Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 1533–1538
DOI: 10.15439/2015F170
ACSIS, Vol. 5

Smart Technologies for Improved Software Maintenance

Zane Bicevska
DIVI Grupa Ltd
Riga, Latvia
Email: Zane.Bicevska@di.lv

Janis Bicevskis
University of Latvia
Riga, Latvia
Email: Janis.Bicevskis@lu.lv

Ivo Oditis
DIVI Grupa Ltd
Riga, Latvia
Email: Ivo.Oditis@di.lv

Abstract–***Steadily increasing complexity of software systems makes them difficult to configure and use without special IT knowledge. One of the solutions is to improve software systems making them "smarter", i.e. to supplement software systems with features of self-management, at least partially. This paper describes several software components known as smart technologies, which facilitate software use and maintenance. As to date smart technologies incorporate version updating, execution environment testing, self-testing, runtime verification and business process execution. The proposed approach has been successfully applied in several software projects.***

Keywords-Autonomic computing, smart technologies, self-managing systems, software maintenance

I. Introduction

Rapid development of information technologies has created systems of unprecedented complexity; some authors [1] refer to as „computing systems with complexity approaching boundaries of human ability". They indicate that the ultimate dream of a pervasive computing – billions of computing systems simultaneously connected to the internet – can quickly become unmanageable and may soon turn into evil "nightmare". The authors predict even further increase of information systems' complexity that would almost eliminate human ability to perform software installation, configuration, optimization and maintenance.

Solution to this problem certainly lies within prospects of information technologies. In complex systems operations that are too sophisticated to be managed by a regular user should be entrusted to the system itself. This can be executed by implementing respective components into software and setting environment, in which the system is used.

IBM has proposed a solution described in its autonomic computing manifesto in 2001. The main statement implies targeted development of information systems that were able to self-management thus overcoming gap between users and increasingly complex world of information technologies.

The research leading to these results has received funding from the research project "Information and Communication Technology Competence Center" of EU Structural funds, contract nr. L-KC-11-0003 signed between ICT Competence Centre and Investment and Development Agency of Latvia, Research No. 1.5 "Platform for business process description and modelling in event-oriented systems".

The manifesto listed four aspects of autonomic computing:

- Self-configuration - automated configuration of components and systems follows high-level policies, rest of system adjusts automatically and seamlessly;
- Self-optimization - components and systems continually seek opportunities to improve their own performance and efficiency;
- Self-healing - system automatically detects, diagnoses, and repairs localized software and hardware problems;
- Self-protection - system automatically defends against malicious attacks or cascading failures.

Achievements of autonomic computing movement during its first decade after publication of the manifesto have been explicitly demonstrated in [2], as well as in [3]. As of now, manifesto's targets have been met only to some extent.

The concept of smart technologies was created by authors [4], and its main objectives are similar to those of autonomic computing. The approach contains a set of practically applicable improvements of non-functional features to simplify the maintenance and daily use of information systems. Below are described five types of smart technologies, which need was identified in real software development projects. The proposed smart technologies cover only part of requirements outlined in the autonomic computing manifesto. Nevertheless they are suitable for practical implementations and can serve as valuable improvement of new and existing software systems.

The second chapter of this paper deals with related research and solutions. The third chapter describes the proposed architecture of smart technologies.

II. Related Works

The autonomic computing manifesto declares a vision of fully independent computer systems (not just software) that are able to self-management. It also defines evaluation criteria to check the maturity of autonomic systems [5] - from basic level (manually maintainable information systems) to completely autonomic systems that are able to function operate accordingly to guidelines set by humans.

The manifesto does not include any instructions about implementation issues, but some authors discuss ideas about essential components of autonomic systems. For instance R. Sterritt [6] describes an autonomic environment consisting of autonomic elements, which are mutually connected via

978-83-60810-66-8/$25.00©2015, IEEE

autonomous channels. Every autonomic element has a kernel, so called manageable component (the component implementing the business logic), and it is controlled by an "autonomous supervisor". The supervising component uses sensors and effectors, and its main functions are monitoring of internal and external states, accumulation of knowledge base and communication with other autonomic components using autonomous communication channels. A separate component in this system is so-called "heartbeat monitor" which communicates with any existing system components through autonomous communication channels and supervises the system as a whole.

The autonomic computing approach has also been criticized [7], and the main reasons are as follows:

- the lack of precise definitions;
- avoidance of the real complexity of the problem;
- ignoring of inter-componential links.

Despite these criticisms, autonomic system objectives are so attractive that there seemed to be no reason to abandon the ideas. In 2003 IBM extended the list of autonomic aspects to eight characteristic aspects [1]. The initial autonomic characteristics were enhanced by system's ability to "know itself" and manage its resources in a proper way. An autonomic system should know its environment as well as the context surrounding its activity and act accordingly – to adjust and operate in heterogeneous environment accordingly open standards - , as well as anticipate the optimized resources needed while keeping its complexity hidden. Some years later the, so called, self-management features were supplemented with new self-properties reaching a total of 24 features [3]. Continuing efforts on autonomic systems include both, theoretical research and practical implementation [2].

The concept of smart technologies created by authors [4] is consistent with the primary objective of autonomic computing. Unlike the traditional implementation of autonomic computing where universal autonomous software components are built, the smart technologies approach deals with embedding of specific system features into information systems directly but in a uniform way.

Although the smart technologies approach and the autonomic computing approach seemingly share some similarities, it should be emphasized that the smart technologies approach was developed independently. The practical results gained in IT projects provide evidence of the usefulness of the approach.

III. Components of Smart Technologies

There are five fields of smart technologies where practical results were gained: embedded software versioning and data syncing, embedded dynamic business model, testing of external environment, self-testing, and runtime verification.

A. Software Versioning and Data Syncing

Every successful software solution is being used and improved significantly longer than the development of its first version has taken. Information systems are in use for many years, and the software is gradually modified, updated with new features, improved to approximate to user needs.

To ensure reliability of software in long-term, the system should already in its initial development time include not only the required (customer specified) functionality, but also supporting mechanism – "updater" (see Fig. 1) for software, data structures and templates upgrading.

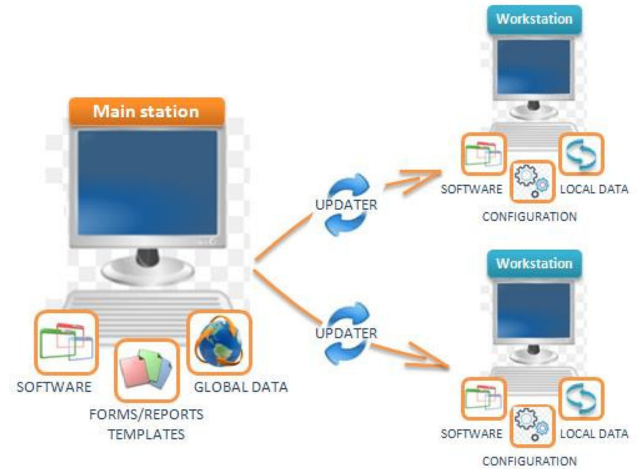

Fig. 1 Software versioning

The supporting mechanisms should be built into systems, and they should include features for deploying of new versions without any user intervention. The following should be ensured automatically during deployment process:

- check the compliance of the new software version with the external environment
- download and install a new software version
- update configuration and information about data structures, screen forms, report templates etc.
- migrate stored data into the new data structures of the database as well as the personalization and configuration data
- perform self-testing of the new system's version to check correctness of the essential system's functionality
- create backups to be able to recover the system in case of incidents

The majority of information systems today support some of the characteristics listed above, but in most cases - to a limited extent only. Authors of this paper have prototyped the characteristics in some projects, the research results are described in [8].

B. Execution environment testing

One of the most spectacular smart technology solutions is described by the authors in [9]. It is quite common that programs have specific requirements for their successful operation at a given environment – the computer, network, operating system, etc. The proposed solution implies gathering these requirements in a "software profile" to be able to validate the execution environment before starting the information system. Such validation should be performed on

demand, for instance, before each session; however, some authors propose validation during installation. Validation of execution environment allows avoiding failure in business processes in case the information system relies on properties of the environment.

Quite often, software is developed based on assumptions about other component's work, not on their specification [10], [11]. Similarly, developers sometimes assume that software, which works in development environment, will keep working after it is deployed elsewhere, hence encoding some assumptions about the environment into the program. As a result, when the software is installed in other environment, which is different from the development environment, the software may fail or work only partially correct.

The authors [12] propose a technology, which allows independent environment checks, performed by the software in order to validate if the execution environment is suitable for normal execution (see Fig. 2). Unlike the built-in test method, which validates the ability of software itself to fulfill its "contracts"; this technology measures livability of the external conditions. Only if the results of all checks are satisfactory, the program can be considered prepared for work at a given environment, otherwise the session is stopped, giving the user an explanation, why it is not possible to perform work.

A program execution profile is a document achieved when all the requirement descriptions of software are combined together. The profile can be formalized as a separate document and supplemented to typical software deliverables such as code and documentation. The main, but not the only use of the profile is validation of execution environment during program use.

The practical environment testing task is carried out by environment validation modules. Each module is an atomic unit, which enforces validation of a single type of requirement; this is done by reading information from the environment and comparing it to reference values. In a simple scenario, each requirement describes required value of some resource's attribute (for instance, data base server must be reachable). When the testing functionality of the module is invoked, it uses the information available in execution environment to do the "inspection".

To be able to modify the set of checks to be performed without modifying the program code, information about the checks (both the algorithms and reference values) must be stored outside the code. This concept is different from other approaches used in practice – both from the ones, which validate the environment straightaway after installation or updating, and from the others, which try to "hide" the checks in source code.

To be able to describe requirements regarding execution environment, a formal language is required to encode the requirements, moreover, the language must be extendable, when new kinds of requirements are defined. Such aspect complicates the construction of test coordinator, since it has to be compatible with a language, which is not fully defined during development of coordinator. The problem is solved by assigning the coordinator only the role of language syntax analysis, but the semantic analysis of requirements is performed in environment validation modules.

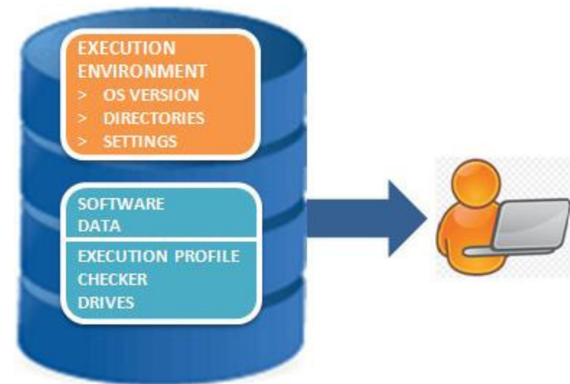

Fig. 2 Execution environment testing

The practical implementation showed that development of the proposed approach requires relatively little programming resources.

C. Self-testing

The research of the authors [13] offers an original approach to software testing, named as self-testing. Self-testing is a software's ability to test itself automatically prior to operation, and it can be performed even in a productive environment. The self-testing feature in software is similar to hardware self-tests that are executed every time after the device is turned on. Instead of traditional testing that verifies correctness of software in testing environments using testing tools, the self-testing property is built-in software component that executes accumulated test cases using means of the information system. It helps to perform tests not only in testing environment, but also to verify software correctness in action with real data in production environment.

Self-testing contains two main components:

- test cases that are designed for checking of critical functions of the software
- built-in automated testing mechanism providing automatic execution of tests and result comparison with benchmark values.

Designing of test cases covering the critical functionality (lack of these essential functions causes inoperability of the whole system) is a part of requirement analysis.

Implementation mechanism of self-testing approach uses software instrumentation, and it has been offered quite a while ago [14], [15]. The idea is to supplement the source code with extra routines for self-testing purposes that are executed if the software is run in the testing mode. The points in source code where the routines are included are named as test points. Testing routines allow to monitor values of variables and to compare them with benchmark values therefore checking the correctness of the information

system. Unfortunately, this solution is usable only for those information systems whose development is in the testers' influence sphere.

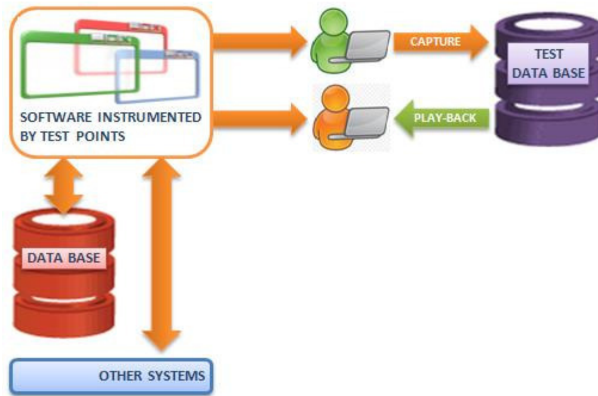

Fig. 3 Self-testing

Self-testing can be used in four modes (see Fig. 3):

1. test-capturing – running of software instrumented by test points and capturing of new test cases into test data base or editing of the existing ones;
2. self-testing – automated self-testing of software by automated execution of the captured test cases;
3. normal usage – running of information system without any testing activities;
4. demo mode – running of information system using pre-captured demo scenarios.

Comparison of self-testing implementations with automated testing tools leads to the following conclusions:

- Unlike the majority of globally recognized testing support tools, the self-testing approach offers some additional options: testing of external interfaces to other information systems and database management systems, testing in production environment, testing with the white-box method, possibility for users without IT knowledge to capture tests.
- The self-testing technology makes possible to test software throughout the whole life cycle of an information system – from early stages of software development till maintenance activities, because it is suitable for testing in all development, testing and production environments.
- The self-testing functionality should be integrated into software already during development of the software.
- The self-testing requires additional work to include the self-testing functionality in the software and to design test cases; on the other hand, self-testing saves time as repeated (regression) testing of the existing functionality is available
- Implementation of the self-testing functionality is useful in incremental life cycle models, in particular if information systems are improved gradually and maintained for many years; it is less useful in linear (waterfall) life cycle models.

Empirical studies show that 60% of information systems' problems would be possible to identify and rectify by self-testing approach [13].

D. Embedded business processes

Development of software engineering tends to devote more attention to precise modelling and designing of information systems instead of extensive programming. Some researchers are even predicting development of information systems without programming at all in very near future. Business process modelling is a compulsory initial phase of every information system development project according to this concept [16].

Workflow based information systems is the area where business process modelling is an essential component for functioning of information systems. Business process of organization is described by a workflow model containing sequential business process steps – activities - together with performers of the activity, deadlines, the actual state of the object in the workflow etc. Documents and reports can also be created during the workflow execution, and this should be included in business process descriptions.

It is common to describe business processes using modelling languages. There can be used universal modelling languages or domain specific languages (DSL). When DSL is chosen, it must ensure two important features: a) the language should be easy understandable for the majority of users, b) it should include all necessary information for automated execution of workflow steps.

The first step in development of information systems is to describe business processes to be supported (see Fig. 4). A set of graphical diagrams are created using DSL, and it serves as business process model. After the model is created the information from the diagrams can be transferred to the database of an information system. The business process descriptions are embedded into the information system, and the engine of the information system can interpret information from the diagrams. Embedded business processes ensure that the information system behaves according to the business process model.

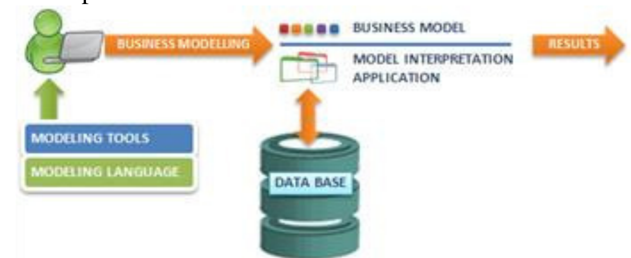

Fig. 4 Embedded business process

As practice shows [17], it is possible to create a special tool for transfer of model's data to executable application relatively quickly. The API of the graphical editor can be used to access the model's repository, to gather the information and to transfer it to applications database. This guaranties that the application operates according to the model developed in a graphical DSL. And the overall quality of the application – usability, reliability, performance etc. – is dependent on the application itself, not on the hypothetical

ability of a code generator to create an application in the desired quality.

The authors have created the domain specific language BILINGVA [18] that is convenient for description of workflows. The approach was tested in practice, and particularly surprising was the positive feedback from users about the graphical representation and implementation of business processes. The diagrams served as some kind of information system's user manual that explained functioning of the information system in a more precise and understandable way than the conventional (written) user manuals.

E. Business process runtime verification

From the beginnings of information systems the topical issues were: does the information system operate correctly?, are the system's results adequate?, and is the information system in the correct state in terms of the relevant business? Sometimes processes must be stopped as soon as possible after inadequate situation has occurred, otherwise more serious problems could rise [19].

Inadequate situations can be caused by many conditions. They can be caused by heterogeneous systems, which are developed at different times and used in a variety of companies. Problems can arise due to poor software quality and lack of testing. Problems may also occur due to incorrect user actions: incorrect execution of business functions, a breach of the input restrictions, or the timing and sequence of process steps.

For example, if warehouse system is not updated with payment information from accounting system timely, goods cannot be issued to customer. On the other hand, this situation is unacceptable for the customer, who has done payment according business process. Obviously, in this case there is no reason to look for errors in information system, but a person should monitor that payment data are imported timely. This is basic task of runtime verification – to verify systems execution in their runtime.

The authors [19] propose a solution for business process runtime verification (see Fig. 5). The basic idea of the solution is to run a separate verification process for each controllable business process (further – base process). Verification processes are described in DSL that has been developed in conjunction with the solution. A base process typically is executed by information systems, while verification processes should be executed on the basis of independent and external controlling software (further – a controller). The steps of the verification process are linked to base process steps and are described by events that acknowledge the execution of the each base process step.

Base processes can be executed manually or automatically by computer, and verification processes are executed independently. Each of base process steps makes some changes in the process "memory" (usually stored in database or file system). The verification controller receives acknowledgement from event agents about base process memory modifications, therefore identifying inconsistencies between the received information and the description of the verification process. If inconsistencies are detected, then they are reported to the support staff.

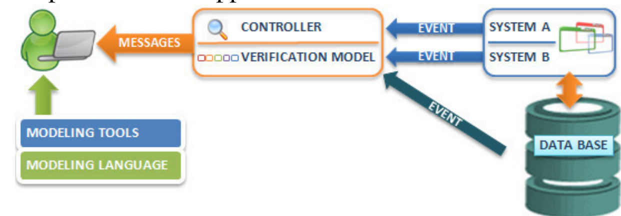

Fig. 5 Runtime verification

The solution provides a number of interesting possibilities, which bring us closer to the goal defined by ideas of autonomous computing:

- runtime verification can be done without modifications of the base process
- process verification can be added dynamically to legacy systems
- verification does not depend on modelling language used for process description, it depends only on possibility of verification agents to identity events of the base process.

Likewise, some solution limitations must be taken into account: verification mechanism can detect only those base process steps which leave some modifications in the systems „memory". Otherwise verification agents cannot work as external process, but must be incorporated into the base process.

However, it must be stressed that the proposed solution can significantly reduce monitoring load of information systems' operational staff. It automates business process runtime verification that typically is done manually and not continuously.

IV. Conclusion

There were spent several years on research to achieve goals similar to autonomic computing – facilitating the use, maintenance and development of systems by including support components in them. The conclusions are as follows:

- several components, created using smart technologies, can provide good support in use, maintenance and development of information systems which are easy enough to implement for a small/medium size organization;
- there are quite many functions, which could be supported by respective smart technologies, for instance, data quality control, confidentiality control, built-in privacy protection [20], performance monitoring, availability monitoring, selecting environments for software compatibility testing [21], automatic testing of WEB services [22] and others;
- smart technology enabled systems are currently not very common due to the fact that these ideas are not popular enough yet; with increasing complexity of information systems, smart technologies will surely

grow in importance and will help to deal with complex system development and maintenance issues.

References

[1] Kephart J., Chess D. The Vision of Autonomic Computing. Computer Magazine, IEEE, 2003, DOI=10.1109/mc.2003.1160055

[2] KEPHART, Jeffrey O. Autonomic computing: the first decade. In: ICAC. 2011. p. 1-2., DOI=10.1145/1998582.1998584

[3] Lalanda P., McCann J. A., Diaconescu A. Autonomic Computing: Principles, Design and Implementation. Springer, 2013, 288 p., DOI= 10.1007/978-1-4471-5007-7

[4] Bičevska Z., Bičevskis J. Smart Technologies in Software Life Cycle. In: Proceedings of Product-Focused Software Process Improvement. 8th International Conference, PROFES 2007, July 2-4, 2007 (Münch, J., Abrahamsson, P., eds.), Riga, Latvia, vol. 4589/2007, 2007. pp.262-272, DOI= 10.1007/978-3-540-73460-4_24

[5] Nami M. K., Bertels. A. Survey of Autonomic Computing Systems. In: ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems, 2007. p.26, DOI= 10.1109/conielecomp.2007.48

[6] Sterritt R., Bustard D. Towards an autonomic computing environment. In: Proceedings of 14th International Workshop on Database and Expert Systems Applications (Marík, V., Retschitzegger, W., Stepánková, O., eds.), Prague, Czech Republic, 2003. pp.694 – 698, DOI= 10.1109/dexa.2003.1232103

[7] Herrmann K., Muhl G., Geihs K. Self management: the solution to complexity or just another problem? Distributed Systems Online, 2005, 1, vol. 6, DOI= 10.1109/mdso.2005.3

[8] Bičevska Z., Bičevskis J. Application of Smart Technologies in Software Development: Automated Version Updating. In: Scientific papers, vol. 733 (Bārzdiņš, J., Freivalds, R.-M., Bičevskis, J., eds.), University of Latvia, 2008, pp.24 -37.

[9] Rauhvargers, K. On the Implementation of a Meta-data Driven Self Testing Model. In: Software Engineering Techniques in Progress (Hruška, T., Madeyski, L., Ochodek, M., eds.), Brno, Czech Republic, 2008, pp.153-166.

[10] Arnautovic E., Kaindl H., Falb J., Popp R., Szep A. Gradual transition towards autonomic software systems based on high-level communication specification. In: Proceedings of the 2007 ACM symposium on Applied computing, 2007, pp.84-89., DOI= 10.1145/1244002.1244024

[11] Orso A., Jean M., Rosenblum D. Component Metadata for Software Engineering Tasks. In: EDO '00: Revised Papers from the Second International Workshop on Engineering Distributed Objects, London, vol. 1999, 2001, pp.129-144, DOI= 10.1007/3-540-45254-0_12

[12] Rauhvargers K., Bicevskis J. Environment Testing Enabled Software – a Step Towards Execution Context Awareness. In: H.-M. Haav, A. Kalja (eds.), Databases and Information Systems, Selected Papers from the 8th International Baltic Conference, vol. 187, IOS Press, (2009), pp. 169–179.

[13] Diebelis E., Bičevskis J. Software Self-Testing. In: Proceedings of the 10th International Baltic Conference on Databases and Information Systems, Baltic DB&IS 2012, July 8-11, 2012, Vilnius, Lithuania. IOS Press, vol. 249, 2013, pp. 249 – 262

[14] Bichevskii YY, Borzov YV. Prioriteti v otladke bolsih programmnih sistem Programmirovanie, 1982, vol. 3, pp. 31-34 (in Russian).

[15] Chengying M., Yansheng L., Jinlong Z. Regression testing for component-based software via built-in test design. In: Proceedings of the ACM symposium on Applied computing, March 11 - 15, 2007, Seoul, Korea, 2007. pp.1416-1421, DOI= 10.1145/1244002.1244307

[16] Draheim D. Business Process Technology: A Unified View on Business Processes, Workflows and Enterprise Applications. Springer Berlin Heidelberg ISBN: 978-3-642-01587-8 (Print) 978-3-642-01588-5 (Online), www.springer.com (2010), DOI= 10.1007/978-3-642-01588-5

[17] Bicevskis J., Cerina-Berzina J., Karnitis G., Lace L., Medvedis I., Nesterovs S. Practitioners View on Domain Specific Business Process Modeling. In: Databases and Information Systems VI. Selected papers from the Ninth International Baltic Conference DB&IS 2010, IOS Press, 2011, pp. 169-182.

[18] Cerina-Berzina J., Bicevskis J., Karnitis G. Information systems development based on visual Domain Specific Language BiLingva. Selected Papers from the 4th IFIP TC 2 Central and East Europe Conference on Software Engineering Techniques, CEE-SET 2009, Krakow, Poland, LNCS 7054 Springer, 2011, pp. 124-135., DOI= 10.1007/978-3-642-28038-2_10

[19] Oditis I., Bicevskis J. Asynchronous Runtime Verification of Business Processes. In Proceedings of the 7th International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN), Riga, 2015, pp. 103-108.

[20] Nai-Wei L, Alexander Y. Danger Theory-based Privacy Protection Model for Social Networks In Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, 2014, pp. 1397–1406., DOI= 10.15439/2014f129

[21] Pobereznik L. A method for selecting environments for software compatibility testing In Proceedings of the 2013 Federated Conference on Computer Science and Information Systems pp. 1343–1348

[22] Bluemke I., Kurek M., Małgorzata Purwin M. Tool for Automatic Testing of Web Services In Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 1553–1558., DOI=10.15439/2014f93