LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 38.–52. lpp.

Towards a Semantic Execution Environment Testing Model

Krišs Rauhvargers, Jānis Bičevskis

University of Latvia, Raiņa blvd. 19, Rīga, Latvia
Kriss.Rauhvargers@bank.lv, Janis.Bicevskis@lu.lv

Abstract. The paper analyzes one component of "smart technologies" – a model for program execution environment verification that employs software meta-data descriptions of quality requirements to ensure the conformity of characteristics of the surrounding environment to those necessary. The study is based on practical software deployment and maintenance experience in areas where the production environment is inadequate and defies normal software operation. The solution is to develop a "profile" for each software item which would contain information about software requirements regarding its execution, for instance, OS version, configuration file and registry entries, regional settings, etc. The profile document is added to software deliverables together with a set of tools capable of verifying the adequacy of the execution environment according to the document. The profile document can be used in both the installation and operational phases of software.

Keywords. Maintenance, Testing, Smart Technologies, Self-healing systems.

1 Introduction

As new computing paradigms, such as distributed computing, service-oriented architecture, and business process support are emerging, software becomes more complex and more difficult to maintain. Quite much of the software being developed today is aimed to serve a single business and the customers are willing to invest as little funds as possible. Therefore, the software is built for a specific environmental platform and may have strong bonds to it.

One of the core principles of developing "Smart Technology Compatible Software" [1] is to create software that is able to analyze the external environment and, like a cognitive being, adapt to it or at least to state that the environment is not suitable for normal existence. If this kind of software were feasible, the installation process and maintenance would be greatly simplified.

The present paper analyzes the requirement stated in [1] and provides a methodology that, when properly implemented, solves the problem. Methodology discussed in the paper is based on describing knowledge about software dependencies on external environment outside the executable code, and creating a human and computer-readable document – a software profile document, based on the collected data. Using the profile document, environment may then be checked during initial software deployment and later in the software life-cycle.

The paper is further organized as follows: the next chapter describes the background of the research and states the research questions. Chapter 3 discusses work in related directions of software engineering. In chapter 4 we present our methodology and aspects to consider when implementing the methodology. Chapter 5 is a short report on first practical experiences already achieved.

2 Background

When a company reaches certain size or functional business area coverage, it may need custom-tailored software applications to be able to handle its business functions. Independently from the chosen software platform, there is a selection to be made – whether to outsource the development from other company or to develop the applications in-house. The former allows to "keep out of IT" by not requiring much involvement of company's staff in the development work, while the latter allows to be more flexible in specifying requirements and maintenance since the software code is available and can be modified upon request.

The historical basis of this paper is formed by an in-house solutions development and maintenance model where security and safety are highly significant. Because of such priorities in this model, the software is being "pushed" through testing and acceptance-testing environments before it may be installed in "live" environment. To install software at a particular operational environment (that is, any environment other than development), a special code compilation is carried out for the particular environment and then it is installed manually according to instructions given by the developers.

The manual installation model is reasonable in the particular case, since the software is designed to be installed easily and the software applications are not distributed to many users. To be more precise, in the client-server architecture, the client components are developed somewhat like "portable applications", i.e. applications that do not need client-side installation at all and are fetched from the server for execution. However, the server side of the applications has to be configured and installed manually.

All the instructions for setup and installation are prepared in textual form by the developers and are executed by people authorized to do so (systems administrators). The instructions typically include tasks such as compilation, copying executable files and libraries to servers, registering and configuring components, altering operating system configuration.

The instructions also contain information that can be treated as requirements regarding the execution of the particular piece of software – needs for other specific software items to be installed, configuration settings, file locations, specific tasks to be performed before the installation – to be summarized as "execution requirements".

The execution requirements hold through the lifetime of the particular software application, i.e., they must be satisfied whenever the software application is executed.

Upon software migration to another physical or logical environment – for instance, moving to a newer server – a series of questions arises. For instance, what

are the current execution requirements of system? What else directories should we re-create in the new environment? Which configuration file does this system use? Did we have to open any specific ports in the network firewall for this system?

2.1 Systems Interaction

In realistic environments, different software applications may be installed in the same computer. The execution requirements may differ from system to system and some of them may be contradictory. For instance, system A may require date format setting as specified by ISO 8601:2001 (yyyy-mm-dd) and, at the same time, system B may rely on American English date format (mm/dd/yyyy). Of course, work-arounds exist for the situation described, if only the problem is known during the installation of whichever system is installed later.

Software systems' integration may be necessary to avoid data and functionality duplication in systems. For instance, the organization's customer data may be shared between the CRM (Customer Relationship Management) software system as the "host" of the data and other "guest" systems such as inventory system, accountancy system etc. The integration may be carried out by using public interfaces of the hosting system – an API (Advanced Programming Interface), if such exists or by using internal mechanisms of the host system. The latter solution is technically possible only in the in-house development model and may sometimes be used.

Of course, the interface that the host system provides and the guest systems are dependent of may change by time. It is not a very likely scenario in the case of integration using public API, but quite likely when private interfaces are exploited. This leads to unpredictable effects at maintaining the guest systems when updates for the host system are deployed – in-depth regressive integration tests are required.

2.2 The Research Question

In many systems, typically those designed to support unique business process instances rather than process outcomes (documents); it is not possible to perform a "test run" of the system to verify that the system still works. Such tests could have unpredictable effect on the business process. Hence, if business processes occur rarely but are significant to their owners, there is a need for methods to verify the software system configuration and to check if it is up-to-date without running the business support applications.

Automated software installers are another option for software deployment. However, neither the maintenance model described here, nor the automated installers provide a way for checking if execution requirements of different systems are not contradictory and if all requirements are satisfied. The easiest known way is performing manual checks of all requirements. Of course, in the real world it may appear too time-consuming.

Installation package wizards such as Macrovision InstallShield or NullSoft NSIS offer features for checking pre-installation requirements. Also, they may provide a "repair" feature if the system is known not to be malfunctioning. However, this does

not allow performing a preventive checking to find out if the configuration is still acceptable.

The present paper is a study on formalization of system execution requirements taking into account the semantic nature of every requirement.

The main questions of research are:

- Is it possible to automate the verification of execution requirements?
- If the process is automated, can the requirements still be human-readable?
- Can the verification process be unified throughout the enterprise?

3 Related Work

The field of automated software testing has been studied extensively; however mainly research concentrates on testing the software internals, trying to verify that the software is built according to the specification. Formal verification frameworks such as Context UNIT and Mobile UNITY [2] have been described as well as practical implementations such as Java PathExplorer [3] are present.

The concept of an execution requirement indicates that the system under test is not aware of the particular circumstance – it will work fine if the conditions are met. The software system may have a built-in mechanism of self-protection, but it is not intended to adapt the situation.

The present research concentrates on aspects of the execution environment that the system is not aware of.

3.1 Self-Healing Systems and Built-In Tests

The topic of self-healing systems can be considered relatively close to the research topic of this paper. Both research topics share the same goal – a system working at an operating environment that is known not to be perfect.

A comprehensive study in the field of self-healing systems by D. Tosi [4] identifies and analyzes the key elements of a self-healing system.

The concept of self-healing software is defined as a system that monitors the surrounding environment at run-time, detects failures and, knowing its “normal” way of operation, can “heal” itself. Such a system can then be called a “fully autonomic system”, which is, according to classification of [5], the highest (5th) level AS (Autonomic System). Such systems are far beyond the scope of current research which concentrates on practical support for transition between 2nd level (Managed Level[1]) and 3rd level (Predictive Level[2]) of autonomic systems.

[1] Managed Level: At this level, system management technologies can be used to collect information from different systems. It helps administrators to collect and analyze information. Most analysis is done by IT professionals. This is the starting point of automation of IT tasks.

[2] Predictive Level: At this level, individual components monitor themselves, analyze changes, and offer advice. Therefore, dependency on persons is reduced and decision making is improved.

In D. Tosi's paper [4], the self-management objective is decomposed into a number of dimensions: requirements (security, performance), monitoring/detection (includes self-adaptation, self-optimization, and self-healing), and repair. According to this decomposition, the current paper is a research on monitoring and detection, specifically on system's reaction to changes in the runtime environment.

The author of [4] also mentions the need for a knowledge base that describes the "normal" environment of the system.

In [6] Wang et al. present a concept of software component with built-in tests. A built-in test (BIT) is a set of code functions that perform verifications, if the component is working as predicted. Authors of [6] suggest that a component should contain one or more BITs that can be executed; and implement some specific interface to be called when the system is run in maintenance mode. Here, the authors define a concept of "maintenance mode" [6] for a software system, which is proposed as a special execution mode of the system when built-in tests can be activated but the business functionality of the system is not touched.

The authors of [7] propose a framework named Component+ that mainly focuses on benefiting from the use of BITs in software components. Authors note that every component implements its fixed interface which other components may rely on. This can be called a „contract" between components. Authors of the paper suggest that BITs are used for testing, whether the component is capable of serving the defined interfaces, i.e. if contracts between components are not violated. It is remarkable that the authors mention the importance of „Quality of Service" (QoS) testing for verification of the operating environment, but their proposed model concentrates on testing the contracts between the components, hence, QoS testing is a „by-product".

3.2 Unit Testing and Test-Driven Development

At a first glance, the test-driven development [8] seems to be a good solution for the research problem. Though unit testing technique has a great effect at improving software quality and results at integration testing, trying to use the unit tests for validating the surrounding environment would be a misuse of the particular technology. This is because unit testing is designed to focus on testing a single unit. According to the guidelines, a unit test shouldn't pass the boundaries of the unit to be tested. To overcome the need for "outside world" reactions during tests, the unit testing model proposes the use of mock objects and fake objects [9]. Both kinds of objects are used to simulate the interface of external units required by the unit under test. The use of such substitute code indicates that unit testing may not be useful for integration between components, and, moreover, integration between the system and the surrounding environment.

3.3 Similar Ideas in Hardware Appliances

A similar topic has been investigated quite profoundly in hardware engineering. Runtime checks of the surrounding environment seem to be natural and have been studied extensively in hardware world. Research is being carried out in different areas – both on self-tests of internal state of the item, for instance, for a CPU or a

RAM module, during the stand-by time [10], and on overall inspections of the surrounding environment, i.e. parts of the "hosting" system. A system embedded in a car is a good example. Upon initialization, the central processor performs a comprehensive examination of all the distributed sensors and sub-processor units; if any of them fails, it shows a warning sign to the driver or even does not allow starting the engine.

Similarly to software systems, nowadays the hardware systems have also become component-based. For instance, a digital photo camera relies upon a specific type of interface for its memory chip (e.g. Compact Flash type I cards supported only). Upon initialization, the camera checks whether the card is currently available, the type and the capacity of the flash card. The user may be prompted "Error, wrong card inserted!", "CF full". This can be compared to the "contract testing" described in [7].

The idea of the current paper is similar to the methodology described here - verification if the external components the system under test relies on, fulfills their duties.

4 The Proposed Model

4.1 Model Summary

Methodology proposed by this paper is based on describing knowledge about the software system outside the executable code (as some sort of meta-data about the system), and using the collected data for creation of a computer document – a software „profile". Such document is later used for execution requirement testing.

A software requirements profile document contains listings of both the internal links between software components and dependencies on external services, facilities and interfaces. Using appropriate tools, the profile document can be employed for different purposes throughout the life-time of the system. The present paper describes the usage of software profile document as the core of execution environment verification by external verification modules – small pieces of software aimed to perform verification routines (further in text – EVM), but other applications may exist.

The following model is offered:

1. bonds of the program code with the environment are registered as software metadata during development;
2. upon finishing development of a particular software item, a software profile – a document summarizing execution requirements for the particular item is generated from the code meta-data;
3. the software item is delivered into other execution environment together with its profile document;
4. conformance of the execution environment is verified using data from the profile document.

A summary of the model is shown in Figure 1

The described situation applies to internally stable [1] systems. That is, the system under test has already passed the integration tests in the development environment and is known to be working under certain circumstances.

There are two areas of software life-cycle where the software profile methodology is useful – the initial verification of execution requirements during software installation into an environment where the software has not yet been used, and the routine re-validation of run-time requirements every time the software application is executed. Re-validations take place in the same way as initial validation; the only difference is that the user is not prompted for initiation data.

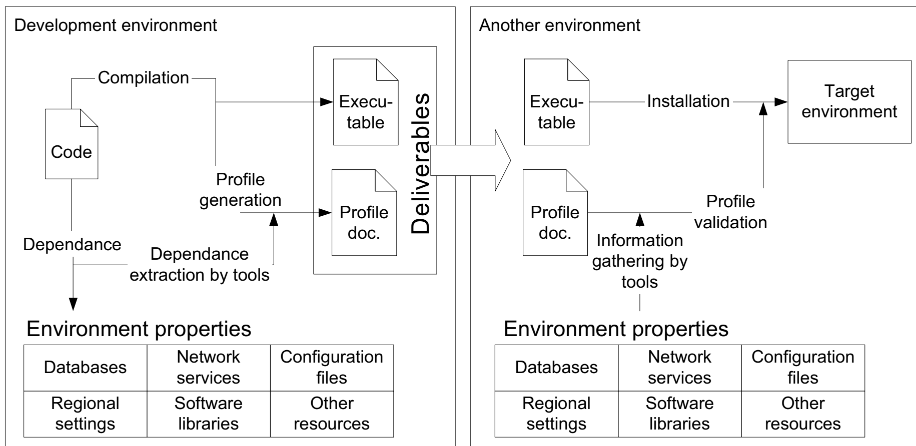

Fig. 1. The Tool-Driven Process of Software Profile Generation it the Development Environment and Use in Other Environments.

4.2 Motivation for the Proposed Model

The model proposed in the paper helps overcome several shortcomings of other related technologies that could possibly be used to achieve the same goals.

When compared to built-in tests, the software profile methodology has the advantage of test descriptions not being encoded in the system itself. As a result, the knowledge is not hidden from software maintainers and the list of known requirements may be supplemented without recompiling the software.

The contents of software profile document should rather be treated as descriptions of software properties than test descriptions, and hence the tests can be more flexible.

It is possible to execute different tests per single requirement to verify different aspects of the requirement. It is also possible to replace the test algorithm if and when needed. The scenario is quite likely since the properties of environment may change. For instance, the software may require that “Outgoing TCP traffic on 80 shall be allowed on the computer”. Verification of the requirement in a program-

matic manner relies heavily on the kind of firewall software installed on the particular computer and hence, upon changing the firewall software, the EVM should also be replaced with an updated version.

4.3 General Architecture of Runtime Validation Framework

As recommended by [6, 7], runtime validation should not interrupt the regular work of the system under test. Our model conforms with the thesis completely and is designed to separate not only the regular and maintenance modes, but also the executable code.

The software profile approach is different from a typical BIT architecture in a way that the tests are actually not embedded in the system. To enable a component for self-testing of the execution environment, only functionality for loading the software profile validation runtime („loaders") have to be encoded into the system.

When software is run in maintenance mode, the loader functions are used for loading the verification runtime core and handing the execution control to it. As the core is loaded, it looks for the software profile document of the system under test and analyzes it. Knowing the EVMs currently available for testing specific requirements, the core parses the software profile document and invokes an appropriate test routine using a specific EVM (or multiple modules) on each requirement listed in the document.

Hence, the model relies strongly on the dynamic loading feature of the execution environment. The feature is first employed to load the verification runtime core module and later, to load the specific verification tools. The required effect can be achieved easily in today's execution frameworks - using the reflection and dynamic load feature of .Net framework, using the ClassLoader interface of Java, even easier in scripting languages such as PHP or Python.

4.4 Execution Requirements and External Verification Modules

In the context of the paper, an execution requirement is a verifiable demand statement about the execution environment that typically has some human-understandable meaning and that holds through the lifetime of the system. For instance, a requirement may be formulated as "TCP traffic to host 192.168.1.1 on port 21 must be allowed" or "Write access is necessary to directory /home/$USERNAME".

Other typical types of execution requirements include:

- Other components – versions, names, availability. An application typically depends on one or more external code libraries to be available - for instance, XML support, specific database drivers, MS Excel object model API etc.
- Configuration files – INI files, Windows registry, .Net framework configuration files. The requirements of this kind typically ask for a configuration file to be located at a specific location or for values to be set for specific keys.
- File system access – requirements regarding existence or non-existence of specific nodes in the file system, permissions on file shares, etc.

- Network dependencies – requirements on protocols, ports, required network locations to be available.
- Relational data bases – requirements, specific to DB vendor, defining a dependence of the software system upon a certain data base. The requirements may include database locations, names, and requirements for existence of certain DB objects (tables, stored procedures, functions) or even the interface definitions of DB objects. For instance, one may require that a table "CUSTOMERS" having a field "NAME" exists in the database.

This list can be continued and is by no means limited to kinds of requirements listed here. Nearly every technology used in contemporary software development has some properties that may be significant for a software component employing the technology. For instance, both Microsoft's COM+ or Java's Hibernate (and of course, other products, too) services allow the configuration of distributed transactions' isolation level to be set declaratively [11, 12]. In both cases, a component relying on the particular technology can claim for a specific setting for distributed transaction configuration. Such a claim can be formulated as an execution requirement.

It is also advisable to define business system-specific requirements, that is, requirements that are useful only in the context of the system under test. For instance, "at least one administrator account must be present in the users' registry" or "all `inbox` folders of the system should be user writable" are system-specific, as the concepts of "administrator account", "user's registry" and „inbox folders" achieve their semantic meaning only in the context of the system. One should also provide EVMs that support verification of such requirements. This kind of EVM is actually a built-in test that is externalized from the system, but it is more concerned to checking if other components are in order rather than checking if the component itself is fulfilling its contracts. The verification modules used for this kind of tests can be bundled in the same code assemblies as the business system, and hence be versioned together with the system. Such approach allows the tests focus more on the internal stability [1] of the system and complies well with the component built-in testing as described in [13].

A specific kind of requirements is the transitivity requirement which can be read as "I will do my job if some other particular component does what it is supposed to do"; that is, a requirement for another component or application to pass the verification process successfully. For instance, in the client-server model, client components may not be able to perform their duties if the server component is not configured properly.

To complete the list of vital elements of the software profile framework (hereinafter - SWPF), one must mention the EVM – external verification modules.

An EVM is a small functional component that encapsulates logic for verifying one or a few execution requirements. Technically, an EVM looks for evidences in the execution environment and hence decides if the particular requirement is satisfied. The nature of an EVM should be similar to the way humans would verify the requirements – first analyzing the requirement and deciding what evidences are required to be sure that the requirement is fulfilled, second – performing the verification. Different kinds of evidences may exist and it is a task for the EVM developer to decide which ones are satisfactory.

For instance, when developing an EVM that verifies if a specific Windows security patch is installed in the system, the first idea would be to use Windows Management Instrumentation[1] to read the list of all patches installed and then look through the list to see if it contains the name of the needed patch. However, this may be a bit tricky, because WMI may not be installed on the particular computer or may not itself be updated and hence the specific function for reading the patch list may not work. When looking for another approach, one will notice that every patch installation results in a new "uninstall" directory created in Windows installation directory, for instance "$NtUninstallKB825119$" where KB825119 MS is a knowledge base number for the patch. Hence, one may consider that the existence of such a directory is a good enough evidence for validation.

A clear distinction must be set between validating a requirement in whole and validating the evidences for a requirement. There exists an N:N type of relation between the two (validating a requirement may mean looking for N evidences, a single evidence may refer to different requirements). The resolution of requirements into evidence searches is a task of the SWPF core, but technical implementation details are out of the scope of the present paper and are a question of further research.

4.5 Kinds of Data Employed by the SWP Framework

To achieve the functionality described in the previous chapters, different kinds of data are required for the software run in maintenance mode.

The most obvious knowledge base is the software profile document itself. It supplements the system under test and is delivered into particular execution environment together with binary deliverables or source code. The knowledge – listings of execution requirements - is first summarized by the developers and can later be complemented by other parties involved. The data format for storing requirements is not dictated by the methodology and may vary upon implementation; however a possibility to store complex data structures is essential. Other requirements for the description language are modularity and possibility to extend the available markup as new kinds of requirements may appear. Hence, we propose the use of XML as the carrier and XSD as a validation tool and reference. An in-depth study of the format is a question of further research.

Another knowledge base employed by the methodology, is the „inventory" list that belongs to a particular execution environment. The list contains information about the EVMs that can be used at the environment. Since the EVMs should be designed to handle verification of requirements as specific as possible, the full spectrum of EVMs may be quite ample. For instance, it includes a wide variety of EVMs that handle requirements specific to a single business system only (e.g., an EVM for a requirement "Version 2.9.4 of the HR system shall not be present at the environment"). Not all the EVMs owned by the organization may be needed at every particular environment and hence "inventory list" should be bound to the environment.

[1] Windows Management Instrumentation (WMI) is the Microsoft implementation of Web-based Enterprise Management (WBEM), which is an industry initiative to develop a standard technology for accessing management information in an enterprise environment (MSDN library, 2007)

Another task for the inventory list is to tell the verification runtime, when to use the particular EVM described in the list. That is, the list describes patterns to look for in the software profile document that are related to the particular EVM. In the XML-based form of software profile document, the names (and the namespaces) of the XML elements recognized by the EVM should be depicted in the inventory list.

4.6 Creating a Software Profile Document

The software profile notion is essential for systems which are developed in a different environment than the operational one (where the concept "operational environment" includes development, testing and/or acceptance testing environments, production environment).

It is assumed that the system under test is internally stable, i.e. that system developers have assured that the system is functioning and have managed to run the system in the development environment. This should be true in order to allow transition to the testing phase. Since the satisfactory requirements are met in the development environment, it can be a good sample for gathering the requirements.

The gathering of data required for generation of the software profile should be begun as close to the beginning of information system life-cycle as possible in order to minimize the documentation work to be done close to the delivery date.

A good time span for registering requirements is the coding phase when detailed system design specification is transformed into executable code. All kinds of dependencies become known to the developers during this phase: the ones dictated by the business problem (declared in the requirements specification document), the ones discovered during system planning (functional specification, class, and component diagrams) and the technical limitations that have arisen due to development methods, organizational standards, etc.

The methodology anticipates that requirement descriptions needed for the creation of the profile document are gathered from source code where it has been previously entered by the coders.

A substantial part of dependencies is known even before the coding phase i.e., during the design phase. If software is developed using model driven development approach, requirements can be recorded at an earlier phase than development. Dependency information could be attached to the model as object stereotypes if a UML model is used. During the PSM (platform specific model [14]) transformation to program code, the stereotypes would be transformed into code meta-attributes as described further in this chapter. Hence, the initial dependencies would be recorded even before the coding phase begins.

.NET framework provides a convenient way for describing code meta-data. It is called "declarative attributes" – a supplementary information block that is assigned to a particular code class, function or the software assembly all together. The .NET framework allows defining one's own attribute classes to extend the set of available declarative attributes.

Hence it is possible to define one or more declarative attribute classes for each type of requirements and use the attributes to describe the code.

For instance, an attribute class "*NetworkDependency*" with parameters *protocol*, *direction,* and *port* can be defined. When creating a program which depends on net-

work services, a function *ReadData* that accesses the network (or the class containing the *ReadData* function) can be assigned the declarative attribute:

```
<NetworkDependency(Protocol.TCP, Direction.Out, 80)> _
Public Function ReadData()
```

Upon preparing the system for delivery, a tool for dependency extraction would scan the program code and find the meta-data attribute attached to the function 'ReadData()'. As a result, a record line would be created in the software profile document.

Most parts of the software profile document can be generated by using the following approach:

- data needed for the profile document header part are encoded as meta-attributes of the whole .Net assembly (or analogous concept in different platform, for instance, a component in terms of Java technology)
- dependencies on execution environment, i.e. requirements, are described as declarative meta-attributes of the code object which demands particular dependency, or at a higher hierarchical level if fine-grained requirements do not matter. Part of the requirements may be generated using MDA tools.
- all requirements listed in the program code are gathered by help of specific code analysis tools
- the requirements list is reviewed:
  - dependencies that do not require more information are instantly transformed into requirements
  - for dependencies referencing the development environment (for instance, a requirement "Short date format – the same as in development environment") specific tools are used to get the requested information from the environment (in case of the example – a function that queries the operating system for the short date format)
- the gathered list of requirements is merged with requirements from linked code libraries' profile documents, using the minimum supplement approach
- the obtained software profile document is reviewed by the developer to eliminate redundant requirements and to add the missing items.

5 First Practical Experience

Initial practical development has already been carried out according to the methodology provided in the paper. The development has resulted in a fully functional proof-of-concept version of the software profile verification toolset.

The demonstration software was developed using Microsoft .Net framework 2.0 platform. The chosen framework supports dynamic loading of modules from user code, which is a significant requirement for a good implementation of SWPF.

The code was separated into code assemblies according to the architecture of SWPF: the "business application program" which can be launched at maintenance

mode (1), core module of SWPF (2), and an assembly containing implementations of some typical verification modules (3).

When the program (1) is loaded, it enters the "maintenance" mode and performs environment testing. To do the testing, it loads the SWPF core assembly (2) which further handles the tests. The SWPF core looks for an adequate software profile document. It considers that the software profile file is located in the same directory as the (1) executable files and named according to format *<executablefile>.swp*.

The *SWP* file is an XML file containing assembly identification information (for ensuring that the application currently under test is the correct one) and a listing of requirements each described using an XML element. The profile document used in the experiment was created manually.

When the SWP document is loaded, it is parsed into individual requirements. In the conceptual model we have introduced support only for two very simple, but potentially useful used requirement types:

- Regional settings – short date format requirement. In Latvian grammar, the format "dd.mm.yyyy" is advised. However, in Microsoft Windows the default locale settings are different and short date format is provided as "yyyy.mm.dd". The format string is frequently being changed during installation of the OS to the grammatically correct one; hence the actual setting in deployment environment may vary. In an isolated enterprise environment, the setting is typically the same on all computers and therefore the systems developed in-house are more likely not to be aware of the possible differences.
- File system – checking if specific path exists. In our experiment, the path existence evidence was used to check if "Windows XP service pack 2" is present in the system. The service pack was known to install itself at a specific location in the file system.

To find the appropriate tool for testing a particular requirement, the SWPF uses an "inventory list" – another XML file describing locations of EVMs. The conceptual model does not introduce distinction between requirements and requirement evidences; it assumes a 1:1 relation between requirements and EVMs (an EVM may handle one type of requirements). Therefore, items in the inventory list have a property describing the name of the requirement XML node that the current EVM can handle. When an appropriate EVM is found for evaluating a requirement, the assembly containing EVM's code is loaded and the requirement information is handed to the EVM object which further evaluates the requirement.

A test run of the application was performed on several computers in different environments and it was found out that even such a trivial environmental test may show inadequacies on some computers. Some of the computers tested did not have the required short date format, all had Windows XP SP 2 installed.

6 Conclusion

The paradigm of the software execution profile document is a step towards the development of smart technology compatible software. The methodology can be adapted in a fully manual manner – by composing the requirements profile document based on the know-how obtained during development and later using the document to check if the installation environment conforms to the requirements (this process can be reduced to textual installation descriptions as used by classical methods). However, the full power of the methodology provided can be gained when specific tools are used to generate the profile document and to validate the execution environment.

The use of the software profile concept described in the paper – verification of execution environment during installation and at run-time – is not the only one possible. Some of the other applications are:

- documentation used for systems maintenance
- discovery of cross-system bonds without inspecting the environment
- environment clean-up upon disposal of an outdated system (or previous version of the system)
- by creating a centralized registry of software execution profiles, it can automatically be perceived as a registry of available resources which allows answers to maintenance questions such as "Is this database still in use?", "Why do we have to have port 80 open on the external firewall?"
- the ideology of the software execution profile can be applied not only to executable code programs and libraries, but also, for instance, SQL "applications"

The solution for testing execution environment provided in the paper can be implemented incrementally – by expanding the set of resource types that can be verified.

The first practical experience in applying the described methodology has already been achieved and indicates that the approach is suitable for practical use.

Acknowledgements

The research is supported by the European Social Fund (ESF).

References

1. Bičevska Z., Bičevskis J.: Smart Technologies in Software Life Cycle. In: Münch J., Abrahamsson P. (eds.): Product-Focused Software Process Improvement. Lecture Notes in Computer Science, Vol. 4589. Springer-Verlag, Berlin Heidelberg (2007)
2. Roman, G.-C., Julien, C., Payton, J.: A Formal Treatment of Context-Awareness. In: Wermelinger, M., Margaria-Steffen, T. (eds.): Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, Vol. 2984. Springer-Verlag, Berlin (2004)

3. Havelund, K., Rosu, G.: An Overview of the Runtime Verification Tool Java PathExplorer. Formal Methods in System Design Vol. 24(2), pp 189-215 (2004)
4. Tosi, D.: Research Perspectives in Self-Healing Systems. Report of the University of Milano-Bieocca (2004)
5. Nami, M., R., Bertels, K.: A Survey of Autonomic Computing Systems. In: ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems. IEEE Computer Society. Washington, DC, USA. (2007)
6. Wang, Y., King, G. Wickburg, H.: A Method for Built-in Tests in Component-based Software Maintenance. In: Proceedings of the Third European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA. (1999)
7. Barbier, F., Belloir, N.: Component Behavior Prediction and Monitoring through Built-In Test. In: 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03). IEEE Computer Society, Los Alamitos, CA, USA (2003)
8. Janzen, D. Saiedian H.: Test-Driven Development: Concepts, Taxonomy, and Future Direction. IEEE Computer. September 2005 (Vol. 38, No. 9) pp. 43-50 (2005)
9. Kim, T., Park, C., Wu, C.: Mock Object Models for Test Driven Development. In: SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications. IEEE Computer Society. Washington, DC, USA (2006)
10. Shamshiri, S., Esmaeilzadeh, H., Navabi, Z.: Instruction Level Test Methodology for CPU Core Software-Based Self-Testing. In: HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International. IEEE Computer Society. Washington, DC, USA (2004)
11. Troelsen, A.: Developer's Workshop to COM and ATL 3.0. Worldwide publishing, 2000
12. Hibernate Reference Documentation http://www.hibernate.org/hib_docs/reference/en /html/session-configuration.html
13. Beydeda, S.: Research in Testing COTS Components - Built-in Testing Approaches. In: ACS/IEEE 2005 International Conference on Computer Systems and Applications. Bonn, Germany (2005)
14. Kleppe, A., Warmer, J. Bast, W.: MDA Explained: The Model Driven Architecture--Practice and Promise. Addison Wesley, (2003)