		
Input	-		
Output	Sequence is loaded into application		
Step	**Action**	**Expected reaction**	
1	User selects to load sequence	A file input is displayed to user	
2	Users selects valid configuration file	A sequence is loaded into application from the chosen file	

ID	4	**Name**	Run
Interface	Client / application GUI / application logic		
Input	Click event		
Output	Tests started		
Step	**Action**	**Expected reaction**	
1	User enters tab Settings	Tab window is opened	
2	User selects option Run	Tests start running	

ID	5	**Name**	Continue
Interface	Client / application GUI / application logic		
Input	Click event		
Output	Tests continue		
Step	**Action**	**Expected reaction**	
1	User enters tab Settings	Tab window is opened	
2	User selects option Continue	Stopped tests will run	

ID	6	**Name**	Pause
Interface	Client / application GUI / application logic		
Input	Click event		
Output	Tests were paused		
Step	**Action**	**Expected reaction**	
1	User enters tab Settings	Tab window is opened	
2	User selects option Pause	Running tests will be paused	

ID	7	**Name**	Cancel
Interface	Client / application GUI / application logic		
Input	Click event		
Output	Tests stopped		
Step	**Action**	**Expected reaction**	
1	User enters tab Settings	Tab window is opened	
2	User selects option Cancel	Running tests will stop	

ID	8	**Name**	Cancel
Interface	Client / application GUI / application logic		
Input	Click event		
Output	Set input/output folder		
Step	**Action**	**Expected reaction**	
1	User chooses input/output folder	Input/output folder is chosen	
2	User selects option Cancel	Chosen folders are canceled	
3	User selects option OK	Chosen folders are set	
4	User selects option Cancel	Selected options are canceled	
5	User selects option OK	Selected options are successfully set	

ID	9	**Name**	Cancel
Interface	Client / application GUI / application logic		
Input	Click event		
Output	Selected output option		
Step	**Action**	**Expected reaction**	
1	User selects one output option	Output option is selected	
2	User selects option Cancel	Selected options are canceled	
3	User selects option OK	Selected options are successfully set	

ID	10	**Name**	Cancel
Interface	Client / application GUI / application logic		
Input	Click event		
Output	Selected summary option		
Step	**Action**	**Expected reaction**	
1	User selects one summary option	Summary option is selected	
2	User selects option Cancel	Selected options are canceled	
3	User selects option OK	Selected options are successfully set	

ID	11	**Name**	Cancel
Interface	Client / application GUI / application logic		
Input	Click event		
Output	Set parameters for selected tests		
Step	**Action**	**Expected reaction**	
1	User selects test	Selected test is shown	
2	User selects option Add new	Advanced options are shown	
3	User set parameters for chosen test	Parameters are set	
4	User selects option Copy	Test is copied with set parameters	
5	User selects option Delete	Current test is deleted	
6	User selects option Delete All	All tests are deleted	

5 Implementation

5.1 Creation of shared object from Marek Sys libraries

Firstly, we created *.o* files from files in *src/* folder of Marek Sys library by command

```
gcc -c -fPIC utilities.c -o utilities.o
```

Secondly, we made shared object *.so* by command

```
gcc -shared -o liboutput.so library1.o library2.o library3.o
```

Which gave us shared object which can be implemented via Cython