RIGA TECHNICAL UNIVERSITY
Faculty of Computer Science and Information Technology
Institute of Applied Computer Systems

Aleksandrs SUHORUKOVS
Student of the Doctoral study programme "Computer Systems"

METHODS, TOOLS AND EFFICIENCY OF COMPUTER SYSTEM AUTOMATED TESTING

Summary of Doctoral Thesis

Scientific supervisor
Dr. sc. ing., professor
L. ZAITSEVA

Riga 2011

UDK 004.415.53 (043.2)
Su 264 m

Suhorukovs A. Methods, tools and efficiency of computer system automated testing. Summary of Doctoral Thesis. -R.:RTU, 2011. -37 p.

Printed according to the decision of RTU Institute of Applied Computer Systems Board meeting, July 1, 2011, Protocol No. 72.

This work has been supported by the European Social Fund within the project “Support for the implementation of doctoral studies at Riga Technical University”.

ISBN 978-9934-10-204-2

DOCTORAL THESIS
IS SUBMITTED FOR THE DOCTOR'S DEGREE
IN ENGINEERING SCIENCE
AT RIGA TECHNICAL UNIVERSITY

The defence of the thesis submitted for the doctoral degree in engineering science (computer systems) took place at an open session in 1/3 Meza Street, auditorium 202, on December 5, 2011.

OFFICIAL REVIEWERS

Professor Dr.sc.ing. Uldis Sukovskis
Riga Technical University, Latvia

Dr.sc.comp. Mārtiņš Gills
AS Norvik Banka

Professor Ph.D. Radi Romansky
Technical University – Sofia, Bulgaria

DECLARATION

I hereby confirm that I have developed this thesis submitted for the doctoral degree in engineering science at Riga Technical University. This thesis has not been submitted for the doctoral degree at any other university.

Aleksandrs Suhorukovs …….(Signature)

Date: ……

The doctoral thesis is written in Latvian. It consists of introduction, 6 sections, conclusion, bibliography, including 30 figures and 8 tables. The thesis is printed on 130 pages. The bibliography consists of 130 entries.

ABSTRACT

The present Thesis is dedicated to the methods and tools of computer systems testing automation as well as estimation of their efficiency.

Topical directions of automated testing evolution are overviewed, popular test automation tools are classified. Unified automated testing model is developed allowing to describe automatable activities of software testing on high level. Mathematical model of automated testing efficiency estimation is developed allowing to achieve rational usage of a testing time. Automated test suite generation methods and adaptive performance testing methods are developed, providing the basis for developed tools, which were applied in the real testing projects.

CONTENTS

1. GENERAL DESCRIPTION OF THE THESIS

1.1. Research motivation

The thesis is dedicated to methods and tools of computer systems testing automation as well as estimation of their efficiency.

Nowadays the topic of software test automation is gaining more and more interest. Technologies evolve, systems grow in size and complexity, development cycles are more rapid. For these reasons amounts of tests explode and need for their quick execution arise. In these conditions in respect that software testing is time-consuming process testing efficiency can be increased by application of automated tests as their execution require much less human time comparing to manual test execution.

Need for multiple repeated test executions is the reason automation expediency. The need may arise when several input data sets should be verified or when changes get introduced into the system frequently. Testing efficiency can be notably improved if such repeating activities are delegated to tools.

Automated test never get tired, does not loose its attention and always complete a dedicated task precisely. Manual testing has its own benefits and drawbacks. Since both manual and automated testing have benefits and drawbacks the question about which of them should be applied in particular case is not trivial. Two extremes – ignoring automation possibilities completely and attempts to automate everything – are not rational both from laboriousness and testing quality points of view [19]. Identification of appropriate balance is the problem which currently do not have simple solution.

Technologies of testing automation are well developed today in several directions and are suitable for testing of different even specific systems [30]. Many tools are available for ensuring automation of functional and performance testing automation. Unit testing solutions are available for almost every existing programming language [12]. Diversity of tools ensures that wider range of tests can be automated but also makes choice of the most appropriate tool more difficult.

Many organizations in Latvia and all over the world attempted, attempt and already use facilities of test automation. However till now there are different opinions about how test automation should be done and what place should automation take in overall testing and

software development in general [21, 25]. Because of diversity of opinions it is difficult to decide what methods and to what extent have to be applied for particular project needs. It can lead to inefficient test planning when test automation is refused at all or when tests get automated which would be more beneficial when done manually.

When testing automation is already in use or when its introduction is planned it is important to estimate and predict efficiency of its application. Currently there is no method of automated testing efficiency estimation which would allow to compare selected automation strategy with corresponding manual testing strategy or other alternative strategies of automation.

Aforementioned problems explain the relevance of the topic and necessity for deeper research in methods and efficiency of computer system testing automation field.

1.2. The goal and the tasks of the thesis

The goal of the thesis is on the basis of testing methods analysis and test automation tools investigation to develop methods of testing automation, which would allow to achieve economy of testing-related time and/or better testing quality, and a model of automated test efficiency evaluation, as well as to apply the developed methods and tools in real projects.

In order to achieve the goal the following tasks should be completed:

1) to investigate existing automation methods of various testing activities and to classify currently available test automation tools;
2) to develop automated testing model which would help to identify potentially automatable activities in software testing;
3) to develop automated test efficiency evaluation model;
4) to develop test automation solution which would allow to achieve better testing efficiency than existing solutions;
5) to apply the developed solutions in real projects.

The specified tasks correspond to the structure of the thesis. Each task has a dedicated thesis chapter, but developed solutions are described in two chapters. General conclusions are placed in conclusions section.

1.3. Research methods, scientific novelty, practical value

The field of the research is computer system testing with application of software tools which automate execution of testing activities.

The object of the research is computer system automated testing methods, efficiency and tools which implement them.

The methods of the research include set theory, graph theory, algorithm design and analysis.

The novelty of the thesis is:

- automated test efficiency estimation model is developed;
- automated test suite generation methods are developed based on simple and compound state models;
- performance testing adaptive load scheduler model is developed and its implementation algorithm is proposed.

The main practical value of the results of the thesis is related to the developed test suite generation methods and performance testing adaptive load scheduler model which prove to be more efficient than other methods under certain conditions. These methods are implemented in tools developed by the thesis author. The tools were successfully applied in fourteen real testing projects in different organizations.

1.4. Structure of the thesis

The thesis consists of the introduction, six chapters and the conclusion.

The first chapter describes historical evolution of automated testing, overviews several directions of test generation methods research and introduces developed classification of test automation tools used to classify 32 popular tools.

The second chapter describes the development process of test automation tools and testware. Developed unified automated testing model is also described.

The third chapter introduces the developed automated testing efficiency evaluation model.

The fourth chapter describes developed solution for test suite generation.

The fifth chapter describes developed performance testing tool and adaptive performance testing method.

The sixth chapter overviews introduction and usage experience of the developed solutions described in the chapter four and five in real projects.

The thesis consists of 130 pages of text, 30 figures, 8 tables and 130 bibliographical references.

2. CONTENTS OF THE THESIS

2.1. Automation in software testing

On the basis of various sources [1, 4, 9, 12, 18, 24] it is possible to identify three important periods in which evolution of current automated testing tools started:

- 1970s – test generation, symbolic execution tools;
- 1980s – black box test automation, capture/playback tools;
- 1990s – modern unit testing frameworks.

In the first chapter of the thesis test generation methods are classified and test automation tool classification is developed and applied to classification of several currently available tools.

The task of test case generation can be described as follows. The system can be viewed as a function transforming input data to output data. Input data can be files, input from keyboard, mouse movements etc. Output can be generated files, displayed values or graphics etc. The task of test case generation is to find out for a given program such input data or sets of input data which correspond to some specified criteria.

There are several approaches to classification of test case generation methods. In the thesis four classifications are described based on:

1) significance of source code;
2) program execution type;
3) target criteria;
4) model types.

By significance of source code test case generation methods can be divided in:

1) white box methods – tests get selected based on information about program implementation: internal design of the program is known and its source code is available;
2) black box methods – tests get selected based on program external behaviour, defined by requirement specification or by the program execution itself;
3) grey box methods – combine first two methods, usually it means that source code is used for test case design, but the goals of testing are based on program functionality.

By program execution type test case generation methods can be divided in:

1) static – tests are generated without executing the program by using static analysis of source code and methods of symbolic execution [10, 15];
2) dynamic – program is executed and test cases are generated based on data acquired during the execution [16];
3) hybrid – combine facilities of both aforementioned approaches [11].

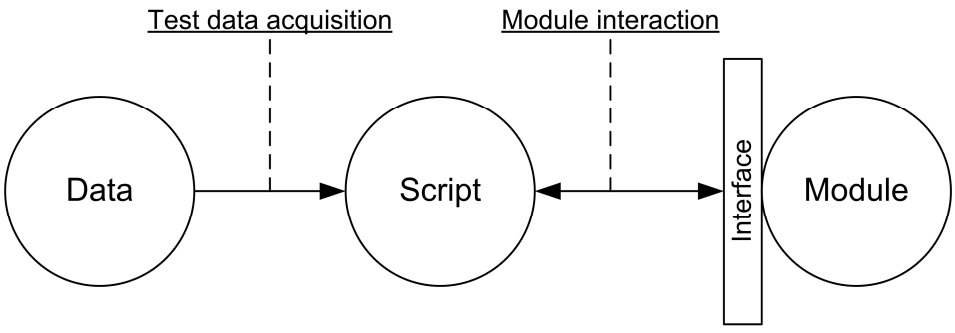
By target criteria test case generation methods can be divided in [6]:

1) based on random paths – tests get selected using random data or random execution paths;
2) based on goals – test cases are generated taking into account that the execution path should correspond to some specified constraints (test goals);
3) based on paths – test cases are generated in such way that the program should traverse some specific execution paths.

By model types test case generation methods are divided according to model types which form the basis of generation algorithm [2]. Several methods exist allowing to generate test cases from contract-like specifications [17, 20], abstract type specifications [8, 22], labelled transition systems [13, 14] and many other model types.

Generation algorithms are heavily dependent on modelling approach. This fact can explain why test case automated generation till now is not widely used in software development industry: each different model or algorithm provide just partial, very limited view on program under test. In order to cover the program more widely several different models and generation algorithms should be applied what require much time for modelling and much knowledge of generation algorithms.

For test execution automation (or just test automation) tool classification based on development and execution context of automation test a dynamic automated test model was developed which is illustrated in figure 2.1.



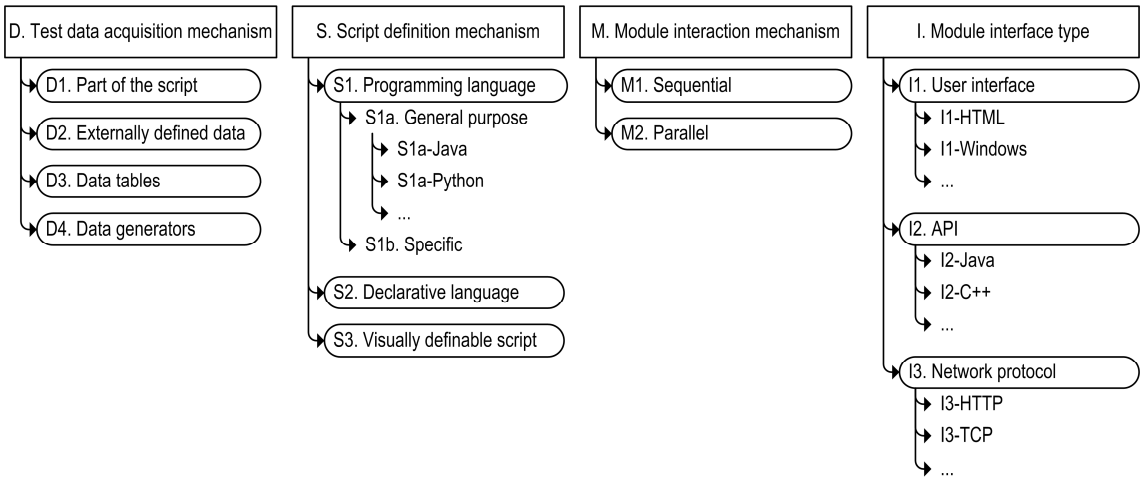
2.1. fig. Automated test model

In the developed model four aspects can be identified which are selected as classification criteria. Other criteria examples which can be derived from the model such as data definition mechanism or type of module are not relevant for analysis of test automation tool applicability.

1. Test data acquisition mechanism. This criterion determines how the script acquires data, whether data are separated from script or data are part of the script. This criterion is denoted by letter D.
2. Script definition mechanism. Determines how the script gets developed, what structure it has and how it gets interpreted. This criterion is denoted by letter S.
3. Module interaction mechanism. Determines the way of module usage, whether actions are executed sequentially or in parallel. This criterion is denoted by letter M.
4. Module interface type. Determines level on which script interacts with the module. This criterion is denoted by letter I.

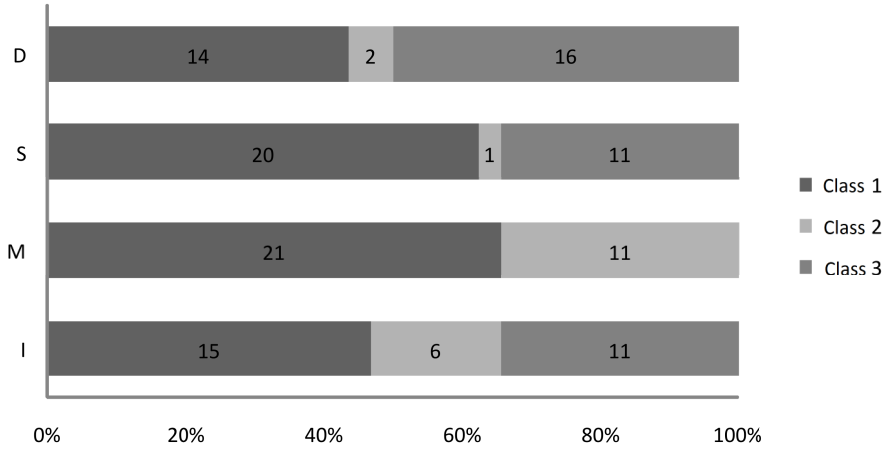
Each criterion defines several variants (classes) which are denoted by letter of the corresponding criterion and class number, for example D3 where D denotes a criterion and 3 denotes the third class by criterion D.

All criteria with possible values are consolidated in figure 2.2. All criteria and their classes are described and justified in detail in the thesis.



2.2. fig. Tools classification criteria

On the basis of the developed classification 32 tools were analyzed and classified creating partitioning of these tools into classes by each criterion. Full classification table of the tools is included in the thesis but summary statistics of the tool classification is illustrated in figure 2.3.



2.3. fig. Classification of the tools by four criteria

The smallest number of classified tools belong to class D4 (data generators) – no tools; D2 (data separated from script) – two tools and S2 (script developed in declarative language) – one tool. Small number of D2 and S2 class tools can be explained so that these classes can be viewed as transitional forms. Data separation from script can be easily improved and transformed to data table functionality (D3 class) allowing much wider test automation facilities. The same reasoning apply to S2 class: if script has to be developed in declarative language it is relatively easy to develop visual interface simplifying the development and then the tool becomes of S3 class.

Among analyzed tools no one can be included in class D4 (data generators). It means that tools of such type are relatively rare. Test generation tools mostly exist as a separate category of tools which do not provide test automation facilities.

The proposed classification can be used to identify the most appropriate test automation tool classes depending on task specifics.

2.2. Automated test development processes

Various different test automation tools exist appropriate for different testing tasks each with its own peculiarities. However despite of this diversity there are situations when existing tools do not fit well and new solution specific to some particular task has to be developed. In the second chapter of the thesis design specifics of test automation tools are analyzed.

From the perspective of test automation tool design it is appropriate to look at test automation on three levels.

- User interface (UI) level (corresponds to S1 class). Activities performed by real user through some visual interface, either graphical (GUI, Graphical User Interface) or console (CUI, Console User Interface), are automated.
- Function call level (corresponds to S2 class). Automated activities emulate other modules (or systems) interacting with module (or system) under test by either direct function invocations or calls through some higher level interface, such as COM or CORBA.
- Communication level (corresponds to S3 class). Automated activities emulate other modules interacting with module under test through some communication protocol, e.g. network (Ethernet, TCP, HTTP, etc) or direct connection (COM, LPT, USB, etc).

Classification criterion S described in the first chapter of the thesis has the most impact on automated test tool design. Other criteria impact design details. For each of aforementioned levels the most important aspects of tool design are analyzed in the second chapter of the thesis.

When the tool is selected or developed automated test development can begin. If the number of tests is big, these test suites can be referred to as testware emphasizing that it is a kind of software with similar requirements for modularity, extensibility, maintainability etc. On the other hand testware is a specific kind of software with specific features.

According to [7] testware consists of:

- test set;
- script set;
- data set;
- utility set;
- test suites;
- testware library;
- test results.

Physical deployment of these elements depends on test automation tool applied as well as on choices of test designer. Test efficiency to high extent depend on which data are separated from scripts and which remain as script constants, which functionality is separated to test utilities and which resides in scripts themselves. Combination of tests into test suites depends on testing goals – which aspects of the systems have to be tested and how fast test execution should be.

According to [5] automated testware design process can be divided into two phases:

1) test requirement analysis which leads to development of test requirement matrix and possible testing techniques evaluation (steps: analysis of goals, selection of verification methods, testing requirements analysis, selection of test requirement matrix, mapping of testing techniques);
2) testware design which leads to test procedure definitions (specifications) with acceptable level of detail (steps: testware model definition, test architecture definition, automated or manual test mapping, test data mapping).

This process is described and analyzed in detail in the thesis, various test automation heuristics are described as well.

Several automated testing process models exist. M. Fewster and D. Graham propose process model consisting of five phases [7]:

1) testing condition identification;
2) test design;
3) test implementation;
4) test execution;
5) comparison of test outcomes.

E. Dustin et al. propose Automated Test Life-Cycle Methodology (ATLM) [5], consisting of six phases:

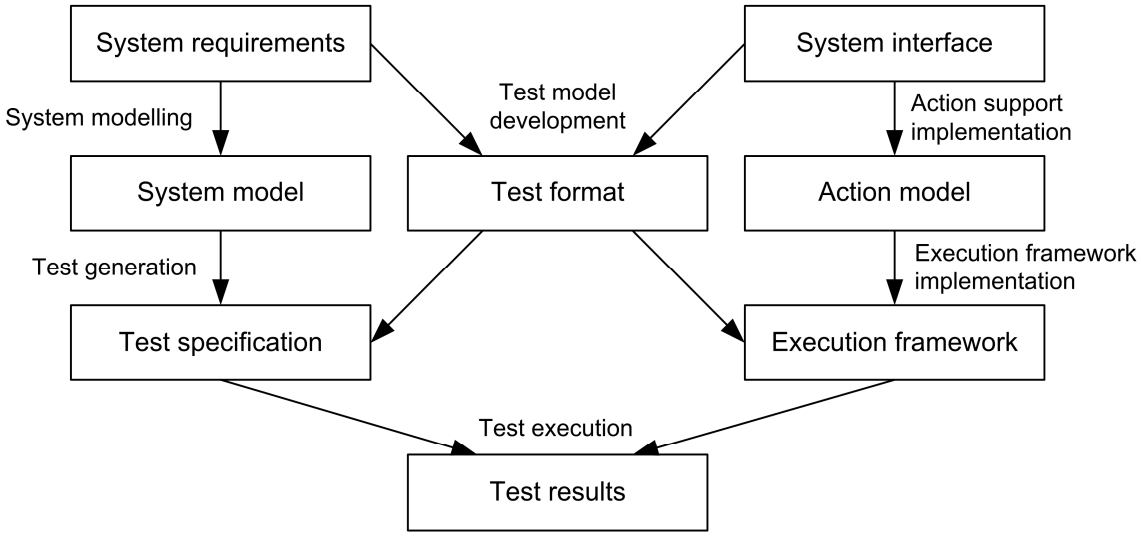
1) decision to automate test;
2) test tool acquisition;
3) automated testing introduction process;
4) test planning, design and development;
5) execution and management of tests;
6) process evaluation and improvement.

Comparing these two models it can be concluded that Fewster-Graham process model is narrower than ATLM and covers only 4^{th} and 5^{th} ATLM phases.

Both these models have a drawback: they do not take in account possibility of automated test generation and assume that all automated tests are developed manually. To eliminate this drawback unified automated testing model was developed in the thesis which includes possibility of test generation.

The elements of the model are activities and components. Components are work products created during testing process. Model activities are tasks performed either manually or automatically by which based on one components other components get created.

Relationships between model activities and components are illustrated in figure 2.4 where components are shown as rectangles and activities are shown as arrows.



2.4. fig. Unified automated testing model

Justification of model structure and possibilities of its application are described in the thesis.

Implementation of the model depends from various aspects such as testing goals, specifics of system under test etc. The model is flexible in the sense that it allows activities to be either manual or automated with an exception that test execution is assumed to be automated always, as the model is dedicated to automated testing. On the other hand action support implementation and implementation of execution framework are non-automatable activities.

Implementation of components and activities also depends on tool class (according to the classification described in the first chapter) used as an execution framework. For example, tools of D1 class, where test data are part of the script, test specification may be the script itself. Test generation can be implemented based on script templates, if the tool is of S1 or S2 class, i.e. the script is defined in programming or declarative language. The generator from a template can create several scripts corresponding to different test cases. The tool (execution framework) interprets these scripts as test specification, executes them and produces test results.

The model's implementation possibilities for tool of other classes are also reviewed in the thesis.

The model is applicable not only in the task of separate test generation and execution but also for generation and execution of test suites. In this case test specification takes form of test execution sequence, the generator creates the sequence according to properties of existing tests. Execution framework in this case is implemented as a test driver, which executes these sequences. This approach is described in more detail in the fourth chapter of the thesis where developed test suite generation solutions are described.

2.3. Automated test measures and efficiency evaluation

The goal of the testing process is defect detection. Therefore there should be a method allowing from potentially infinite set of conceivable tests to select a subset that could detect possibly more defects in limited testing time.

In the thesis the definition of testing efficiency is based on Pfleeger [23] definition which can be described with formula:

$$E = D/T, \tag{2.1}$$

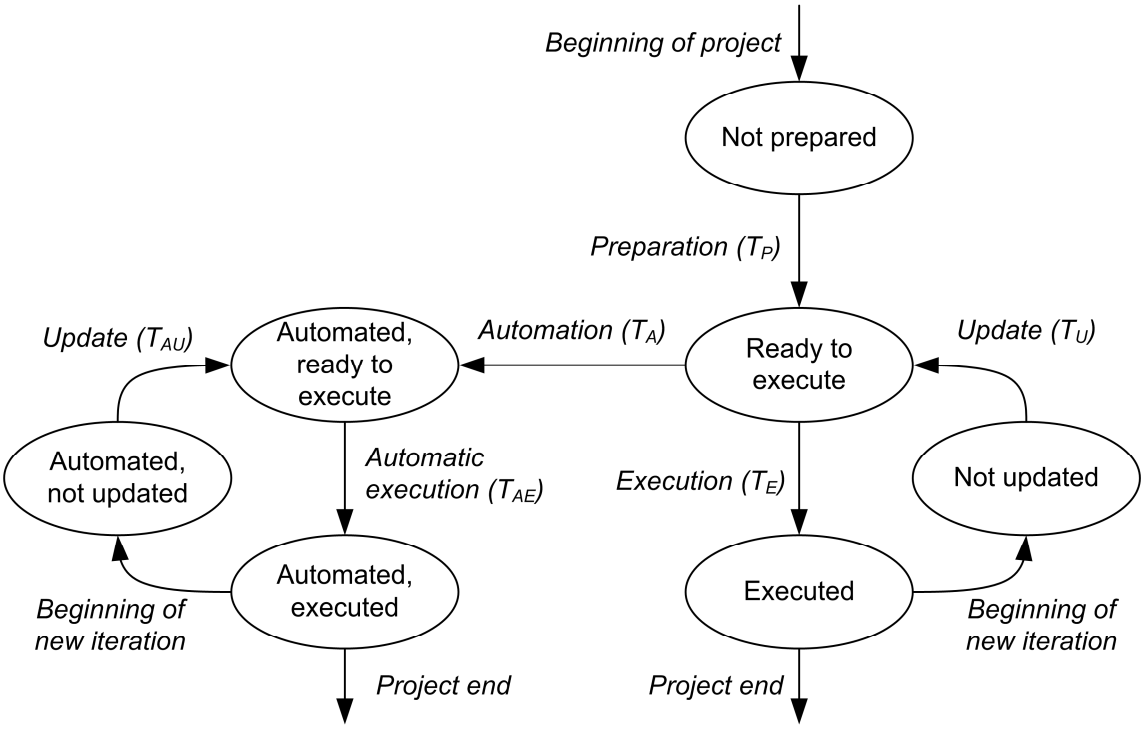
where

E – testing efficiency;

D – number of defects detected during testing;

T – time spend on testing in person-hours.

Test which gets repeatedly executed in several project iterations at every time moment is in one of states illustrated by diagram in figure 2.5. Notation T is used to denote time necessary to make the transition between states (for example to automate a test, to execute a test etc).



2.5. fig. Automated test states

This model can be significantly simplified based on practical considerations.

1. Execution of automated test does not require much human time and therefore can be assumed to be zero:

$$T_{AE} = 0. \tag{2.2}$$

2. Preparation and update of automated tests are more complex than preparation and update of corresponding manual tests. This complexity causes increase of time necessary for the case if automation is applied and this complexity would be appropriate to describe with automation complexity factor α (taking into account that preparation time of automated test include preparation time of a test as such and also time of its automation):

$$\alpha = \frac{T_P + T_A}{T_P} = \frac{T_{AU}}{T_U}. \tag{2.3}$$

The time necessary for test τ in i-th iteration depends on the state of the test at the beginning of iteration and also depends on whether the test is going to be automated during

the iteration. The formulas (assuming simplifications described) for determining time $T_{T\,i}(\tau)$ necessary for test τ in i-th iteration are shown in table 2.1.

2.1. table

Time $T_{T\,i}$ spent on a test during single iteration

State at the beginning of iteration	**State at the end of iteration**	
	Executed	Automated executed
Not prepared	$T_P + T_E$	αT_P
Not updated	$T_U + T_E$	$T_U + (\alpha - 1)T_P$
Automated, not updated	—	αT_U

When the time to be spent on test τ in i-th iteration and test significance $\vartheta_i(\tau)$ (which is calculated as a sum of risks of test's target defects) are known, it is possible to calculate predicted test efficiency $E_i(\tau)$ in the iteration. Taking formula (2.1) as a basis and reducing it to a case of single test in one iteration, number of defects should be replaced with test significance (predicted number of defect points which test will detect) and instead of total time of testing time to be spent on test in i-th iteration should be taken. As a result the following formula is obtained:

$$E_i(\tau) = \frac{\vartheta_i(\tau)}{T_{T\,i}(\tau)}\ , \tag{2.4}$$

where

$E_i(\tau)$ – efficiency of test τ in i-th iteration;

$\vartheta_i(\tau)$ – significance of test τ in i-th iteration;

$T_{T\,i}(\tau)$ – time spent on test τ in i-th iteration.

In the thesis formula (2.5) is obtained which allows to estimate predicted test automation relative efficiency (efficiency that can be obtained with automated test during first n iterations divided by efficiency of the same test if it does not get automated):

$$\mathcal{A}_n = \frac{1}{\alpha}\left(1 + \frac{nT_E}{T_P + (n-1)T_U}\right), \tag{2.5}$$

where

α – automation complexity factor;

T_P – preparation time of manual test;

T_U – update time of manual test;

T_E – execution time of manual test;

n – number of iterations for which the ratio is calculated.

$\mathcal{A}_n$ value greater than 1 means that during the first n iterations automated test reaches higher efficiency than the same manual test and vice versa. It is possible to state that if condition $\alpha T_U < T_U + T_E$ is satisfied then for some number of iterations (that can be fairly high) automation will justify itself. If this condition is not satisfied then automation of such test will never justify itself ($\mathcal{A}_n < 1$ for all n).

In the thesis a practical algorithm is proposed which allows to select test set ω for particular iteration with efficiency close to optimal.

1. At the beginning the set of selected test is empty $\omega = \emptyset$, but remaining iteration time is equal to total iteration time $T_{\text{rem}} = T$.
2. For each test τ from tests not yet selected for the iteration and for which $T_{T\,i}(\tau) < T_{\text{rem}}$, efficiency $E_i(\tau)$ has to be calculated by using data about significance of test τ and time necessary for it (according to table 2.1) considering automation possibility as well.
3. Test τ_{ef} should be selected for which efficiency was the highest, the test is added to the set ω and remaining time of iteration should be decreased by time necessary for test τ_{ef}: $T_{\text{rem}} \coloneqq T_{\text{rem}} - T_{T\,i}(\tau_{\text{ef}})$.
4. The process continues from step 2 until there are tests remaining which are not selected yet and for which $T_{T\,i}(\tau) < T_{\text{rem}}$.

This algorithm of test selection ensures that test set ω is selected with efficiency close to the highest possible for a single iteration. However if number of iteration is predicted to bi high the algorithm is too greedy. For example test automation rarely justifies itself in the same iteration when it gets done, benefits of automation show up later.

Therefore it can be reasonable to select tests in such way that efficiency of test set would be possibly high not for single current iteration but for next k iterations cumulative efficiency of the test would be close to the highest possible. In this case in the proposed algorithm step 2 has to be changed and it would be necessary to calculate not efficiency $E_i(\tau)$ but cumulative efficiency $E_{i:k}(\tau)$:

$$E_{i:k}(\tau) = \frac{\sum_{j=i}^{i+k-1} \vartheta_j(\tau)}{\sum_{j=i}^{i+k-1} T_{T\,j}(\tau)}\ , \tag{2.6}$$

where

$E_{i:k}(\tau)$ – cumulative efficiency of test τ in k iterations, beginning with i-th iteration;

$\vartheta_j(\tau)$ – significance of test τ in j-th iteration;

$T_{T\,j}(\tau)$ – time spent on test τ in j-th iteration.

So, the developed mathematical model of test efficiency allows to estimate predicted test efficiency in one particular iteration as well as in several test process iterations forward. The developed algorithm of efficient test set selection can help to achieve more rational use of testing time by concentrating on tests capable to detect defects with higher probability.

2.4. Development of automated test suite generation solution

Execution of one single test case is relatively simple process. If test set contain many test cases separate execution of each test case is inefficient. Therefore it is important to discover methods for execution automation of big test sets. The fourth chapter of the thesis is dedicated to alternative ways of test set execution and test driver design. As a result a library is developed implementing test suite generation from a test set.

Automated execution of test set can be classified by two parameters:

1) by execution activation type – test set execution can be started manually or automatically;
2) by result type – test set execution can produce raw results, correspondence to the expected results or oracle verification.

To execute a test set all test cases belonging to the test set should be executed by running corresponding test scripts. Software module responsible for this activity is denoted as test driver. Test drivers can be implemented in various ways but all implementations can be divided into two different categories:

1) static – execution sequence of test cases is strictly defined by test driver developer;
2) dynamic – sequence of test cases is not defined directly, automated planning algorithm of a test driver constructs the sequence automatically from execution conditions of separate test cases.

In the thesis these two categories are analyzed in detail. Static test drivers are divided into linear and structural, dynamic drivers can be based on dependencies between tests, on dependencies with constraints, on states or on state systems.

For the task of automated test suite generation (test sequence constructions from separate test cases) a library TSGL (Test Suite Generation Library) was developed. According to notation of unified automated testing model described in the second chapter of the thesis TSGL implements additional features of test generator and execution framework. Its output can be test execution sequence described in XML format which in this case plays a role of test specification, also the library implements functions allowing to call tests in this sequence. The input for the library is unstructured test set and test metadata where information about each test's beginning and end states is described. Implementation of the tests themselves and functionality of their execution is not a feature of TSGL and other tools are necessary for these tasks, for example any of those described in the first chapter of the thesis.

TSGL library is based on dynamic test driver which works according to information about states of the program under test. The main component of the driver is test sequencer which from a given test set creates a test sequence in such way that the next test begins at the same state where the previous test finished. Two sequencer types are implemented in the library:

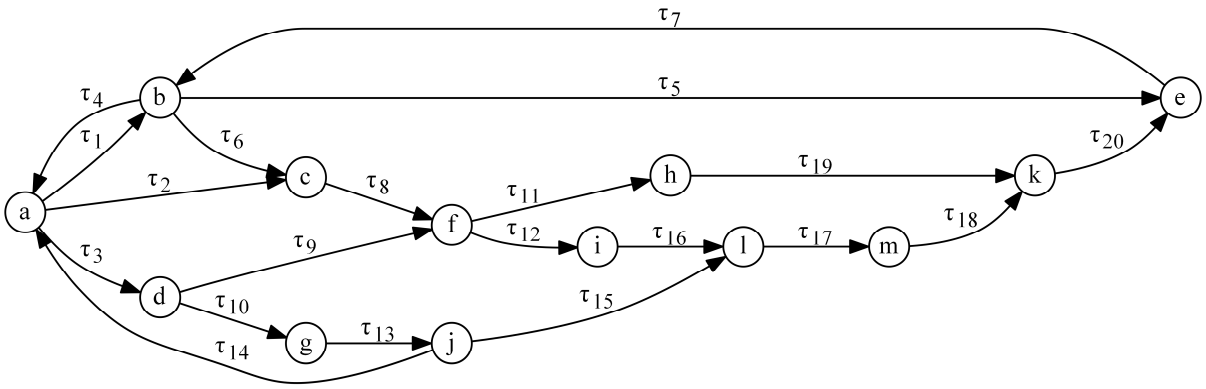
1) based on simple state systems;
2) based on compound state systems.

The input of the test sequencer is a test set [31]. However from test sequencer point of view test content is not as important as test metadata:

- beginning state of a test case where test execution begins;
- end state of a test case where test case execution finishes.

Activity of the sequencer consist of the following steps:

1) for each test case it reads information about its beginning and end states;
2) in memory it constructs a state transition graph where vertices correspond to states of the program, but edges corresponds to test cases τ_i as illustrated in example in figure 2.6;
3) constructs a path in the graph which begins in a given beginning state of the system, contains all edges of the graph and finishes in the same beginning state of the system; this path corresponds to the result test suite or more precisely the sequence of edges in the path corresponds to the sequence of test cases in a test suite.



2.6. fig. Example of a state transition graph

Although shorter test suites (those containing smaller number of edges) are better, it is not necessary to select the shortest path because if test suite will be executed automatically it will not spend any human time. However test suite should be acceptably short.

Test suite generation method based on simple states is limited especially in case of user interface level automated tests. Compound states can be modelled as sets of name-value pairs [28]. These name-value pairs will be denoted as state components. State components which are important for one test can be irrelevant for another. There are three possible awareness types of test with respect to values of state components. Awareness types will be denoted as follows:

- $R(x, y)$ – the test requires certain state component value x as a precondition and changes it to value y as a postcondition;
- $S(y)$ – the test does not require any specific value of state component as a precondition (it can work with any one), but after test execution the state component changes its value to y as a postcondition;
- U – test does not require and does not change the value of state component.

Two tests τ_1 and τ_2 can be sequenced into subpath $<\tau_1, \tau_2>$ if for each state component there is no conflicts between postconditions of test τ_1 and preconditions of test τ_2. This subpath will also have its preconditions, postconditions and awareness. Using the same principle subpath can be sequenced with other test or with other already existing subpath producing longer subpath. If separate test can be viewed as a subpath consisting of single test, the sequencing operation is defined on subpath set.

Table 2.2 contains rules by which from awareness of subpaths C_1 and C_2 awareness of resulting subpath $C_3 = <C_1, C_2>$ can be determined.

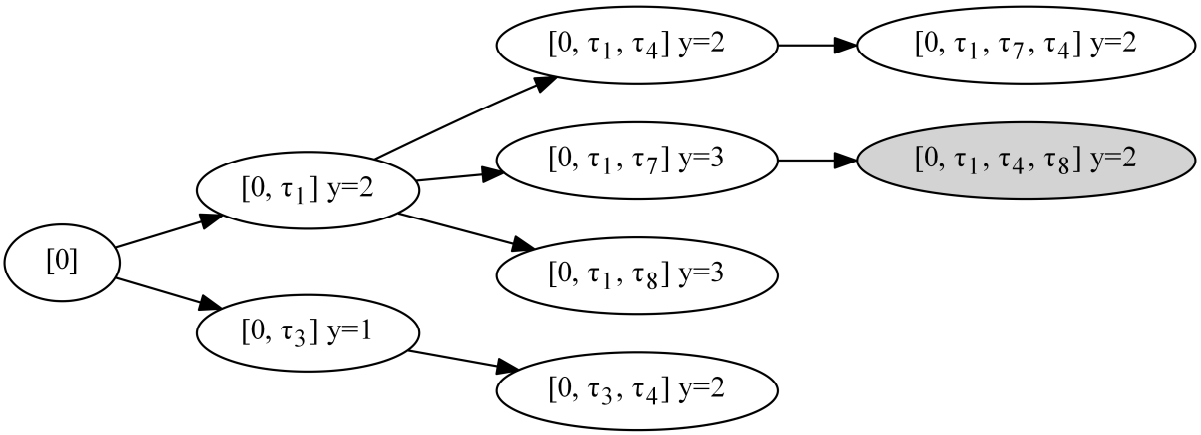
2.2. table

Subpath sequencing rules

Left subpath awareness	**Right subpath awareness**		
	***R*(x_2, y_2)**	***S*(y_2)**	***U***
R(x_1, y_1)	*R*(x_1, y_2), if $y_1=x_2$ otherwise – impossible	*R*(x_1, y_2)	*R*(x_1, y_1)
S(y_1)	*R*(x_2, y_2), if $y_1=x_2$ otherwise – impossible	*S*(y_2)	*S*(y_1)
U	*R*(x_2, y_2)	*S*(y_2)	*U*

TSGL implements a method of test suite generation based on subpath tree.

1. Fictitious "zero test" is used as a tree root. Zero test models the beginning state of the system guarding initial values of all state components. Zero test has awareness *R*(*x*, *x*) for each state component, because it is necessary for test suite to end at the same state where it begins. Each tree node represents some subpath.
2. Every next level of the tree is generated from the previous one. For each node of the level it is necessary to find tests which can be sequenced after the subpath of the node. New subpaths (longer by one element) become nodes of next tree level – children of current node. This process is illustrated in figure 2.7 where state consists of one state component *y* for simplicity of the example.



2.7. fig. Example of subpath tree fragment

The process is repeated until subpath is found which contains all tests of a test set and ends with zero test. The algorithm has an exponential complexity and in practice when

number of tests is big the method in its pure form is not suitable. In the thesis heuristics implemented in TSGL are described that make this algorithm more practical.

The developed TSGL library can be used to automatically generate test suites from a test set in cases when dependencies between tests can be represented as simple or compound state systems.

2.5. Development of performance testing tool Picus

Software performance testing is an important part of multiuser system quality estimation. In order to test system performance it is necessary to gather performance measures in conditions of simultaneous work of many users.

Two basic functions which should be implemented in a tool of this class are:

- emulation of real user activities as well as ability to emulate simultaneous work of these users;
- measurement of system reaction time and correctness.

During the work on the thesis a performance testing tool Picus was developed. The main feature of Picus is its support of extensions [27]. It supports application programming interfaces (API) for:

1) protocols (allow communication level scripts to interact with different kinds of systems);
2) schedules (determine how a load changes over test time);
3) statistics aggregators (determine a strategy how statistical data of test results get aggregated, passed to higher levels, and how end results are produced);
4) communicators (determine how the core component console interacts with load agents which allow to scale test capacity).

According to classification described in the first chapter of the thesis Picus belongs to classes D3 (supports data tables), S1a-Java (uses Java programming language for script development), M2 (supports parallel script execution), I3-HTTP (works on protocol level, currently supporting HTTP). According to the unified automated testing model described in the second chapter of the thesis Picus is a typical execution framework implementation which can be adapted to interpret different forms of test specifications.

In the thesis Picus structure, peculiarities of implementation and usage are described and justified in detail.

The classical approach to creating workload in performance tests is to design load schedule of the test before its execution. Accurate design of both test scripts and load schedules is important for reaching the test goals [26]. In this approach load schedule specification is an input to a component of load agent which is denoted as scheduler. Scheduler is responsible for workload change during test according to specified load schedule. During the test performance measures are gathered forming test results.

In some cases design and use of fixed load schedules is sufficient and even the best method. For example if the goal is to evaluate various performance metrics under different load levels, a schedule should be designed in which number of virtual users gradually increases from zero to designed system capacity. The test then has to be executed and results have to be processed to get the gathered performance measures as functions of load level (number of virtual users) [3]:

$$p = F(v), \tag{2.7}$$

where

p – value of performance measure;

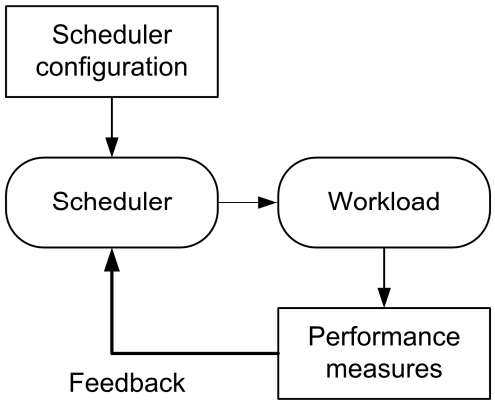
v – number of virtual users;

F – function being investigated.

Existence of such functions and their representation in graphical form can be valuable for system bottleneck detection as well as for evaluation of whether real capacity of the system under test corresponds to designed one.

The drawback of this approach is relatively long time necessary to execute the test. If test goal is narrower than aforementioned one it can be inefficient.

In the thesis other method is proposed in which the scheduler automatically adapts workload in order to more quickly identify point v_0 for a given p_0 such that $F(v_0) \geq p_0$ and $F(v_0 - 1) < p_0$ (if F is nondecreasing). In other words this method is applicable if it is necessary to identify load level at which some performance measure of interest reaches given value p_0. The method is illustrated in figure 2.8.



2.8. fig. Adaptive load schedule model

In this model scheduler is not just executer of pre-specified load schedule but is active decision maker about what to do next [29]. Decisions made by scheduler depend on two factors.

1. Configuration of scheduler. These parameters define targets to reach during test time and which planning algorithm is able to handle.
2. Performance measures which have been gathered at the moment of decision making. Scheduler can analyze measures gathered previously and base its decisions about workload change on these measures.

The main difference of this model from the model of fixed load schedule is feedback received by scheduler in a form of performance measures.

Simple adaptive scheduler algorithm is described below which is suitable for reaching various goals of performance testing. Without loss of generality it is assumed that function $F(v)$ is monotone nondecreasing. The algorithm consists of two phases.

The goal of the first phase is to detect an interval to which point v_0 belongs. To make it happen quickly number of users gets increased exponentially in time by iteratively doubling number of virtual users.

1. Begin with one virtual user ($v = 1$) and measure $F(v)$.
2. While $F(v) < p_0$ double number of virtual users ($v := 2v$) and re-measure $F(v)$.
3. When value of v is reached for which $F(v) \geq p_0$ stop, it means that v_0 is in interval $[v/2, v]$.

The goal of the second phase is to gradually narrow the detected interval $[v_{\text{low}}, v_{\text{high}}]$:

1. If $v_{\text{high}} - v_{\text{low}} > 1$ for point $v := v_{\text{low}} + (v_{\text{high}} - v_{\text{low}})/2$ measure $F(v)$.
2. If $F(v) \geq p_0$ then assign $v_{\text{high}} := v$, otherwise $v_{\text{low}} := v$.

3. Repeat from step (1) until $v_{\text{high}} - v_{\text{low}} = 1$. At this moment v_{high} holds v_0 value.

In other words in the second phase binary search algorithm is applied to find v_0 in interval $[v_{\text{low}}, v_{\text{high}}]$. Number n of evaluations of function $F(v)$ in the described algorithm can be estimated with formula:

$$n = 2\lceil \log_2 v_0 \rceil, \tag{2.8}$$

where

v_0 – number of virtual users to be found.

As number of estimations of function $F(v)$ is logarithmically dependent on v_0, efficiency of the algorithm is higher comparing to fixed load schedule approach where number of estimations is linearly dependent on v_0.

In order for algorithm to be effective some other aspects should be taken into account:

- After load increase the scheduler should wait until all newly added virtual users get into normal work rhythm and only then necessary performance measures should be gathered. The wait time should be at least two times longer than single script execution requires.
- After load decrease the scheduler should wait until all stopped virtual users correctly finish their activities and the system stabilizes after load which was generated these users; only then necessary performance measures should be gathered.

Script execution time means time necessary for execution of all script's transactions and sum of all wait times between transactions. These precautions are necessary to better measure effect of load and eliminate noise in measured data which can be introduced by workload change effects.

It is necessary to mention that the described algorithm can be inappropriate if there are other factors which can affect performance measures more than workload level generated by load agents. In this case function $F(v)$ is stochastically fluctuating and assumption that it is monotone nondecreasing is not satisfied.

In the thesis it is experimentally shown that for specific performance testing tasks application of adaptive scheduler allows to reach substantially better time efficiency by decreasing test execution time from linear to logarithmic.

2.6. Application of the developed solutions

In the sixth chapter of the thesis it is described how solutions described in previous two chapters were applied in real projects by providing outsource testing services. Customers in these projects was several Latvian banks and public institutions. Large multiuser web-based systems were tested.

Developed test suite generation solutions and TSGL library was applied in two testing projects of "Centre of New Technologies" Ltd. TSGL library was applied in just two projects because the functionality it implements becomes necessary only in very non-trivial cases. In simple test automation tasks usually static test drivers are sufficient enough and TSGL application would inadequately complicate produced testware.

However in one testing project the task of automation was complex enough for TSGL introduction to provide positive effect. The essence of the task was as follows: it was necessary to verify if users have permissions to perform those and only those activities in the system which correspond to their roles (levels of permissions) in the system. Complexity of the task can be described by following values:

- number of activities to verify: about 120;
- number of user roles: about 45.

System integrity was a critical factor and therefore all combinations of activities and roles had to be verified. In manual testing case verification of single activity for single user role would require 10 minutes, so verification of all combinations would require 900 person-hours or 5.6 person-months.

With test automation it was possible to reduce testing time of a single combination to about one minute. In this case full retesting would take 90 hours. Such time was still unacceptable as it was expected that discrepancies would be found, fixed and the full test should be repeatedly executed.

By simplifying test scripts and allowing TSGL to automatically generate test suites time necessary for full test reduced to 30 hours. When execution of the test was parallelized to three computers total time of execution decreased to 10 hours and this execution time was acceptable. Because of detected discrepancies defects introduced with new changes it was necessary to repeat full test 4 times, therefore usage of TSGL was fully justified.

In this test algorithm of test suite generation based on compound states was used. State components were following:

1) user role – level of permissions of user currently logged in;
2) active window – system window currently active un user's desktop.

The main role of TSGL in this project was elimination of script activities required only to open necessary windows and to return to base system state after test is done. It was sufficient to specify at which window test should begin and end but the correct sequence of execution was automatically selected by TSGL thus reducing execution times.

In the second project where TSGL was applied it was used together with Picus providing ability to generate sequence of virtual user activities. Use of TSGL was justified with a fact that number of activities virtual users had to perform was large (about 50) and possibility to perform the activities depended on current state of the system. In this project positive effect of TSGL application was not as high as in the first project because number of activities was moderate.

From the moment of development of its first version Picus has been applied in 12 testing projects of "Centre of New Technologies" Ltd. All projects were concerned with testing of web-based systems and existing Picus module of HTTP protocol support was used. To illustrate variety of projects four of them are described in detail.

1. The system under test was developed using Java language and Struts framework. Tomcat was used as an application container and web server. Two Picus scripts were developed. One script had to send unique data of specific format to the system, therefore Picus was extended to support data tables during this project. It was implemented to support data tables in CSV form from which script can take one data row when it is necessary. For big enough tables it is guaranteed that script takes unique data set each time.
2. The system under test was developed using Oracle tools and Oracle Web Application Server was used as a web server. Five scripts were developed which performed data filtering activities of different type. Large data amounts were used in HTTP requests with unusual encoding, therefore Picus HTTP module was extended for this precedent to support such requests.
3. The system under test was developed using Ruby-on-Rails and Microsoft Infopath technology combinations. MS IIS was used as a web server. To ensure emulation of interaction with Infopath it was necessary to introduce in Picus mechanism to enable exchanging XML data over HTTP protocol. Web cookie handling mechanism as it was implemented at that moment was not compatible with the system under test, therefore cookie handling was extended and improved in Picus HTTP module.

4. The system under test was developed using Java and JSP languages. Tomcat was used as an application container and as a web server. For this system it was not necessary to introduce changes in Picus as existing functionality was already sufficient.

It can be seen from the examples that as a result of several real projects Picus HTTP module was improved in various aspects. Currently it is sufficiently stable and for the latest projects it was applied without modifications what means it is ready to be used for testing of almost any web-based system.

All aforementioned as well as other projects were successfully finished and took from one to three weeks from the moment of fixing testing requirements until test summary report preparation. Technical test preparation activities including script development, Picus adaptation and configuration of test parameters took from two to five work days. The acquired experience shows that Picus tool is well suited to its purpose. Time required for projects was similar to time spent on similar projects where other tools were used or even better. It can be explained with Picus adaptability to non-trivial situations.

3. RESULTS OF THE THESIS

The goal of the thesis was to develop software testing automation methods which would allow to achieve higher automated testing efficiency. As a result of the work automated test suite generation methods and adaptive performance testing method were developed and applied in real projects.

The main achievements of the thesis.

1. Test automation tool classification is developed, 32 test automation tools are classified according to the developed classification allowing to choose the most appropriate tool for testing specific software.
2. Unified automated testing model is developed allowing to identify potentially automatable activities in software testing. The model is applicable in various test automation contexts and differently implementable depending on tool class.
3. Mathematical model of automated and manual test efficiency evaluation is developed. Based on the model effective test set selection algorithm is developed which can be used to which can be used to select tests based on defect risk and testing time estimations. The tests get selected in a way allowing to detect more defects with higher probability in limited time.
4. Two automated test suite generation methods are developed: based on simple and compound state systems. The methods are implemented in TSGL library which was applied in two real projects.
5. Performance testing tool Picus is developed special with its extensibility support. Picus was applied in 13 real projects.
6. Adaptive performance testing method is developed and implemented as Picus add-on.

The developed solutions have a practical value as they can be used in real software development projects to improve efficiency of automated testing. The developed tools and experience of their use gained in projects show that the solutions are usable for practical tasks and in many cases useful.

4. APPROBATION OF THE THESIS

4.1. Conference presentations

The results of the thesis were presented at the following international conferences:

1. 50th International Scientific Conference of Riga Technical University, 12-16 October 2009, Riga, Latvia.
2. 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009), Workshop on Intelligent Educational Systems and Technology-enhanced Learning (INTEL-EDU), 7 September 2009, Riga, Latvia.
3. International Conference on Advanced Learning Technologies (ICALT-2009), 15-17 July 2009, Riga, Latvia.
4. 49th International Scientific Conference of Riga Technical University, 13-15 October 2008, Riga, Latvia.
5. International Conference on Information Technologies (InfoTech-2008), 19-20 September 2008, Varna, Bulgaria.
6. International Conference on Engineering Education & Research (ICEER 2007), 2-7 December 2007, Melbourne, Australia.
7. 48th International Scientific Conference of Riga Technical University, 11-13 October 2007, Riga, Latvia.

4.2. Scientific papers

The results of the thesis are published in the following scientific papers:

1. Sukhorukov A. Automated Test Chaining Based on Compound States // Proceedings of the International Conference on Information Technologies (InfoTech-2010). – Sofia, Bulgaria: Publishing House of Technical University, 2010. – pp. 127–132.
2. Sukhorukov A. Self-Directed Performance Testing // Scientific Journal of Riga Technical University. Series 5 – Computer Science. Applied Computer Systems. – 2010. – Vol. 43. – pp. 84–89.

3. Сухоруков А. Целенаправленное обучение на примере модели и классификатора инструментов автоматизации тестирования ПО // Образовательные технологии и общество. – Казань, Татарстан, РФ: Казанский государственный технологический университет, 2010. – Том 13, № 1. – стр. 370–377.
4. Sukhorukov A. Test Case Generation for Validation of E-Learning Course // Advances in Databases and Information Systems, 13th East-European Conference, ADBIS 2009 Associated Workshops and Doctoral Consortium. Local Proceedings. – Riga, Latvia: Riga Technical University, 2009. – pp. 230–237.
5. Sukhorukov A. Architecture for Automated Validation of E-Learning Courses // Proceedings of the Ninth IEEE International Conference on Advanced Learning Technologies (ICALT-2009). – Washington, DC, USA: IEEE Computer Society, 2009. – pp. 152–153.
6. Sukhorukov A. Problems of Test-Driven Aspect-Oriented Development // Scientific Proceedings of Riga Technical University. Series 5 – Computer Science. Applied Computer Systems. – 2009. – Vol. 38. – pp. 180–186.
7. Sukhorukov A. Performance testing tool Picus // Proceedings of the 22nd International Conference on Systems for Automation of Engineering and Research (SAER-2008). – Bulgaria: King, 2008. – pp. 165–172.
8. Sukhorukov A., Zaitseva L. Automated test state management framework // Scientific Proceedings of Riga Technical University. Series 5 – Computer Science. Applied Computer Systems. – 2008. – Vol. 34. – pp. 215–224.
9. Sukhorukov A., Zaitseva L. Applicability of Automated Testing to E-Learning Software Validation // Proceedings of the International Conference on Engineering Education & Research (ICEER 2007). – Melbourne, Australia: Victoria University, 2007. – pp. 1–6.

BIBLIOGRAPHY[1]

1. Adrion W., Branstad M., Cherniavsky J. Validation, Verification, and Testing of Computer Software // ACM Computing Surveys. – 1982. – Vol. 14, No. 2. – pp. 159–192.
2. Aichernig B. et al. State of the Art Survey - Part a: Model-based Test Case Generation. Technical Report 1-19a on project MOGENTES. – Graz University of Technology. – 2008.
3. Amza C. et al. Specification and implementation of dynamic web site benchmarks // 5th IEEE Workshop on Workload Characterization (WWC-5). – IEEE Press, 2002. – pp. 3–13.
4. Boehm B. W. Seven Basic Principles of Software Engineering // Journal of Systems and Software. – 1983. – Vol. 3, No. 1. – pp. 3–24.
5. Dustin E., Rashka J., Paul J. Automated Software Testing: Introduction, Management and Performance. – Boston, MA, USA, 1999. – 608 p.
6. Ferguson R., Korel B. The chaining approach for software test data generation // ACM Transactions on Software Engineering and Methodology. – 1996. – Vol. 5, No. 1. – pp. 63–86.
7. Fewster M., Graham D. Software Test Automation: Effective Use of Test Execution Tools. – New York, NY, USA: ACM Press/Addison-Wesley, 1999. – 596 p.
8. Futatsugi K. et al. Principles of OBJ2 // Proceedings of the 12th ACM Symposium on Principles of Programming Languages. – 1995. – pp. 21–28.
9. Gelperin D., Hetzel B. The Growth of Software Testing // Communications of the ACM. – 1988. – Vol. 31, No. 6. – pp. 687–695.
10. Godefroid P. Compositional dynamic test generation // Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 2007. – pp. 47–54.
11. Gupta N., Mathur A., Soffa M. Automated test data generation using an iterative relaxation method // Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering. – 1998. – pp. 231–244.
12. Hamill P. Unit Testing Frameworks. – Sebastopol, CA, USA: O'Reilly, 2004. – 304 p.

[1] Summary of the thesis contains shortened list of references. The thesis contains 130 bibliography sources.

13. ISO 8807:1989. Information processing systems – Open Systems Interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. – 1989.
14. ITU-T Recommendation Z.100 (11/99). Specification and Description Language (SDL). – 2000. – 246 p.
15. King J. Symbolic execution and program testing // Communications of the ACM. – 1976. – Vol. 19, No. 7. – pp. 385–394.
16. Korel B. Automated software test data generation // IEEE Transactions on Software Engineering. – 1990. – Vol. 16, No. 8. – pp. 870–879.
17. Leavens G. T., Baker A. L., Ruby C. Preliminary design of JML: A behavioral interface specification language for Java // ACM SIGSOFT Software Engineering Notes. – 2006. – Vol. 31, No. 3. – pp. 1–38.
18. Lewis W. E. Software Testing and Continuous Quality Improvement. 3rd ed. – Auerbach Publications, 2008. – 688 p.
19. Marick B. When should a test be automated // Proceedings of The 11th International Software/Internet Quality Week. – 1998. – pp. 1–20.
20. Meyer B. Eiffel: The Language. – Prentice Hall, 1991. – 300 p.
21. Mosley D. J., Posey B. A. Just Enough Software Test Automation. – Prentice Hall, 2002. – 288 p.
22. Mosses P. D. CASL: A guided tour of its design // Proceedings of WADT'98. – Springer-Verlag, 1999. – pp. 216–240.
23. Pfleeger S. L. Software Engineering: Theory and Practice. 2nd ed. – Prentice Hall, 2001. – 659 p.
24. Pressman R. S. Software Engineering: A Practicioner's Approach. 4th ed. – McGraw-Hill, 1997. – 852 p.
25. Runeson P. A survey of unit testing practices // IEEE Software. – 2006. – Vol. 23, No. 4. – pp. 22–29.
26. Subraya B. M. Integrated approach to web performance testing: A practitioner's guide. – PA, USA: IRM Press, 2006. – 368 p.
27. Suhorukovs A. Veiktspējas testēšanas rīka Picus izstrāde un pielietošanas iespējas // Latvijas IT uzņēmumu 9. konference „Testēšanas teorija un prakse": konferences materiāli. – Rīga, Latvija, 2008. – lpp. 31. –39.

28. Sukhorukov A. Automated Test Chaining Based on Compound States // Proceedings of the International Conference on Information Technologies (InfoTech-2010). – Sofia, Bulgaria: Publishing House of Technical University, 2010. – pp. 127–132.
29. Sukhorukov A. Self-Directed Performancd Testing // Scientific Journal of RTU. Series 5. – 2010. – Vol. 43. – pp. 84–89.
30. Sukhorukov A., Zaitseva L. Applicability of Automated Testing to E-Learning Software Validation // Proceedings of the International Conference on Engineering Education & Research (ICEER 2007). – Melbourne, Australia: Victoria University, 2007. – pp. 1–6.
31. Sukhorukov A., Zaitseva L. Automated test state management framework // Scientific Proceedings of Riga Technical University. Series 5. – 2008. – Vol. 34. – pp. 215–224.