Baltic J. Modern Computing, Vol. 3 (2015), No. 4, 273-293

A Practitioner's Approach to Achieve Autonomic Computing Goals

Janis BICEVSKIS[2], Zane BICEVSKA[1], Kriss RAUHVARGERS[2], Edgars DIEBELIS[1], Ivo ODITIS[1], Juris BORZOVS[2]

[1] DIVI Grupa Ltd, 40-33 Avotu Street, Riga, LV-1009, Latvia
[2] University of Latvia, 19 Raina Blvd., Riga, LV-1586, Latvia

Janis.Bicevskis@lu.lv, Zane.Bicevska@di.lv, Kriss.Rauhvargers@lu.lv, Edgars.Diebelis@di.lv, Ivo.Oditis@di.lv, Juris.Borzovs@lu.lv

Abstract. The paper proposes a new approach to the development of software resulting in a greater facility in using and maintaining an IT system; thus moving closer to the main goal of all autonomic systems – self management. The authors propose including five practice driven components into the software. These five components are smart technologies: version updating, testing of execution environment, self-testing, runtime verification and business process execution. The smart technologies have been successfully implemented in several IT systems.

Keywords: Autonomic computing; smart technologies; business process modelling; smart technology framework.

Introduction

Information technologies provide unprecedented opportunities to automate many processes of human life. Actions which have only a few decades ago been the preserve of human beings can be executed by programmable equipment now. But the mankind's progress has also brought up new challenges. One of them is complexity of computing systems. The authors (Kephart and Chess, 2003) refer to as „computing systems with complexity approaching boundaries of human ability". The IBM autonomic computing manifesto (Horn, 2001) claims: "It's time to design and build computing systems capable to manage themselves, adjusting to varying circumstances, and preparing their resources to handle most efficiently the workloads we put upon them."

Information system developers are continuously faced with information systems complexity issues. A frequently used approach how to deal with the problem is including specific components into information systems that help to deploy, use and maintain them. In many cases, support functions are created for particular information systems without any generalization. For instance many information systems have built-in error handling components. In emergency cases the component takes over, informs developers about an accident and delivers additional information needed to handle the problem. This component is undoubtedly helps to maintain the system, allowing to identify the causes

of accidents and to prevent them. On the other hand components like that described above usually are not a part of information system's basic functionality. These are so-called non-functional features of information systems, as they do not provide any functional gains for the information systems and "just" sustain developers during the information system maintenance.

The concept of smart technologies (Bičevska and Bičevskis, 2007) has similar objectives as the concept of autonomic computing (Horn, 2001). The main goal of both is to create self-managing information systems, but the desired levels of generalization are different. The autonomic computing pursues an ambitious goal to create sophisticated information technologies in general whilst the approach of smart technologies provides a set of practically applicable non-functional features to simplify the maintenance and daily use of information systems (see Fig.1).

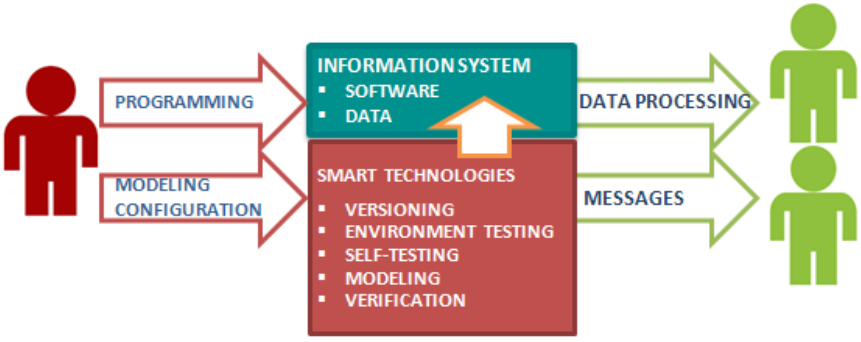

Fig. 1. Smart Technologies (overview)

The idea behind the smart technologies is to supplement an information system with specific built-in non-functional modules providing self-management features. The concept of "smart technologies" in the sense of this research first appears in (Bičevska and Bičevskis, 2007) in relation to software system solutions that at least partly contain system self-management features. Unfortunately, similarly to the concept of autonomic computing also the concept of "smart technologies" is missing an unambiguous definition that would provide evaluation criteria whether a particular solution falls into the category of "smart technologies" or not. We also admit that the objectives of both autonomic computing and smart technologies concepts are very similar; however ways to reach them differ greatly. This paper describes five types of smart technologies identified during practical software development and various experiments:

- Built-in software versioning and data syncing – automated deployment of information system components (software, data structure descriptions, screen and report forms, etc.) from a central repository to local workstations and servers including conversion of data structures and migration of historical data into the new structures. Results of the research (Bičevska and Bičevskis, 2008) have been successfully implemented in financing and budget controlling application widely used by public authorities in Latvia.
- Testing of external environment – automated checking of external environment to be able to discover discrepancies and to inform users about potentially necessary actions. The offered approach foresees a special "passport" to be created for a

particular information system that contains the requirements for the external environment of the information system (operating system, library versions, configuration values for the directory physical location, computer technical parameters, etc.) in order to ensure system's correct functioning. Results of the research (Rauhvargers and Bicevskis, 2009), (Rauhvargers, 2008) have been anticipated in several information systems in Latvia.

- Self-testing – ability of information system to check its own integrity and uptime. Technically it is implemented as built-in support components for testing of information system on stored test cases in operating environment (Bicevska and Bicevskis, 2008), (Diebelis et al., 2009)], (Diebelis and Bičevskis, 2011), (Diebelis and Bičevskis, 2013).
- Embedded dynamic business model – ability of information system to run according to an external business process model described in domain specific language (Bicevskis et al., 2010). The business model should be specified at a level of detail so it can be run (interpreted) directly. This approach is successfully implemented in event-driven information systems (Cerina-Berzina et al., 2011).
- Runtime verification – business process execution control in production environment. The component is implemented as an independent control process at predefined business process points to ensure that the whole business process works according to business needs (Oditis and Bicevskis, 2015).

Undoubtedly it is possible to find many other types of smart technologies, which would be worth to explore and use.

Although the smart technologies approach and the autonomic computing manifesto (Horn, 2001) approach seemingly share some similarities, it should be emphasized that the smart technologies approach was developed independently. Unlike autonomic computing ideas smart technologies are not looking for a universal solution, but they are limited to easily implantable and practically attractive features.

The practical results gained in IT projects since 2007 provide evidence of the usefulness of the approach. Analyzing more than 70 SCOPUS indexed articles devoted to the autonomic computing, authors could not find researches which would be substantially similar with smart technologies approach.

This paper is a continuation of the research described in (Bičevska et al., 2015). It contains more detailed characteristics of smart technologies, including description of practical context, technical solutions and the results achieved.

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

1. Related work

Smart technologies and autonomic computing have a similar goal - reduce the complexity of system use by delegating some part of user support functions to the information system itself. 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 (Nami and Bertels, 2007) - from basic level (manually maintainable information systems) to completely autonomic systems that are able to function operate accordingly to guidelines

set by humans. The main statement implies targeted development of information systems that are able to self-management thus overcoming gap between users and increasingly complex world of information technologies.

The manifesto lists 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. It uses early warning to anticipate and prevent system wide failures.

The manifesto does not include any instructions about implementation issues, but some sources discuss ideas about essential components of autonomic systems. For instance R. Sterritt (Sterritt and Bustard, 2003) describes an autonomic environment consisting of autonomic elements, which are mutually connected via 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 it's 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.

Other authors (Arnautovic et al., 2007) propose to decentralize the communication among components grouping them into functionally related groups wider exchange of data is allowed, perhaps even in their own communication dialect.

Some researchers indicate that, in order to create truly autonomous systems, their components should be clearly documented. The documentation should be provided in a computer-readable format, so, it is necessary to complement the components with descriptive metadata to facilitate their operation and maintenance further (Tosi. 2004), (Orso et al., 2001).

In 2003 IBM extended the list to eight characteristic aspects (Kephart and Chess, 2003), adding system's ability to "know itself" and manage its resources, system's ability to know its environment and the context surrounding its activity, and act accordingly – to adjust, operate in heterogeneous environment accordingly its open standards, as well as anticipate the optimized resources needed while keeping its complexity hidden. The fundamental concept in autonomic computing is the idea of self-regulation and the self-governing operation of the entire information system, thus disburdening users and administrators from complexity of system's use and maintenance.

Achievements of autonomic computing movement during its first decade after publication of the manifesto have been explicitly demonstrated in (Kephart, 2011). Particularly we emphasize (Lalanda et al., 2013), where the list of self-management features was extended with new ones, reaching a total number of 24, as well as it contains analysis of most important achievements in implementation of autonomic

computing. As of now, manifesto's targets have been met only to some extent. Paradoxically, to solve the problem—make things simpler for administrators and users of IT—we need to create more complex systems. Continuing efforts on autonomic systems include both, theoretical research and practical implementation (Kephart, 2011).

The autonomic computing approach has also been criticized (Herrmann et al., 2005), and the main reasons are:

- 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.

The concept of smart technologies 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.

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.

2. Components of smart technologies

There are five fields of smart technologies where practical results were gained since 2007: embedded software versioning and data syncing, execution environment testing, self-testing, embedded dynamic business model and runtime verification.

2.1. Software Versioning and Data Syncing

Context

Development of successful information system takes many years, during which the system is being improved, supplemented with new possibilities and features. And every time you made changes to the information system, a new version of the system of it should be deployed. During previous decades it was carried out by delivering new installation packages. The users installed new version of the information system and hat to rely on the installation process. This approach is suitable for rather simple systems where transition to new versions does not include complex data structure changes and data migrations to the new version, or includes to a limited extend only. Commercial license-based software products are sometimes complemented with the old version support. For instance Microsoft Office versions have built-in support for objects created in the previous versions.

The custom-made information systems are in worse situation. Often, the new versions of programs are unable to work with the old data, and vice versa – the old programs are unable to work with the new data structures. For example, planning and budgeting

depends on the budget structure and it is variable in the planning institutions and the state as well. It means the database structure and reporting forms are periodically changing. Hence the transition to a new version of information system can't be achieved with a simple software re-installation, it also requires data migration to new structures and transition of configuration information to a new version of the information system. Common users often fail to perform it correctly, and it can result in a distorted institutional activity.

The first thought is that this problem could be easily solved by creating a centralized web-based solution. But budget planning and execution is closely related to locally-kept accounting records of each institution. Locally stored data can be difficult to centralize due to security or organizational reasons.

The described situation is repeated in many systems, and the challenge was to create an automated system version delivery, ensuring the integrity of the entire information system, not only for software components.

Solution

Solution to the problem is offered by creating a component of smart technologies - Software Versioning and Data Syncing. 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 – an "updater" (see Fig. 2) for software and data (data structures, templates, reports, configuration etc.) upgrading. The supporting mechanism (so called "updater") should be built into systems

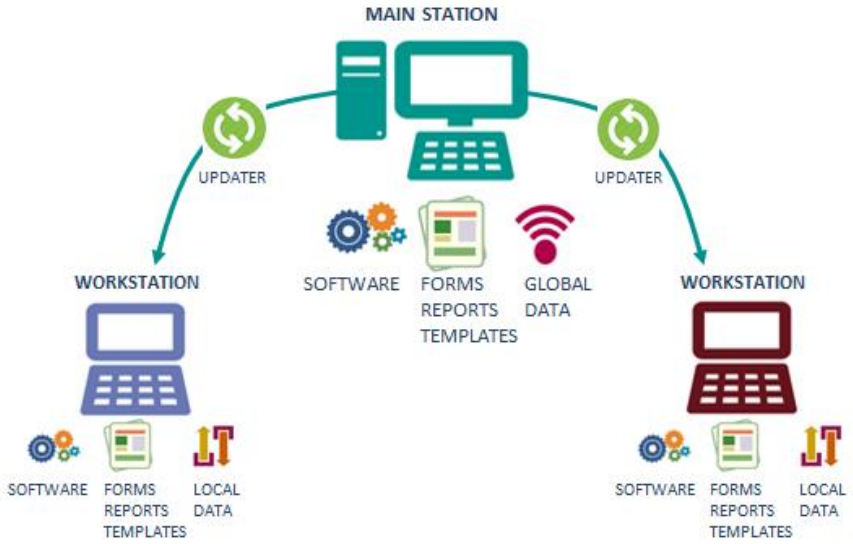

Fig. 2. Smart technology component: Updater

and it should include features for deploying of new versions without any user intervention. The following should be ensured automatically during deployment process:

- Create backups to be able to recover the system in case of incidents;
- Check whether the software to be delivered is in compliance with the external environment;
- Download and install a new version of the software;
- 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.

The majority of information systems today support some of the characteristics listed above, but in most cases - to a limited extent only.

Results

Authors of this paper have implemented the characteristics in several software development projects, the obtained results are described in (Bičevska and Bičevskis, 2008). The solution containing the built-in smart technology component "updater" is successfully being used in more than 100 institutions.

The proposed solution was very topical in distributed information systems with desktop applications, as the new versions of it required reinstallation of the entire information system or its crucial components in many workstations. The situation is changing as web applications are becoming more and more popular. It is possible to store programs and data centrally, so the deployment of new versions can be made of highly qualified personnel and not have to rely on a non-IT staff available in small organizations. For the cases when data is stored locally (for instance due to safety requirements) the problem can be addressed in the above described manner. The rapid expansion of mobile apps sets new challenges as the deployment of new versions in this area is not completely solved yet.

2.2. Execution environment testing

Context

It is quite common that programs have specific requirements for their successful operation in a given environment. The requirements can relate to operating system, network characteristics, workstation parameters, etc. Discrepancy between the information systems requirements to external environment and the concrete execution environment may occur in several situations:

- 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.

- Several versions of the same software are installed on the workstation. The environment that is consistent with the "first" systems version can be incompatible with other systems versions. Or even worse, environment arranging in order to ensure all for the "first" system necessary components can "demolish" the environment needed for other systems versions. The validation should be performed on demand, for instance, before each session; however, some authors propose validation only during installation.

Practical use of information systems shows that many incidents and failures are not related to the functionality of the information system itself, but rather are caused by inadequate infrastructure and the execution environment. It means information systems must be accompanied by automatic means for external environmental testing.

Solution

The authors (Rauhvargers and Bicevskis, 2009) propose a technology, which allows independent environment checks, performed by the software, named – "checker", in order to validate if the execution environment is suitable for normal execution (see Fig. 3). The proposed solution implies gathering these requirements in a "software profile" to be able to validate the execution environment before program's starting. 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.

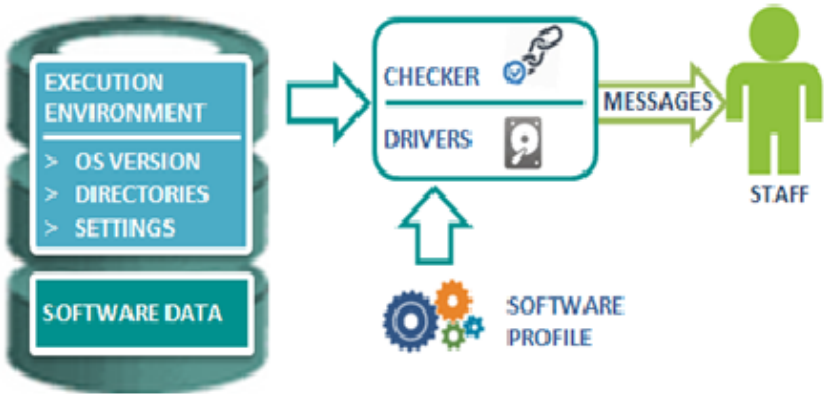

Fig. 3. Smart technology component: Execution environment testing

The practical environment testing task is carried out by "checker", which manages environment validation modules- drivers. Each driver 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, for instance, decimal separator must be the symbol ",", data base server must be reachable, etc.). 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 – in the software profile. This concept differs 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 an aspect complicates the construction of checker, since it has to be compatible with the language, which is not fully defined during development of checker. The problem is solved by assigning the checker only the role of language syntax analysis, but the semantic analysis of requirements is performed in environment validation drivers.

Results

The proposed solution is used in a number of local information systems in Latvia. As described in the precious chapter, an execution environment testing was usually performed when supplying a new version of the information system. The new version was installed only after the current execution environment was checked for its ability to run the new version. Also, receiving alarms from users about the systems malfunctions there was first tested if the execution environment of the concrete workstation meets the environment requirements. In many cases, missing or wrong components of the execution environment were the reason for malfunctions.

The described approach can also be used for other purposes, for instance to monitor the computer systems that are in use in company's internal network and to check the compliance of configurations with standards set by the company.

The practical implementation showed that development of the proposed approach requires relatively little programming resources. The proposed smart technology solution and obtained results are presented by the authors in (Rauhvargers and Bicevskis, 2009).

2.3. Self-testing

Context

One of the smart technologies offers an original approach to software testing, named as self-testing. 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 all user needs. And every time, when the software or the operating environment has been changed, also the correctness of the entire running system has to be checked.

Traditionally, the testing is carried out in a separate environment, so-called test environment. The testing process is supported by testing tools which allow gathering of test cases, executing of test cases and composing of test reports. By default it is assumed that test environment corresponds to the execution environment, respectively the assumption is that well tested information system will work in the execution environment just as well as in a test environment. Unfortunately this assumption is not always fulfilled. There can be many causes for that, some of them are as follows:

- Productive system databases are filled with real data, which content and amount may differ significantly from those in the test environment.
- Interfaces to productive external systems can be unavailable in the test environment as the availability can be determined by the safety requirements, licensing conditions for use, testing support tool restrictions etc.

So even very carefully tested information systems can work incorrectly in the execution environment. To restrict such cases, developers use to deliver developed information systems together with automatically executable test cases for validation of system's critical functionality.

For example, a complicated process graph could be delivered as a test case covering all admissible constructions for testing of a calculation program. The calculations can be performed without any influence of them to the stored data in the production database hence the testing in execution environment could be performed if the calculation software would have mechanisms for this kind of operations. Unfortunately, only in rare cases, the program provides an opportunity to perform testing in the execution environment.

Solution

Self-testing is a smart technology providing the software with a feature to test itself automatically prior to operation. There is similarity between self-testing and hardware self-checking where computer tests its own readiness for operation when it is just turned on. The purpose of self-testing is to use a built-in support component for automated execution of previously accumulated tests cases not only in test, but also in execution environment.

Self-testing contains two components:

- Test cases of system critical functionality that check the set of functions without which the software could not be used. Identification of critical functionality and designing of tests for it, as a rule, is a part of the requirement analysis and testing process.
- A built-in automated testing mechanism (regression testing) providing automatic execution of tests and comparison of results with benchmark values. These features are typically a part of traditional testing tools, and the self-testing approach offers to build them into the system during the development.

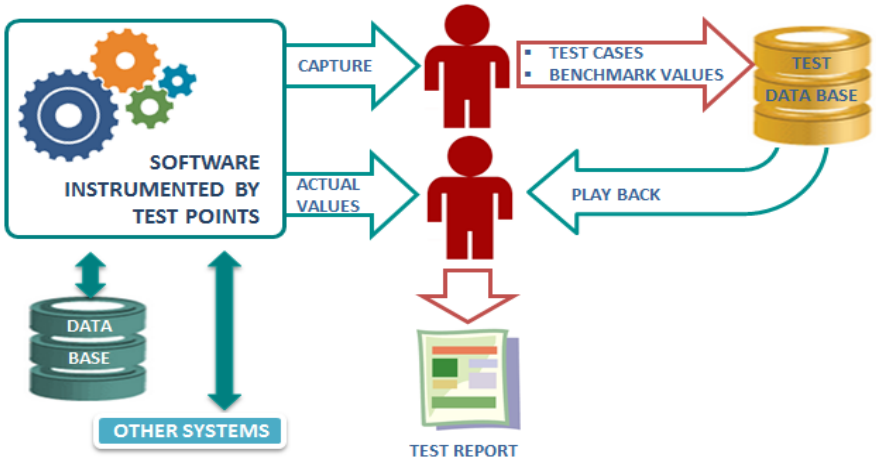

Fig. 4. Smart technology component: Self-testing

Implementation mechanism of self-testing approach uses an idea and means of the software instrumentation, which is already known from the 70-ies. Testing operations are put by programmers into certain places of the source code; these points are named as test points. Testing operations allow to track the changing values and to compare them with a benchmark. Thus it is possible to check the correctness of the information system. Unfortunately, this solution is usable only for that software whose development is in the user's influence sphere.

It should be noted that the idea of built-in support for program testing has been offered quite a while ago (Bichevskii and Borzov, 1982), (Chengying et al., 2007) and it has been implemented in some projects. Regardless the system environment (Development, Test and Production) self-testing functionality can be used in the following system modes (see Fig. 4):

1. Test capture mode - new test cases are captured or existing tests are edited/deleted.
2. Self-testing mode - automated self-testing of software is done by automated execution of stored test cases.
3. Use mode - there are no testing activities – a user simply uses the system. The built-in self-testing mechanism can be used in emergency situations to find out the internal state of the system, which may facilitate the analysis of the causes and consequences of the emergency situation.
4. Demonstration mode. The demonstration mode can be used to demonstrate system's functionality. User can perform system demonstrations using use cases stored in storage files.

The implementation of self-testing feature can be done in ca. 3000 LOC in C++. Additionally, the source code of the particular system should be instrumented with testing activities like accumulating of test cases and executing of them. These investments are justified when the system is designed and developed for long-term use.

Results

Instead of traditional testing that verifies correctness of software using testing tools in test environment, the proposed approach provides to build the testing support during a software development. It helps to perform tests not only in testing phase, but also offers opportunity to verify software correctness in action with real data in production environment.

Nowadays a wide range of testing tools is available, and they are intended to support various testing methodologies. In evaluating the usefulness of the self-testing approach, the opinion of the Automated Testing Institute (ATI), which is a leading authority in the field of testing, was used. The self-testing tool was compared with globally popular testing tools that have received ATI Automation honours. Comparison and evaluation of testing efficiency led to the following key conclusions:

- The self-testing approach and the inclusion of testing support functions in the system offer not only options equal to those offered by other globally recognized testing support tools. Moreover, self-testing additionally offers options that other testing tools do not possess or do it poorly: testing external interfaces with other systems and database management systems, testing in production environment, testing with the white-box method, possibility for users without IT knowledge to capture tests.
- Since testing support is a part of systems developed and it is available throughout the entire life cycle of software, the offered self-testing technology makes possible to test software during development, regression testing as well as maintenance in the development, testing and production environments.
- Self-testing changes the testing process by considerably broaden the role of the developer in software testing. The self-testing functionality should be integrated into software already during development. As shown by the real experience, self-testing would let to identify and rectify 60% of all later discovered bugs (Diebelis and Bičevskis, 2013).
- Implementation of the self-testing functionality is useful in incremental development models, in particular in systems that are gradually improved and maintained for many years, and less useful in linear (waterfall) development models.

The self-testing support tool requires further development with regard to technical implementation aspects: to widen the range of platforms available and to provide support for performance, load, stress and other new testing options. The results are described in (Diebelis and Bičevskis, 2013).

2.4. Embedded business processes

Context

Nowadays the software development cycle is becoming shorter, and changes in information systems should be implemented fast and effectively. So the smart technology research has been also devoted the question how to change information systems functionality fast and without much effort.

The main idea behind embedded business processes is to create an accurate information system model in a high abstraction level and to develop a software

component, so-called interpreter, that works accordingly to the system model. If changes in the systems model are made the functionality of the information system can be automatically changed to a certain extent.

The approach as an idea is known since the 60-ies. The main burden for its wider usage are implementation issues of non-functional features - usability, security, performance etc. The non-functional features are hard to describe using means of functional modelling, and so they can't be obtained from the system model automatically in a good quality.

Smart technologies, unlike other related solutions, propose to create an interpretable system model only for the functional part relating to workflows, not for all systems components and functions. The rest of functionality is created in a traditional way, and a built-in business model interpreter is included into information system allowing derive some part of functionality from business process descriptions.

The approach is useful if several enterprises run similar but not identical business processes. Using the same interpreter but defining specific business process models for every enterprise, it is possible to obtain the desired functionality and to use the same universal software in all enterprises.

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.

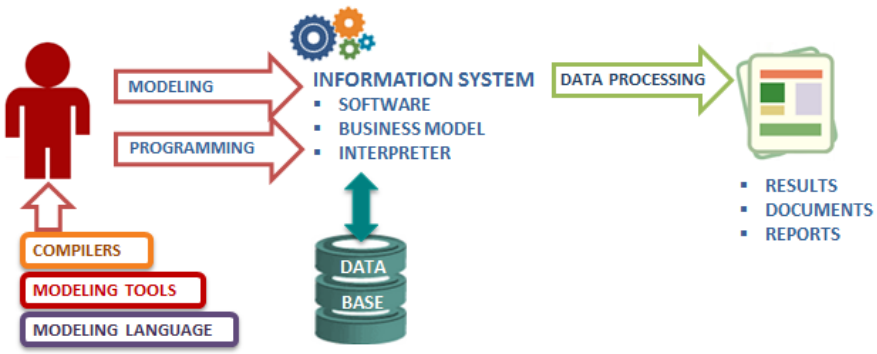

Fig. 5. Smart technology component: Embedded business process

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: (1) the language should be easy understandable for the majority of systems stakeholders, (2) it should include all necessary information for automated execution (for interpreter) of workflow steps. Usage of DSL lets business process experts to build the domain specific business process models, and, on the other hand, it allows IT professionals to provide the system with integrated mechanisms for interpreting of the models.

Solution

At the beginning of information system development the business processes (see Fig. 5) should be described as the information system will be designed to support them. A set of business process descriptions are created using DSL, and it serves as business process model. Graphical representations like diagrams can easily be understood and used by domain experts (as a rule, non-IT specialists) for the business process description. After the business process model is created, the information from the diagrams can be transferred to the database of an information system, and it is a task for IT professionals. The business process descriptions are embedded into the information system, and the engine of the information system can interpret information born from the diagrams. Embedded business processes ensure that the information system behaves according to the business process model.

As practice shows (Cerina-Berzina et al., 2011), 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. Thus guaranties that the application operates according to the model developed in a graphical DSL. And the overall quality of the application – usability, reliability, security, 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.

Contrary to the model driven architecture (MDA) that aims completely automated generation of information systems from business model descriptions, the offered approach provides the development of information systems in the traditional way. The required functionality and non-functional features like usability, security, performance, etc. are enhanced with an extra component for workflow execution that is able to run the information system according to business process descriptions imported from the business process model.

Results

The authors have created the domain specific graphical language BILINGVA (Cerina-Berzina et al., 2011) that is convenient for description of workflows. Graphical editor for diagrams was created using a specific tool building platform (Barzdins et al., 2007), (Barzdins et al., 2009).

The developed solution was successfully applied to the Latvian state institutions, which supervise and monitor various project tenders. Every institution had its own branch-specific business processes, respectively every institution supervised their projects in a different way. An average business process model consisted of ca. 20 graphical diagrams with in average 20 activities (steps) per diagram. The relevant regulatory documents that describe these processes in natural language, took about 200 DIN A4 pages.

Instead of each institution to develop its own information system, a universal application was created being able to interpret different business models. Each institution received a high quality information system with desired functionality that can be easily changed making changes to the business model. The non-functional features of the applications as well as some very specific screen forms were programmed in the traditional way.

The approach was applied in practice, and particularly surprising was the positive feedback from users about the graphical representation 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.

2.5. Business process runtime verification

Context

Business process runtime verification is targeted to enterprises using many different types of software for support of different business processes (Draheim, 2010). The establishment of a heterogeneous environment in large and long–lasting companies is inevitable, because the organizational size and functions are subject to changes over the course of time and develop gradually.

Traditional testing methods are not able to prevent conflicts of different processes and systems in collaboration, where part of the process is done by people, and the other part is supported by software. The software can be designed to support particular processes in different environments at different time frames. Self-management features if they are a part of information system approximate the main objectives of autonomic systems.

This is also evidenced by several authors arguing that static verification and testing of software are insufficient aids for modern business process verification that relates to a service-based architecture in a heterogeneous environment. Processes are implemented by many components changing independently over the course of time. This means the process runtime verification must be conducted through the entire lifetime of the process (Wu, 2013), (Ghezzi and Guinea, 2007).

Runtime verification has been well known for years in the area of embedded systems. It is an approach to computing system analysis and execution based on extracting information from a running system and using this information to detect and possibly to react to observed behaviors satisfying or violating certain properties. Such defense mechanisms may be included in the system during its development or they may be included as independent controls from the base process. The independent character of such mechanism allows making later adjustments by adding or disabling the controlling component when a system is developed, and changes are made. These ideas can be applied in business process runtime verification, too.

An example of a bank's electronic clearing system (ECS) and its file processing module illustrates the functioning of the business process runtime verification. ECS collects files with payment messages from the banks, settles payments and delivers payments to the banks-senders. There are few tasks for runtime verification:

- Identify payment files processed too slowly;
- Report to the staff about ECS process failures.

The state chart depicted in Fig. 6 illustrates simplified payment file process in order to explain the essence of the idea. From the system's point of view, the file processing starts with the "File received" state, i.e., the file is copied onto the system operator's file server

(via FTP or other file transfer services). Then the file is registered in the system's database, and the actual processing is launched:

- The file is decrypted;
- The file is parsed and all messages are extracted;
- All messages are checked in accordance with the particular business rules.

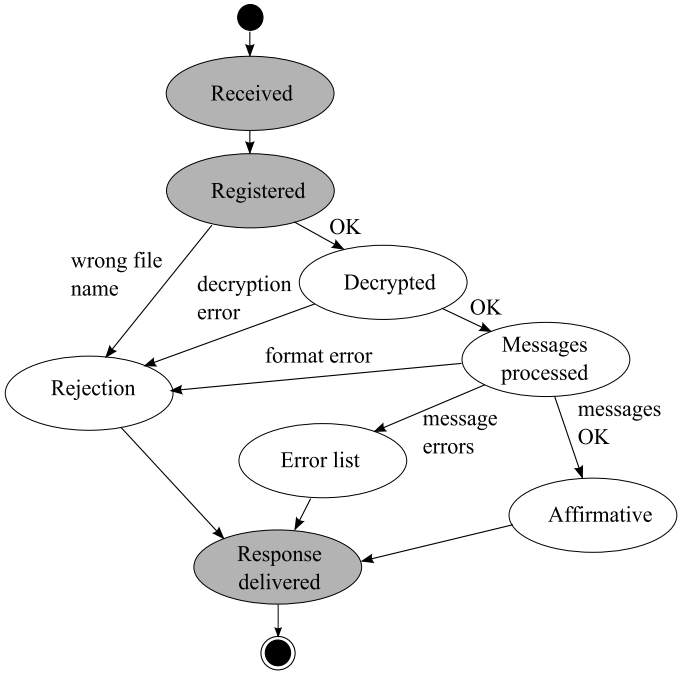

Fig.6. An example of a base process

During the file processing, the system prepares one of three types of responses: affirmative, an error list or a rejection of file procession. When the response is prepared, it is created in a temporary directory as the respective type of file, it is encrypted, and it is delivered to the recipient.

In order to verify file process execution, indicative control states must be part of the file processing state chart. When adding verification points to a process description, it is necessary to think about whether it will be possible to detect a situation in which the process has achieved a specific state. For example, it is possible to detect the creation of a new file in a specified directory or a new record in a database table, but it is practically impossible to identify object value changes in the memory (RAM) of another process. In the example that is seen above, three states have been identified as possible verification states (marked in grey in Fig. 6) – File received, File registered and Response delivered. The process verification description can be created when these states and links among them are utilized.

According to the aforementioned process verification description can be created from "zero", from the existing state chart, as well as from an activity diagram or other process descriptions. The only requirement is to add extensions such as event and time control concept.

Solution

The authors (Oditis and Bicevskis, 2015) propose a solution for business process runtime verification (see Fig. 7), that uses three objects: verification model/description, agents and controller. The basic idea of the solution is to run a separate verification process for each controllable business process (further – base process).

A verification description contains instructions about the correct execution of the base process, an agent is a software module for registering of base process execution events, and a controller compares the events received from agents with the permissible ("correct") events described in the verification model. As a result, the controller may discover the incorrect behavior of base processes. If inconsistencies are detected, the controller is sending messages to the support staff.

If the base process does not have a formalized description/ model, the verification process must be built from the scratch. If the base process is already described in a form of the formalized model, the verification process can be created from the base model by indicating those process steps which will be carried out in the runtime verification process (see Fig. 8).

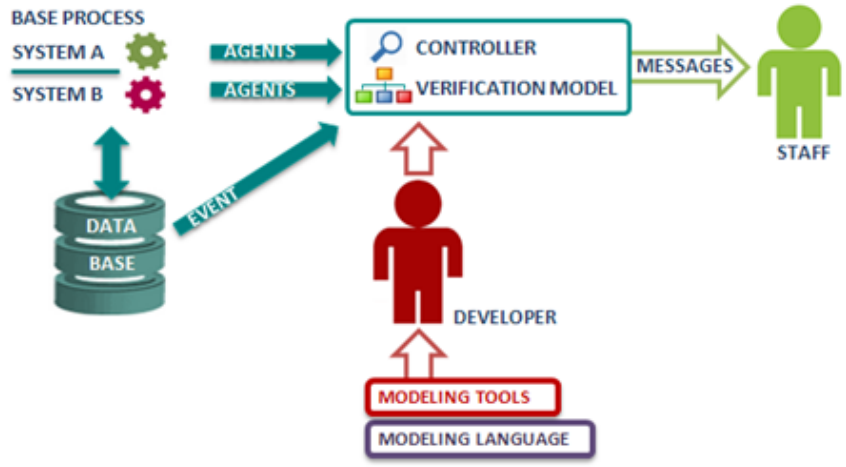

Fig. 7. Smart technology component: Runtime verification

It should also be noted that one basic process can create multiple, different process verification processes, and each of them examines their specific process steps individually.

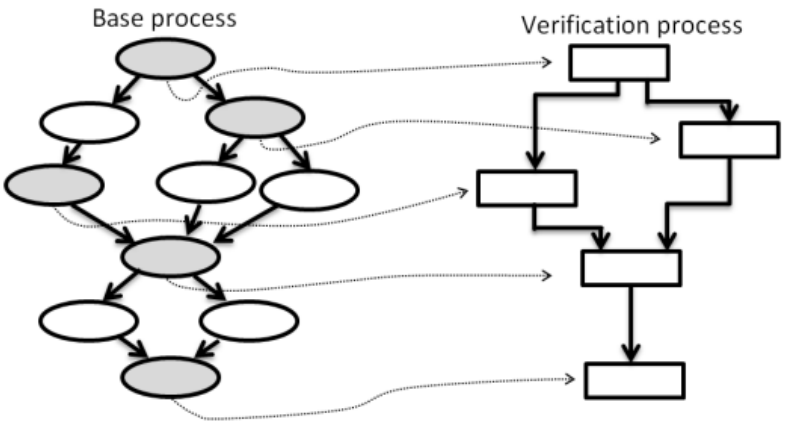

Fig. 8. The base and verification process

Results

The developed runtime verification solution was piloted in bank's electronic clearing system (ECS). The piloting results leads to two main conclusions – (1) the solution is able to detect business process execution defects, and (2) data processing system verification process creates a tiny extra load for the involved information systems infrastructure.

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

- The verification process can be defined without modifying the base process - the base process can have more than one verification process so as to verify all of its various aspects;
- The verification process runs in parallel to a base process and does not interfere with it;
- 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.

Conclusions

Several years were spent 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;
- Smart technologies have advantages for use and maintenance in at least two cases: (1) when the cooperation between the customer and the supplier is long-term, or (2) when the information systems are used by many users without profound IT knowledge;
- Building of smart technologies into information systems requires additional work. Often it is an investment, as the customers are not ready to pay for an implementation of complex mechanisms. Some of them are unable to assess the benefits offered by smart technologies;
- Smart technology enabled systems are currently not very common; due to the fact that these ideas are not popular enough yet, hence the customers don't include requirements for smart technologies into system's specifications.

The smart technologies which are described in this paper achieve the autonomic computing initiative goals only partially. There may be still a vast variety of smart technologies which would be useful to explore and implement practical systems. For instance, these would include – data quality control, access control, performance monitoring, availability monitoring which are easy enough to implement for a small/medium size organization. Additionally, we emphasize that the very usability makes the smart technologies approach different from academic researches. According to authors' experience smart technologies can be used even in a small to medium size IT company with 30-50 employees. Authors is not discussing other research directions i.e. smart spaces, smart environments, smart services and smart objects in the Internet of Things because they have not practical implementation of these technologies.

Acknowledgment

The research leading to these results has received funding from the research project "IT Competence Centre" of EU Structural funds, contract No. L-KC-11-0003 signed between IT Competence Centre and Investment and Development Agency of Latvia, Research No. 1.4 "Research of model based architecture for business processes modeling".

References

Arnautovic, E., Kaindl, H., Falb, J., Popp, R., Szep, A. (2007) 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.

Barzdins, J., Cerans, K., Grasmanis, M., Kalnins, A., Rencis, E., Lace, L., Liepins, R., Sprogis, A., Zarins, A. (2009) Domain Specific languages for Business Process Managment: a Case Study. *In: Proceedings of DSM'09 Workshop of OOPSLA* 2009, Orlando, USA (2009). Available from Internet: http://www.dsmforum.org/events/DSM09/.

Barzdins, J., Zarins, A., Cerans, K., Kalnins, A., Rencis, E., Lace, L., Liepins, R., Sprogis, A. (2007) GrTP: Transformation Based Graphical Tool Building Platform. *Workshop on Model Driven Development of Advanced User Interfaces, MODELS*, Nashville, USA. 2007. Available from Internet: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-297/.

Bičevska, Z., Bičevskis, J. (2007) 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.

Bičevska, Z., Bičevskis, J. (2008) 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.

Bicevska, Z., Bicevskis, J. (2008) Applying Self-Testing: Advantages and Limitations. *In: Databases and Information Systems V - Selected Papers from the Eighth International Baltic Conference, DB&IS 2008*, June 2-5, 2008, Tallinn, Estonia (Haav, H.-M., Kalja, A., eds.), IOS Press, vol. 187, 2008. pp.192-202.

Bičevska, Z., Bičevskis, J., Oditis, I. (2015) Smart Technologies for Improved Software Maintenance. *Preprints of the Federated Conference on Computer Science and Information Systems* pp. 1549–1554.

Bichevskii, YY, Borzov, YV. (1982) Prioriteti v otladke bolsih programmnih sistem *Programmirovanie*, 1982, vol. 3, pp. 31-34 (in Russian). (Bichevskii Ya.Ya., Borzov Yu.V.. Priorities in debugging of large software systems. PROGRAM. & COMP. SOFTWARE. 8:33, 129-131, 1983).

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

Chengying, M., Yansheng, L., Jinlong, Z. (2007) 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.

Cerina-Berzina, J., Bicevskis, J., Karnitis, G. (2011) 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), 124-135.

Diebelis, E., Takeris, V., Bičevskis, J. (2009) Self-Testing - New Approach to Software Quality Assurance. *In: Proceedings of the 13th East-European Conference on Advances in Databases and Information Systems, ADBIS 2009*, September 7-10, 2009, Riga, Latvia (Grundspenkis, J. et al., eds.), 2009. pp.62-77.

Diebelis, E., Bičevskis, J. (2011) Test Points in Self-Testing. *In: Databases and Information Systems VI - Selected papers from 9th International Baltic Conference, DB&IS 2010* (Barzdins, J., Kirikova, M., eds.), IOS Press, vol. 224, 2011. pp.309 – 321.

Diebelis, E., Bičevskis, J. (2013) 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. 249 – 262.

Draheim, Dirk (2010) 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).

Ghezzi, C., Guinea, S. (2007) *Run-time monitoring in service-oriented architectures*, in: Test and analysis of web services, Springer, 2007, pp. 237–264.

Herrmann, K., Muhl, G., Geihs, K. (2005) Self management: the solution to complexity or just another problem? *Distributed Systems Online*, 2005, 1, vol. 6.

Horn, P. (2001) Autonomic Computing: IBM's Perspective on the State of Information Technology. *IBM, 2001*. http://libra.msra.cn/Publication/2764258/autonomic-computing-ibm-s-perspective-on-the-state-of-information-technology.

Kephart, J., Chess, D. (2003) The Vision of Autonomic Computing, IEEE, *Computer Magazine* 36: 41-52, doi:10.1109/MC.2003.11600552003.

Kephart, J. (2011) Autonomic computing: the first decade. ICAC 2011: 1-2.

Lalanda, P., McCann, JA., Diaconescu, A. (2013) *Autonomic Computing: Principles, Design and Implementation*, 2013 – Springer, 288 p.

Nami, M. K., Bertels. A. (2007) Survey of Autonomic Computing Systems. *In: ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems*, 2007. p.26.

Oditis, I., Bicevskis, J. (2015) 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.

Orso, A., Harrold, M., Rosenblum, D. (2001) 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.

Rauhvargers, K., Bicevskis, J. (2009) 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), 169–179.

Rauhvargers, K. (2008) 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.

Sterritt, R., Bustard, D.(2003) 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

Tosi, D. (2004) Research Perspectives in Self-Healing Systems. *Technical report* LTA:2004:06, University of Milano-Bieocca, Milano.

Wu, C. W. W. (2013) Methods for reducing monitoring overhead in runtime verification, Ph.D. dissertation, University of Waterloo, 2013.

Received November 11, 2015, revised December 16, 2015, accepted December 21, 2015