DOI: 10.15439/2015F142
ACSIS, Vol. 5

Approach to Building a Web-based Expert System Interface and Its Application for Software Provisioning in Clouds

Evgeny Pyshkin
Institute of Computing and Control
Peter the Great St. Petersburg Polytechnic University
St. Petersburg, Russia, 195251
Email: pyshkin@icc.spbstu.ru

Andrey Kuznetsov
St. Petersburg Software Center
Motorola Solutions Inc.
St. Petersburg, Russia, 192019
Email: andrei.kuznetsov@motorolasolutions.com

***Abstract*—This paper focuses on a generalized approach to providing user interface to a web-based expert system (WBES). We examine MVC and MVP design patterns used traditionally to construct a web application user interface. In order to leverage the strength of the MVC/MVP design patterns we propose a special ontology representing a user communication domain. We describe a self-service networked infrastructure for automatic deployment of command line interface (CLI) applications. We demonstrate how to apply the proposed ontology for the design of a WBES aimed at supporting client software re-execution in clouds. In particular, we address the problems existing in the area of software development for music information retrieval algorithms implementation.**

I. Introduction

Pervasive nature of modern software is a popular subject of the present-day technology discourse, whether the question concerns computer-assisted education, interface design, usability or elementary forms of programming, which are one of necessary elements of modern information literacy. In particular, service-oriented software and cloud technology significantly transformed the way we use computing and storage capabilities.

There is a constant interest to organizing processes of research software distribution in order to make computational and data resources available for other users. For example, in the domain of information retrieval (IR) many developed approaches are semantic relatedness centered. The focus of such works is on developing algorithms for better semantic relatedness evaluation, semantic classification or clusterization. An obvious way to evaluate an IR algorithm is to use various test collections, while an algorithm itself might be implemented in the form of a computer program. However, those programs often remain unpublished. In some IR domains, particularly, in music information retrieval (MIR), even if a software implementation is reusable, test collections might not be available for a third party researcher either for the reason of their big size or due to the copyright restrictions. That's one of the reasons explaining difficulties of comparing or reproducing results achieved by other researchers. From the study [1] we know some statistics of what MIR researchers are as software developers. The figures are rather discouraging: 82% of researchers did develop software, but only 39% of those took steps to achieve better reproducibility. It is no wonder that only 35% of those developers published any code, whereas 51% said their code had never left their own computer. Often the only way to follow is to believe in the results reported in papers without any possibility to be sure that the reported results came from a research method, and not from bugs in the software. Thus, researchers often re-implement algorithms based on the published descriptions; such re-implementations are not often executed in the same context as it was for the original software.

In [2] the best practices are examined, which are aimed to improve research reproducibility. Among them there are such ones as using collaborative platforms (like *Github*) and resource sharing mechanisms in order to lower the barrier to reproduce the third party work. In [3] a platform for re-executing software in the same context is described. That solution is based on capturing files and environments required for an experiment, and building an archive with subsequent software re-execution on a third party machine.

In [4] the authors took the significant step toward research reusability and reproducibility in the domain of machine learning. They developed and maintain a networked system (OpenML) for sharing and organizing data sets and algorithms solving the typical machine learning tasks. In our work we pay attention to two broader aspects of the reproducibility problem. Firstly, it might not be permitted to distribute the source code, the binary files or the datasets due to the legal issues. Secondly, due to manifold existing supporting tools (such as version control systems, build systems or runtime environments) it is not easy to configure a local environment in a way to be capable to run a third party software not limited by the specific data and task types.

A possible solution addressing both mentioned issues is to distribute both software and datasets as services accessible via a standard client (e.g. a web browser). In such a case neither data copying, nor environment configuration is required. However, distributing algorithms and datasets as services is far from being a trivial problem. There is a certain similarity with Milne and Witten's consideration on data mining. Researchers who want to use Wikipedia as a knowledge source have two major options: either to base their work on secondary structures, or to build their own algorithms from scratch [5]. There are difficulties to share the algorithms due to lack of supporting platforms. We think that the similar situation exists in the domain of MIR.

Clouds can serve as a platform to share algorithms as services. One significant problem is a relatively high barrier to entry for non-experts. In our earlier work we described sources, advantages and problems of deploying research software in clouds and proposed an architecture of a provisioning service for automatic CLI applications deployment in computing clouds [6]. The proposed solution targets problems of build and run error discovery and handling, with special emphasis on errors conditioned by possible misconfiguration of a virtual platform where client software modules have to be executed.

The remaining text is organized as follows. In section II-A we discuss the process of CLI software deployment and provide a brief survey of existing approaches in order to explain their limitations and constraints (conditioning difficulties of provisioning applications as services). For the reason that we consider using expert systems as one of possible ways to overcome such difficulties, in section II-B we pay attention to a web based expert system (WBES) user interfaces. In section II-C we analyze a state-of-the art example of a web based expert system and evaluate its architecture with reuse and change scenarios. We discover that the modification of a model requires changes in all the model-view-presenter (MVP) architecture components, which, in a sense, goes against the motivation to use MVC/MVP pattern in design. In section III we examine two approaches to define a model interface: a domain *aware* approach and a domain *agnostic* approach. In order to better separate a presenter from a model, we propose to complement a subject domain ontology with a user communication ontology. For the problem of provisioning software in clouds, we define an ontology of software provisioning partially described in section III-A as well as a user communication ontology (section III-B). In section IV we describe a software provisioning self-service networked infrastructure, its architecture and its major components. In section V we demonstrate how the proposed approach helps in developing a web-based expert system for CLI applications deployment in computing clouds. We list some experiments we arranged in order to evaluate the knowledge based approach of using introduced networked infrastructure for provisioning a series of projects developed in the domain of MIR. We compare the knowledge based approach with the other existing implementations (section V-A) and describe a scenario based architecture evaluation process (section V-B).

II. Related Work

In order to position our work within the framework of the service distribution domain we have to examine three major issues. First, we attempt to have a look at existing systems for deploying CLI applications in clouds with respect to the user expertise necessary to use such systems properly. Second, we analyze existing works on expert system user interface development with special attention paid to WBESs. Third, we analyze an emerging problem of providing a web-based user interface of an expert system to end users: specifically, how to apply MVC or MVP design patterns (widely used in web application architectures) for a WBES.

A. Deploying a CLI Application in a Cloud

Research software (especially in the MIR domain) is often developed as desktop applications which primarily were not intended to be executed in networked or distributed environments. This aspect causes difficulties of their deployment in clouds without significant changes in software code. For example, an *OpenShift PaaS*[1] provides two ways to deploy an application in a cloud. The first way is to develop a custom cartridge, while the second one is to develop a module for an existing cartridge (e.g. *JavaEE* cartridge, *Python* cartridge, *Ruby-on-Rails* cartridge, etc.). Unfortunately, both approaches seem to be unsuitable for deploying CLI applications having no any networking capabilities. In order to support networking features without modification of an existing application, a proxy component is required [7]. The reality is that a deployer must be provided with the exact configuration describing a runtime environment. If the configuration is not valid (for example, an incorrect *Python* version is selected) users get unrecoverable deployment errors. In order to recover deployment errors automatically we could use such approaches as *AutoBash* [8].

In our earlier work we described how to extend the *AutoBash* approach by applying knowledge engineering formalisms [9]. We designed a proxy architecture which, in turn, is an enhanced *MEDEA*[2] proxy where a knowledge base is leveraged to control deployment and execution processes. In fact, the system described in the following sections of this paper can be considered as a web based expert system helping users to deploy a CLI application both in a cloud and within a desktop environment. Thus, the idea is to make deployment and invocation process as easy as uploading applications and datasets via web forms. Specifically, in the MIR domain the main algorithm experimentation scenarios are the following:

- A researcher wishes to test his/her algorithm by using one of the standard test collections, which might not be publicly available;
- A researcher wishes to compare the algorithm to other algorithms by using the same corpus;
- A researcher wishes to test a third party algorithm by using his/her own test corpus;
- A researcher wishes to test the corpus by running third party algorithms.

In the above mentioned work [9] we analyzed two popular software platforms facilitating the above mentioned scenarios:

[1]http://openshift.redhat.com
[2]MEDEA – Message, Enqueue, Dequeue, Execute, Access

one used in MIR, while the second used as a software deployment infrastructure.

NEMA[3] [10] provides access to the MIR software and data sets through the Internet. The environment was developed in 2008–2010, just about the period when the very first cloud service commercial implementations appeared. In fact, the *NEMA* provides a platform as a service (PaaS) and allows provisioning client software as a service (SaaS). The NEMA uses a set of preconfigured virtual machine images; each image provides a platform (e.g. *Python*, *Java*, etc.), which is completely configured to be used by the *NEMA* flow service. Storing a large set of custom images is expensive, moreover, any image modification has to be done manually; that's why using *NEMA* in public clouds is not easy.

The *MEDEA*[4] [7] infrastructure enables the deployment of an arbitrary CLI application on an arbitrary cloud platform. The *MEDEA* uses standard virtual machine images provided by a cloud and uploads a special wrapper (a task worker, in *MEDEA* terms) to the running virtual machine. The wrapper initializes the respective execution environment (*Python* or *Java*, for example) and then executes a client application. If we consider *MEDEA* as a self-service platform (within the context of MIR), there are two issues to be observed. Firstly, MIR applications often have dependencies on third party components and libraries, therefore, a wrapper might not initialize the environment properly. Sometimes researchers are unable to upload these libraries due to certain license restrictions; sometimes they don't know how to create an application package containing all the required dependencies. Secondly, there are MIR test collections which are not publicly available, so the code executed against those collections shall be considered to be "unsafe", and shall be executed in managed way in order to avoid dataset leaking. Let us mention that within the framework of the proposed architecture we address both issues by introducing a deployment manager that includes a wrapper component as it is described in the following sections.

The special case is *MIREX*[5], which is not a platform but an organization providing a service for testing algorithms delivered by its creators. In addition to the published test collections, some "secret" test collections are also used, and the evaluation process is arranged in the form of an annual contest. Thus, researchers have to wait for the results till the next competition, hence, this is not a way for everyday use.

B. Expert System User Interfaces

As mentioned in [11], "Expert systems is a branch of Artificial Intelligence that makes extensive use of specialized knowledge to solve problems at the level of a human expert." As a result of Internet evolution and telecommunication tools development a new type of expert systems appeared: Web Based Expert Systems (WBES)[12]. In our work we pay attention to one important aspect of WBES design, i.e. user communication with WBES.

The focus of the majority of current works on expert system user interfaces is on the functional requirements (e.g. on capabilities the UI does provide and the reasons for them). Examples of such requirements are listed in [11]. We also found research works focused on a particular expert system development containing some screenshots of an expert system user interface and a discussion on the system architecture (see [13] for instance). Fewer works are focused on WBES development process [14]. We only found few works discussing a level of workload separation between a server and a client of an expert system [15]. The same is true for architectural issues of providing a user interface for a generic expert system [16], [17].

In [18] the authors realized that existing approaches to evaluate an expert system are connected mostly with the rules evaluation, paying less attention to user interaction issues. Some researchers complain about "a lack of a general methodology for developing web-based expert systems" [12] and notice that "web sites that enclose an expert system have been developing ad hoc and their developers do not follow any systematic method or process" [19].

C. A Case Study: Web Based Expert Systems and Design Patterns

In order to better understand WBES user communication issues, we studied one of the rare state-of-the-art examples of the WBES where there is a discussion on WBES architecture and WBES-related design patterns.

For an end user, a WBES is presented as a web application. As we can see from many works (see [20], [21] for example) a common way to implement web applications is to use a model-view-controller (MVC) pattern [22] or its generalization known as a model-view-presenter (MVP) pattern [23]. However, using MVP might present a problem if we evaluate an architecture by using any scenario-based method (for example, *SAAM* [24]).

Let us think, for instance, of an MVC-based solution for an automatic price negotiation proposed in [25]. The authors suggest we use a production knowledge base with an inference engine as a *Model*, while the generated HTML pages are *Views* and a mediator component is a *Controller* (see Figure 1). As it is well known, if we follow an MVC pattern we expect to have such an advantage that each of the three structural components (e.g. a *Model*, a *View* and a *Controller*) can be modified independently. It allows to improve such software quality properties as reusability, modifiability and reduce a ripple effect appearing if one of the components changes significantly [23]. Let's evaluate this statement with the following scenarios:

1) *Reusability*: we can change an expert system domain from price negotiation to CLI application execution. If we consider this rather substantial change, we have to modify the *Model* (despite keeping the inference engine,

[3]NEMA – Networked Environment for Music Analysis: www.music-ir.org/?q-nema/overview

[4]MEDEA – Message, Enqueue, Dequeue, Execute, Access

[5]MIREX – Music Information Retrieval Evaluation eXchange: http://www.music-ir.org/mirex/

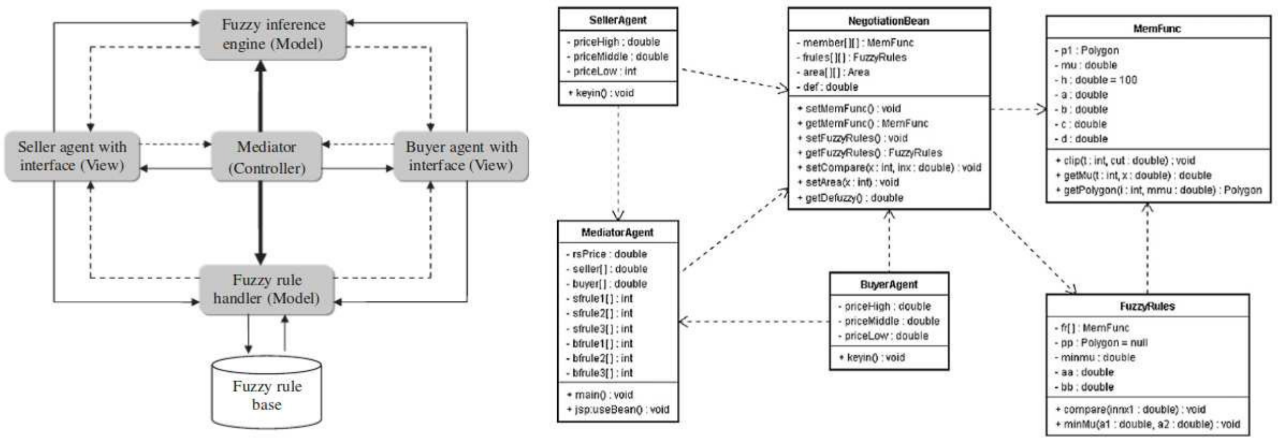


Fig. 1. The MVC design pattern and the price negotiation system class diagram [25]

we still have to change the knowledge base), the *View* (in order to be compatible with another ontology serving us as a dictionary of concepts used for production rules) as well as the *Controller* (since the model interface has to be updated in order to be compatible with that new ontology).

2) *Changeability*: during the evolution the price negotiation domain description changes significantly: temporal aspects are added to the price negotiation rules similar to "if a seller decreased the price on a small value followed by decreasing the price on a medium value then ...". If we consider this change, we have to modify the *View* (in order to make it compatible with another set of concepts) as well as the *Model* and the *Controller* (for the similar reasons as those in the reusability scenario description).

Thus, the above scenarios require changes in all the three MVC major components.

Applying methods of formal architecture evaluation allows to evaluate component relations and discover whether a system follows a loosely coupled design strategy [26]. Let us note that using component interfaces doesn't guarantee system loose coupling since a scenario might affect changes in an interface, which, in turn, leads to the changes in all the components depending on this interface.

Both of the above mentioned scenarios require an ontology to be changed (e.g. the dictionaries used in production rules). This is a key factor since other two components (e.g. the *View* and the *Presenter/Controller*) directly depend on the domain ontology.

We think that the problem is that an interface is strongly connected to the subject domain. If we succeed to remove the subject domain related information from the interface, we are able to construct a user friendly GUI without having to redesign an expert system architecture in order to fit every change in the expert system subject domain.

Following [23], there are two major problems to be resolved while developing a GUI application:

- *UI*: How does the user interact with my data?
- *Data Management*: How do I manage my data?

Each problem falls into three more concrete questions (see Figure 2). In our work we only address the following questions: *What is my data?* (see Section III-A), *How do I change my data?* (see Section IV-C1), and partially the question *How do I display my data?* (see Section IV-C2). In MVC terms the questions are: *What is the interface of my model for the view?* and *What is the interface of my model for the controller?*.

III. Introducing the Model

After the analysis of a series of existing expert system implementations, we realized that there are two basic approaches to define a *Model* interface:

1) **Expert system domain aware model interface:** There are interface methods directly connected to the subject domain as it is implemented in work [25] (e.g. *setPriceHigh* in the example of automated price negotiation on the web). Figure 5 (1) illustrates this issue.
2) **Domain agnostic model interface:** A user interface communicates directly with the inference engine interface as it is implemented in works [27], [28]. It means that there are methods like *assertFact(fact: Fact)* (see Figure 5 (2)).

If we follow the first approach, we have to change the *Model* interface in order to respond to subject domain changes. If we rely on the second one, we are able to keep the *Model*–presenter interaction interface, but changes in the subject domain still require changes both in the *Model* (the knowledge base rules) and in the *Presenter* (since the latter should be able to assert new facts to the knowledge base). Thus, both approaches are not aimed at using the MVP pattern in the best way.

In order to separate the *Presenter* and the *Model* we propose to complement an expert system subject domain ontology (e.g. automatic price negotiation or CLI application deployment) with a user communication subject domain ontology (*UserComm*, see Figure 5 (3)). As far as a problem of user

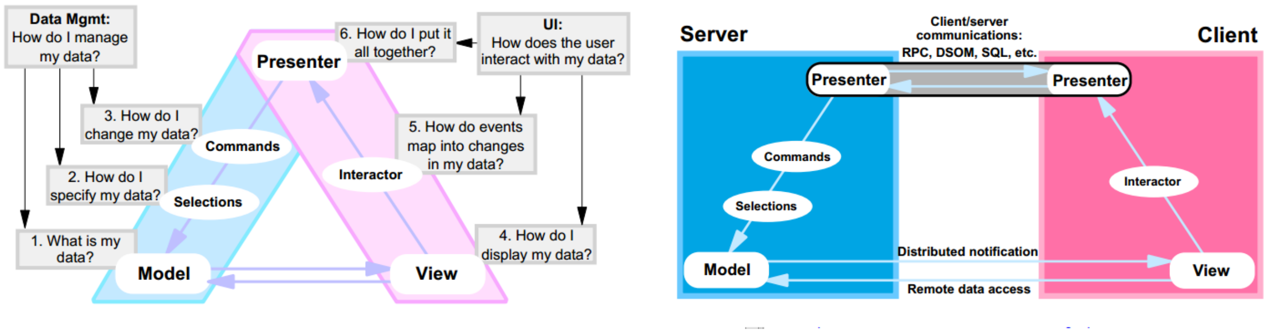


Fig. 2. The MVP design pattern [23]

communication can be described with no connection to the subject domain problems, a user communication model can be developed once and then reused oftentimes as long as its interface is well designed and doesn't change. Therefore, as a *Model*, we propose to use a knowledge base (with rules description and its working memory), but an interface of the model for other components includes only the concepts from the *UserComm* domain.

A. Introducing a Request-Activity and Related Facts Core Ontology

In order to formalize both software provisioning and execution error description, as well as the relationships between an error and an error resolution procedure, knowledge engineering formalisms are required. In [29] and [9] we proposed and argued for a *Software Provisioning Ontology* that describes processes of software code building and execution with much attention paid to represent build and execution errors as well as the actions required to fix the recognized errors. In the earlier mentioned works we also demonstrated how ontologies of specific tasks can be defined by extending the core ontology base entities.

Hereinafter we only introduce basic entities of the above mentioned software provisioning ontology aimed at describing and resolving the problems of CLI software provisioning to a virtual platform. Let us mention again that we focus on deployment problems in relation to the special software class – *research software implementing the MIR algorithms*. The authors of such programs are usually able to implement an algorithm in the form of CLI-based console application, which transforms the input data to the output according to the data formats required by a certain algorithm evaluation system. However, it is common that a developer might not be experienced enough to resolve runtime environment failures or to guarantee that virtual platform requirements are satisfied.

The major concepts of this ontology are *Activities* and activity *Requests* (see Figure 3). An *Activity* is a sequence of *Actions* aimed at achieving an activity goal, while a *Request* can be considered as a new goal setting. An activity might aggregate requests (being subrequests, in a sense), while a request might consist of activities: if an activity fails, an error has to be identified and fixed, then the activity for the same request has to be restarted. If the activity failure can not be fixed, the request is considered failed.

In order to describe activity results, we introduce a concept of an *Activity status*, which is twofold: there may be an *Activity runtime status* and an *Activity completion status*. The *Activity runtime status* instances are an *Activity being executed* and an *Activity suspended*. The *Activity completion status* instances are an *Activity succeeded* and an *Activity failed*. We assume that an activity is completed successfully if the activity goal is reached (for example, for the activity *Unpacking* the artifact has been successfully unpacked). Otherwise the activity is failed (for example, some file artifact has not been unpacked for the reason that the required archiving utility has not been found in the system).

The *Request* features a necessary and appropriate condition that there starts an activity of a particular type. Similar to an activity concept, a *Request* might also have its status, which is also twofold: there are a *Request runtime status* and a *Request completion status*. The *Request runtime status* instances are a *Request being executed* and a *Request suspended*, while the *Request completion status* instances are a *Request succeeded* and a *Request failed*. If at least one activity for the request is completed successfully, the request is considered to be completed successfully too. By contrast, if all the activities associated with the given request failed, the request is considered to be failed.

Subject domain ontologies are rarely used in expert systems directly: they are usually too common to describe the subject domain-related specific tasks. However, we are able to define an ontology of specific tasks by extending the base entities of the core ontology, and in so doing to follow an *extendibility* principle of the ontology design: "an ontology should be designed so as to allow to use shared vocabularies and to support monotonic ontology extension or/and specialization (i.e. new terms might be introduced without revising existing definitions)" [30].

Let us note that in the earlier mentioned work [9] we also demonstrated how to construct the knowledge base production rules in order to manage processes of client application building and execution with detecting respective errors while using some building tool (e.g. *maven*) as a kind of specific building system.

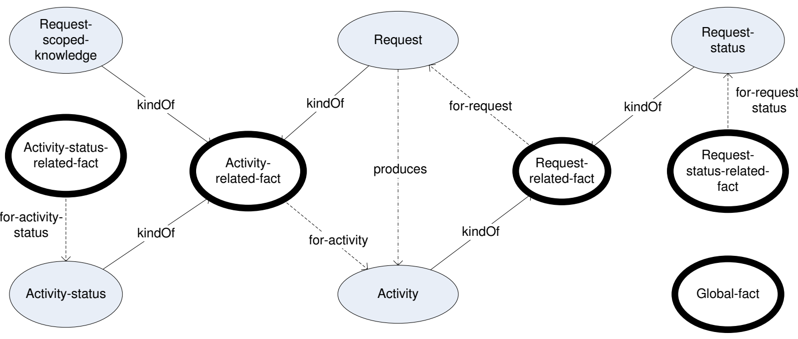


Fig. 3. Activities and requests are main ontology concepts

B. User Communication Ontology

The user communication ontology (that we refer to as *UserComm*) is based on *Request-RequestStatus* and *Related-Facts* concepts of the core ontology mentioned in the previous section.

The *UserComm* ontology provides the following concepts (see Figure 4) which represent users' requests:

- A **user request** extends a *Request* concept and represents a request generated by a user. Associated *request related facts* provide more details on the user request
- An **ordered argument** extends a *request related fact* and represents a part of the user request in the form of an *integer-value*, where a value is an arbitrary string.
- A **key-value argument** extends a *request related fact* concept and represents a part of the user request in the form of *key-value* where *key* and *value* are strings.
- A **key-(multivalue argument)** extends a *request related fact* concept and represents a part of the user request in the form of *key-(array of values)*, where *key* and each *value* are strings.
- A **named artifact** extends a *request related fact* concept and represents a part of the user request in the form of a *name-(binary file)*, where *name* is a string
- An **unnamed artifact** extends a *request related fact* concept and represents a part of the user request in the form of *binary file*. At most one unnamed artifact may be associated with a request.

The *UserComm* domain description serves as an abstraction layer used by both the *Presenter* and the *Model* allowing to hide real expert system's domain from the *Presenter* as shown in Figure 5 (3). UML diagram presenting the *Model* interface to be used by a *Presenter*, a *Controller* or a *View* is shown in Figure 6. In the following sections we describe how the *Presenter* and the *View* use the *Model* interface, and how the expert system domain rules communicate with a user via the *UserComm* ontology abstraction layer.

IV. Software Provisioning Self-Service Networked Infrastructure: An Architecture and Major Components

An architecture of a system for automated experiments with algorithms developed in MIR is shown in Figure 7 (see also [6]). It includes the following components:

- User interface
- Submitted applications repository
- Virtual machine images repository
- Deployment knowledge base
- Deployment manager
- Input provider service
- Result collecting service
- Statistics service
- Authentication and authorization service
- Client virtual machines
- Cloud manager
- Cloud administrator console

Some of the listed components (e.g. a cloud administrator console, authentication and authorization services, etc.) are provided by a cloud infrastructure.

Provisioning client applications to a cloud is supported by two major components of a virtual platform: a *cloud broker* and a *deployment manager* (see Figure 8). The latter is a composition of a *deployment manager agent* and a *configuration manager*. A *knowledge base* (KB), an *inference engine* and its *working memory* are components of an expert system controlling the provisioning process.

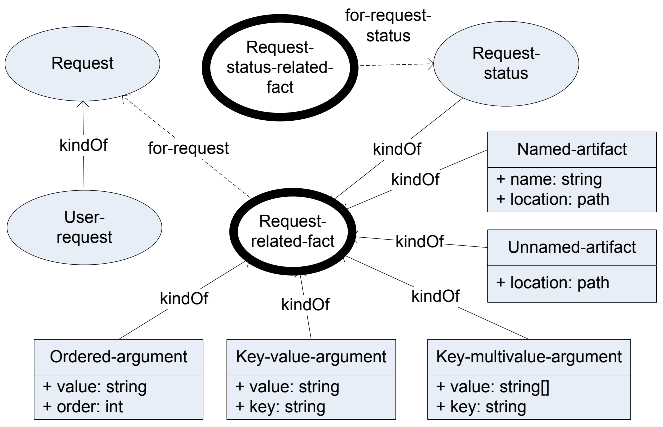


Fig. 4. *UserComm* domain ontology

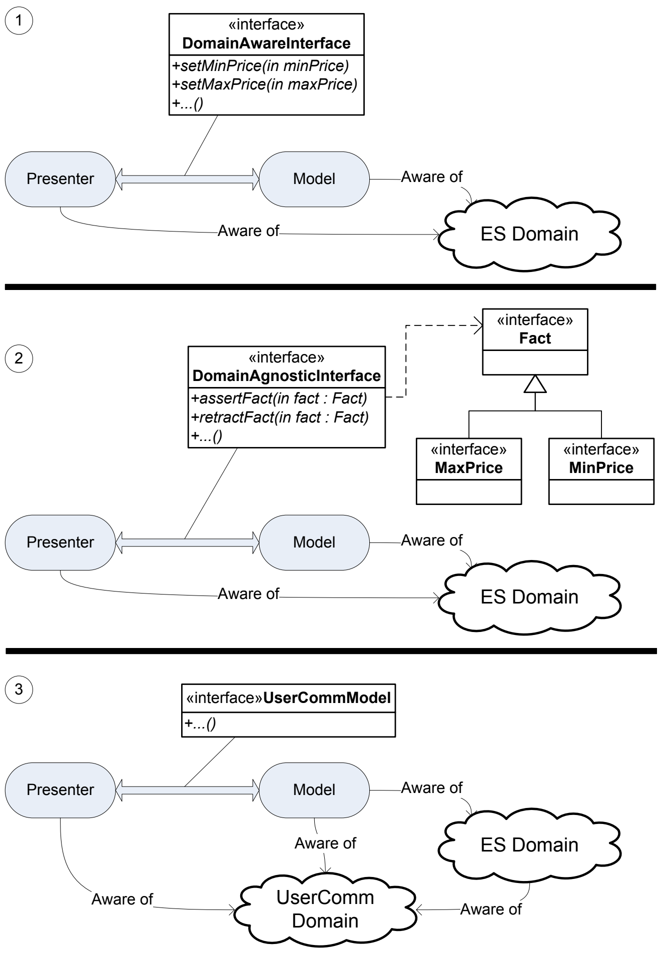
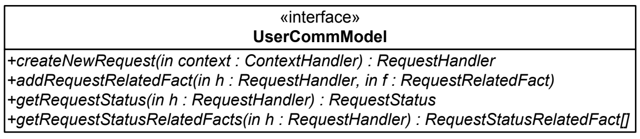


Fig. 5. Model interface for a presenter: (1) expert system domain aware; (2) expert system domain agnostic; (3) user communication domain aware


«interface»
UserCommModel

+createNewRequest(in context : ContextHandler) : RequestHandler
+addRequestRelatedFact(in h : RequestHandler, in f : RequestRelatedFact)
+getRequestStatus(in h : RequestHandler) : RequestStatus
+getRequestStatusRelatedFacts(in h : RequestHandler) : RequestStatusRelatedFact[]

Fig. 6. Model interface

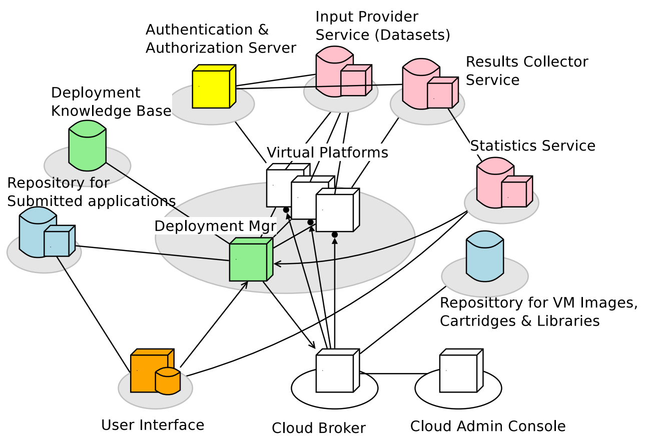

Authentication & Authorization Server
Input Provider Service (Datasets)
Results Collector Service
Deployment Knowledge Base
Virtual Platforms
Statistics Service
Repository for Submitted applications
Deployment Mgr
Repository for VM Images, Cartridges & Libraries
User Interface
Cloud Broker
Cloud Admin Console

Fig. 7. CLI software provisioning service architecture

A. *Deployment Manager*

Similar to the *MEDEA* approach, a wrapper is uploaded to a virtual machine deployed in a cloud. The wrapper provides an HTTP interface and executes a client CLI application in response to user inputs provided via an HTTP proxy interface. Normally the wrapper consists of two components: a *proxy* and an *executor*, where the proxy invokes the executor directly according to HTTP commands received via an HTTP interface. In our implementation, in contrast to a traditional approach, the proxy and the executor never interact directly but via indirect communications using a knowledge base. In terms of the MVP design pattern the *Proxy* is a *Presenter*, the *Knowledge Base* is a *Model*. while the *View* could be either a client side web browser, or a server side HTML code generator component.

Let us note that an expert system often communicates not only with a user but with other components of the system. For instance, for the purpose of CLI application deployment an expert system might need executing a command (a communication with the executor) or changing a platform configuration (a communication with the configuration manager). The problem is how to define an interface that doesn't need to be changed if an expert system domain changes. This problem is similar to the problem of interface definition between a *View* and a *Model*, as well as between a *Presenter* and a *Model* in MVP pattern. Hence, we can use the same approach. We can define an ontology (*ActionExecution* ontology or *ConfigManagement* ontology) used for communication between an expert system and any external component.

B. *Deployment Manager Agent*

The deployment manager agent gathers runtime information about the client application and about the environment state and uses the knowledge base in order to resolve deployment and execution errors such as absence of required components or libraries, improper runtime environment version, etc. The agent interacts with the configuration manager by using the *ConfigManagement* ontology (including high-level commands like "need Python3") in order to reconfigure the platform properly. In turn, the configuration manager interacts with the cloud broker (which is a component provided by a cloud itself) by using low-level commands (e.g. "change VM image"). In so doing, the configuration manager controls the installation of the external components (such as language runtimes, necessary middleware or databases) to the platform. It also controls virtual machines recreation if required.

C. *Proxy*

The proxy component is responsible for providing a capability to access the expert system by supporting two routines:

1) Asserting user requests to the knowledge base;
2) Retrieving the execution status.

In a sense, the proxy acts as an adapter transforming the data representation from one form (HTTP) to another (*UserComm* facts) and vice versa. The *UserCommModel* interface (see Figure 6) is used for interaction with the knowledge base. Request related facts for a request generated by the proxy

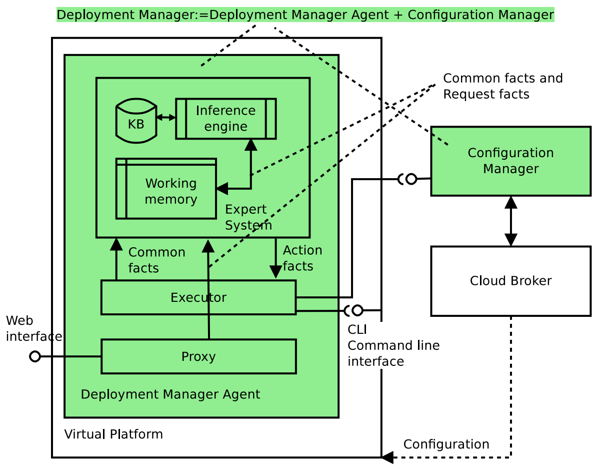


Fig. 8. Managing application deployment in a cloud

should be only concepts from the set of concepts defined by the *UserComm* ontology. In order to interact with a user the REST-like HTTP interface is provided (as defined in Table I).

Two major procedures for HTTP requests processing are described in the following subsections.

1) Proxy: Processing POST Requests: In this section we describe a method for an arbitrary HTTP request transformation to a knowledge base facts representation with use of a fixed set of *UserComm* ontology concepts. The method consists of five major steps:

1) Parse an HTTP request (according to RFC 2616).
2) Map parts of the HTTP request to ontology facts according to the rules defined in Table II.
3) Obtain a $ContextHandler$[6] object from the HTTP request URL.
4) Obtain a $RequestHandler$ object by invoking $createNewRequest$ method (see Figure 6) with a $ContextHandler$ parameter obtained in the previous step. In fact, this invocation asserts new $UserRequest$ object to the knowledge base of the model.
5) Assert all the $RequestRelatedFacts$ by invoking $addRequestRelatedFact$ method with a $RequestHandler$ parameter obtained in the previous step.

2) Proxy: Processing GET Requests: In this section we describe a method for querying a $RequestResult$ with an HTTP request. The method consists of six major steps:

1) Parse an HTTP request according to RFC 2616.
2) Extract $RequestHandler$ object from the request URL.
3) Get a $RequestStatus$ for a $RequestHandler$ by invoking $getRequestStatus$ method (see Figure 6) with a $RequestHandler$ parameter obtained in the previous step.
4) Get a $RequestStatusRelatedFacts$ for a $RequestHandler$ by invoking $getRequestStatusRelatedFacts$ method (see Figure 6).
5) Decide on view layout on the base of the $RequestStatus$ properties (including runtime type information) and a variety of $RequestStatusRelatedFacts$ (or use some default layout).
6) Draw each object with its own widget being a part of the layout.

[6]According to Section III-A a context for a *Request* is an *Activity*. By convention a top-level activity may represent a user and a context for all top-level requests. For a subrequest its context is represented by the request's parent *Activity*

D. Execution Process

As we described in our earlier work [6], we extended the *MEDEA* and *NEMA* execution model by adding the second phase of the execution process. During the first phase (command execution) some debugging information can be written to *stdout*/*stderr*, to the environment logs and so on. These execution results are represented as facts and asserted into the working memory. During the second phase the results are analyzed. As soon as the expert system determines the execution failure and the error cause is determined, the expert system issues appropriate reconfiguration command. This command is handled by the executor, which either performs the necessary operations by itself or delegates them to the configuration manager.

An *executor* is a platform-specific component. It observes a working memory for the presence of action facts. We use a deferred action execution model: as soon as the inference engine does not have any active rules, the executor performs the required actions described in the form of action facts stored in the working memory. Such an approach is tolerant to action facts addition, deletion or modification up until the moment when the agent performs the action.

E. Configuration Manager

The configuration manager interacts with the cloud broker in order to provide the required configuration, i.e. to install/uninstall necessary/unnecessary system components, (frameworks, applications). As soon as the reconfiguration process is completed, the configuration manager asserts new environment configuration facts. As a result, this assertion might activate the rules asserting actions in order to execute the failed command again or to notify the user about an unrecoverable failure detected.

V. Evaluation

First, let us demonstrate how to apply the proposed system architecture to develop a web-based expert system for CLI applications deployment in a cloud. We consider only user communication aspect of the system in this example.

The *Proxy* component provides an API as described in Table I. We extended the proxy interface with a GET method for all URLs supporting POST requests. The response to the GET request to such a URL returns a simple HTML page that a user can use to upload an artifact (zip archive, for example)

TABLE I
REST INTERFACE OF THE PROXY COMPONENT

URL	HTTP method	Description
/action/<contextHandler>/<relative path>?<query string>	POST	Submit a user request within the context $contextHandler$. The request is executed according to the procedure described in section IV-C1.
/status/<requestHandler>	GET	Get execution result for a request identified by $requestHandler$. The request is executed according to the procedure described in section IV-C2.

TABLE II
HTTP REQUEST TO *UserComm* CONCEPTS TRANSFORMATION RULES

HTTP message part	Ontology concept	Description
Segment(*) of <relative path>	OrderedArgument(segNo, segValue)	The segment is represented by its value *segValue* (path segment name as it appears in the URL) and its order *segNo*(i.e. number of '/' signs in the URL before the segment)
Parameter(**) from <query string> ('key=value')	KeyValueArgument(key, value)	The query string parameter is represented by its *key* and *value*.
Parameter from <query string> ('key' or 'key=values[]')	KeyMultivalueArgument(key, value[])	The query string parameter is represented by its *key* and *value[]* (zero or several, but not exactly one value).
HTML form input field (input field type is not 'file')	KeyValueArgument(fieldId, value)	The form input field is represented by its id *fieldId* as it appears in HTML code or in multipart HTTP message and *value* (the contents of the field or the entity value in multipart HTTP message)
HTML form input field (input field type is 'file')	NamedArtifact(fieldId, fileContentsPath)	The file input field is represented by its id *fieldId* as it appears in HTML code or in multipart HTTP message and a path *fileContentsPath* to a local copy of the binary file received from a user
HTTP named entity (not HTML form)	NamedArtifact(entityName, fileContentsPath)	Named entity is represented by its id *entityName* as it appears in multipart HTTP message and a path *fileContentsPath* to a local copy of the binary file received from a user
HTTP default entity	UnamedArtifact(fileContentsPath)	Default entity is represented by a path *fileContentsPath* to a local copy of the binary file received from a user

(*) According to RFC 3986 path of a URL consists of zero or more segments separated by slash ('/') character.
(**) RFC 3986 doesn't set any restrictions on a query string format. In practice Web developers use ampersand ('&') separated 'key=value', 'key=values[]' or 'key' format as defined in RFC 1866.

to the server. Selecting a file to upload followed by clicking on "Submit" button causes the form to be uploaded to the same URL that is used to download the form. In our system the GET URL path is "/action/user/deploy/" so the POST URL path is also "/action/user/deploy/".

According to the algorithm described in Section IV-C1, the proxy parses an HTTP message in the following way: a *contextHandler* is string "user", a relative path consists of one segment "deploy". Hence an input field with an artifact is transformed to $NamedArtifact(``artifact", ``/local/file/path")$ and URL path is transformed to $OrderedArgument(1, ``deploy")$. The proxy invokes $createNewRequest$ method (see Figure 6) of the model and receives a $RequestHandler$ that is used to assert all other facts via $addRequestRelatedFact$ method and returns the $RequestHandler$ to the user's browser. The latter redirects the response to URL "/status/RequestHandler".

In the knowledge base we can construct a rule translating a *UserComm* domain to an expert system domain as follows:

```
RULE 'Deploy an artifact'
IF
  ctx : Context('user')
  req : UserRequest(ctx)
  exists OrderedArgument( order==1
                    AND value=='deploy'
                    AND request==req)
  artifact : NamedArtifact(name=='artifact'
                     AND request==req)
THEN
  assert(Expert system domain facts)
END RULE
```

Similarly, we are able to define the rules to translate the expert system domain back to the *UserComm* domain.

Redirection to "/status/RequestHandler" URL enables users to monitor execution process. In the simplest case the request result might be described as a status string, e.g. "in progress", "completed" and so on.

A. *Expert System Evaluation: Preview*

Currently we tested the approach by implementing a prototype system that was successfully used for automatic deployment of two *Java* projects (built with *maven*) selected among the projects submitted to the *MIREX 2013* contest[7]. During the deployment there were several configuration errors. For each error we provided the knowledge base rule in order to detect and fix configuration errors. Table III lists the examples of build and run errors discovered during our experiments.

B. *Scenario Based System Architecture Evaluation*

1) Reuse Scenario: Domain That Changes: Let's revisit the case study investigated in Section II-C: the change of the *Model* provoked changes in both the *View* and the *Controller*. As we discovered in Section II-C, these changes are conditioned by the fact that the *Model* provides a domain-specific interface for data modification. If we use the described architecture (which includes two domains: the communication domain (*UserComm*) and the expert system domain), changing the expert system domain doesn't lead us to the communication domain changes. Thus, the *Model* interface can be preserved, and the *Presenter* remains unchanged. The *View* might need to be updated in order to support new *RequestStatusRelatedFacts* introduced by the changed expert system domain. The communication interface between a *View* and a *Model* (as well as between a *View* and a *Presenter*) remains unchanged. To sum up, updating the *Model* component leads to changes in the *View* only because this new domain has new concepts to be visualized.

2) Change Scenario: Model That Changes: The *Model* change can be handled the same way as the domain change (see section V-B1). Updating the *Model* might lead to changes in the *View* if this new model has new concepts to be visualized.

3) Change Scenario: Managed and Unmanaged Execution Modes: As we described in [6], the architecture with isolated proxy and executor components allows implementing different execution modes (e.g. system behavior) without modifications of neither the proxy, nor the executor. In practical cases, at least two execution modes are useful: *managed mode* and *unmanaged mode*. Regardless of the current mode, the knowledge base is able to detect and fix recoverable errors in order to support client software automatic deployment.

In the *managed mode* we have a predefined client code invocation command, as well as we use strict validation of the input and output data. Validated output data are automatically published as request status related facts. Especially in case of MIR, in the managed mode it is possible to keep private music collections safe while providing access to these collections for processing by third party algorithms.

In the *unmanaged mode* the invocation command, as well as its input and output data are defined in an HTTP request. The framework doesn't check input and output data, but it still detects and fixes recoverable execution errors.

[7]http://www.music-ir.org/mirex/wiki/2013:Main_Page

While these modes differ significantly in behavior, they are implemented entirely at the knowledge base level. It means that in order to support managed or unmanaged modes, only changes in the knowledge base component (i.e. in the *Model*) are required.

4) Change Scenario: Deploying a CLI application in IaaS or PaaS clouds: The basic idea of this evaluation scenario is to check whether the expert system is able to deploy a CLI application to *PaaS* and *IaaS* clouds. The platform configuration can be handled in three different ways:

1) In *PaaS* clouds the configuration manager can delegate all operations to the cloud broker (as in *OpenShift*). Necessary components can be installed as cartridges developed by the communities (e.g. *Python* or *Ruby* cartridges). This approach is the easiest to implement, but it requires support from the cloud as Figure 9 (left) illustrates.
2) In *PaaS* and *IaaS* clouds the configuration manager is able to install all the required software itself. Software installation can be implemented using the same interfaces that end users have. Indeed, within the context of deployment there is no big difference whether the deployment manager deploys a particular version of a build system (e.g. *maven*), or a MIR research application. This approach is illustrated in Figure 9 (middle).
3) In *IaaS* clouds it is possible to have a set of preconfigured virtual machines, hence configuring means switching of virtual machine images as shown in Figure 9 (right). This way might be resource consuming but in some cases there is no other choice. For instance, it is useful to have two images: one with **nix* OS and the other one with *Windows* OS, because there is no way to install *Windows* applications to **nix* or vice versa with no virtual machines usage.

In its pure forms neither *PaaS* nor *IaaS* fits the task of MIR research software automatic deployment and execution. For example, using the only *PaaS* it is often impossible to implement access to big local data. If we consider an example of *MSD* (Million Songs Dataset [31]) with its size of about 240 GB, we immediately face two problems: 1) a virtual platform is usually limited by only several GBs of disk space, 2) for a virtual platform it is usually not allowed to mount new partitions, volumes or remote file systems. Our subject domain restrictions make it almost impossible to force client software developers to use some network file system similar to *webdav*.

In turn, an *IaaS* does support mounting new partitions. However, installing and configuring, say, a *Python* environment requires installing the *Python* interpreter from the repositories and its additional configuration by using scripts, i.e. the process not trivial for non-experts. On the contrary, in a *PaaS* the same effect may be achieved by only one command for installing the respective cartridge (of course, if the required cartridge exists, and the latter observation is true for a great majority of practical cases and *PaaS* platforms). Therefore we propose to use a hybrid approach *IaaS+PaaS*, where a *PaaS* may be deployed within an *IaaS* cloud.

TABLE III
EXAMPLES OF ERRORS DISCOVERED DURING EXPERIMENTS WITH MIREX 2013 PROJECTS

Error	Component	Detection method	Recovery action
Incorrect encoding of source file	javac	Pattern matching. Look for "error: unmappable character for encoding" in javac log	Add appropriate javac/maven flag to specify encoding. Encoding can be detected automatically or provided by user
copy-maven-plugin runtime exception	maven	Pattern matching. Look for "copy-maven-plugin:0.2.5:copy" and "java.lang.NoClassDefFoundError: Lorg/sonatype/aether/RepositorySystem;" in maven log	Downgrade maven to version 3.0.5
X11 server required	JVM/AWT	Pattern matching. Look for "java.awt.HeadlessException" in JVM classloader log(*)	Install X11 server (we use Xvfb virtual server)

(*)This exception is especially interesting for two reasons: 1) nobody expected AWT exception in CLI application and 2) This exception is caught in client code, but it is not handled properly (it is ignored). The only way to detect that the exception was thrown is to analyze the classloader log.

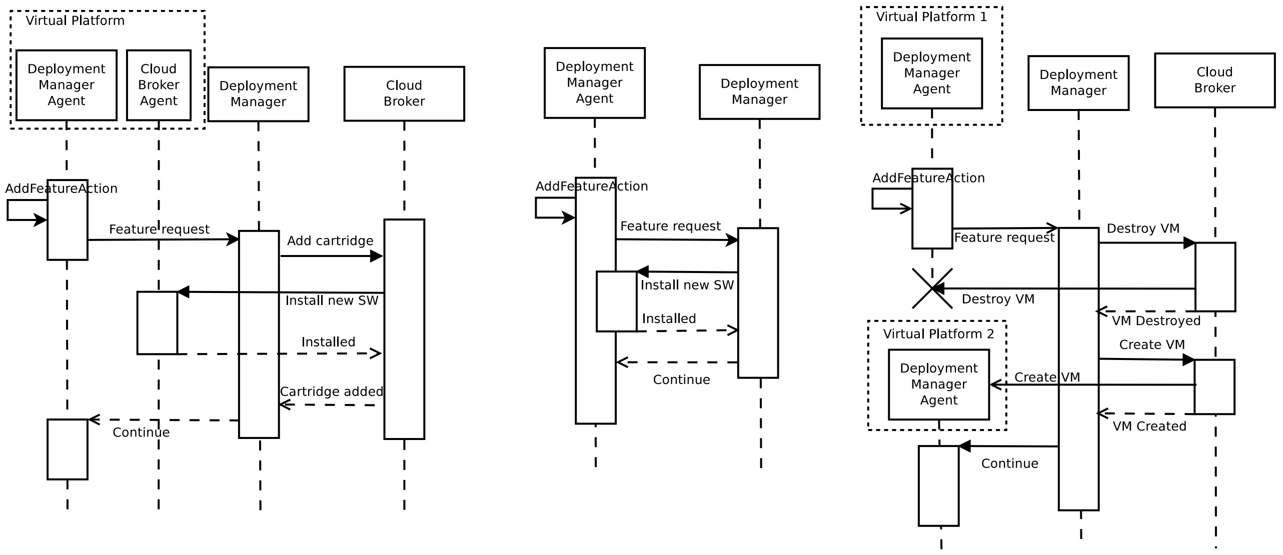


Fig. 9. PaaS/IaaS reconfiguration: (Left) New cartridge installation; (Middle) New software installation; (Right) Virtual machine recreation.

VI. CONCLUSION

In this paper we have examined the problem of CLI application provisioning in clouds. Being a web application interacting with a cloud broker in order to manage cloud resources and their configuration, the proposed infrastructure provides a *PaaS* platform to deploy CLI applications as *SaaS* services. In contrast to existing solutions we have proposed an architecture where a proxy component and an executor don't interact directly. Instead of invoking each other, they assert or retract facts in/from an expert system working memory. Thus, the approach draws on the knowledge engineering formalisms used not only for configuration errors recovery, but also for decision making on how to handle requests in order to protect intellectual property (in regards to MIR one can talk about music collections, software implementations, etc.). Although in this research we experimented mostly with automatic deployment of *Java+maven* applications, we are working on knowledge base extensions that allow deploying native *Windows*, *Python*, *MatLab* and *Vamp*[8] applications.

The proposed architecture addresses some of the most complex tasks of client virtual machine automatic reconfiguration, including the following:

1) Installing operating systems on a virtual machine;
2) Installing and configuring a build environment;
3) Installing and configuring a required version of a runtime environment;
4) Installing third party libraries during building;
5) Installing third party runtime libraries;
6) Providing access to machine learning and test data;
7) Providing access to a storage for execution results;
8) Providing user-side access to a client virtual machine.

We have investigated MVC and MVP design patterns as well as the major difficulties of their application to implementing a user interface for a web-based expert system. By introducing a special ontology representing user communication concepts, we have attempted to achieve an MVP-based implementation of an expert system with respect to the loose coupling design requirements, which, in turn, are strongly connected to improving such software quality properties as reusability and changeability. With regards to the demands of scientific communities, we believe that the introduced approach is in good direction to reproducibility, which is

[8]http://www.vamp-plugins.org/

"not an afterthought – it is something that must be designed into a project" [32]. Let us conclude with sharing the idea that research reproducibility might add an overhead (that we attempted to avoid in our approach). However, even "some reproducible practices are better than none – it does not have to be perfect to be a huge improvement" [2].

ACKNOWLEDGMENT

We would like to express our great appreciation to Nina Popova and Michael Tramontano for the very valuable suggestions. The authors would like to thank Prof. Franck Leprevost for the invitation to present this research within the framework of the 2014 Eastern Europe–Luxembourg Workshop on Cloud Computing, Communications, Security and Services in conjunction with the International Conference on Cloud Networking. An opportunity to have a preliminary discussion of this work has significantly improved our current contribution.

REFERENCES

[1] M. D. Plumbley, C. Cannam, and S. Dixon, "Tutorial on reusable software and reproducibility in music informatics research to be presented in to be presented at the 13th ismir conference," Centre for Digital Music, Queen Mary, University of London, 2012.

[2] S. Sufi, N. C. Hong, S. Hettrick, M. Antonioletti, S. Crouch, A. Hay, D. Inupakutika, M. Jackson, A. Pawlik, G. Peru, J. Robinson, L. Carr, D. De Roure, C. Goble, and M. Parsons, "Software in reproducible research: Advice and best practice collected from experiences at the collaborations workshop," in *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, ser. TRUST '14. New York, NY, USA: ACM, 2014, pp. 2:1–2:4. [Online]. Available: http://doi.acm.org/10.1145/2618137.2618140

[3] Y. Janin, C. Vincent, and R. Duraffort, "Care, the comprehensive archiver for reproducible execution," in *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, ser. TRUST '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:7. [Online]. Available: http://doi.acm.org/10.1145/2618137.2618138

[4] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "Openml: Networked science in machine learning," *SIGKDD Explorations*, vol. 15, no. 2, pp. 49–60, 2013. [Online]. Available: http://doi.acm.org/10.1145/2641190.2641198

[5] D. Milne and I. H. Witten, "An open-source toolkit for mining wikipedia," *Artif. Intell.*, vol. 194, pp. 222–239, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2012.06.007

[6] E. Pyshkin and A. Kuznetsov, "A provisioning service for automatic command line applications deployment in computing clouds," in *2014 IEEE Intl Conf on High Performance Computing and Communications (HPCC)*, Aug 2014, pp. 518–521.

[7] C. Bunch, "Automated configuration and deployment of applications in heterogeneous cloud environments," Ph.D. dissertation, Santa Barbara, CA, USA, 2012, aAI3553710.

[8] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: Improving configuration management with operating system causality analysis," in *In Proceedings of the 21st ACM Symposium on Operating Systems Principles (Stevenson*, 2007, pp. 237–250.

[9] E. Pyshkin, A. Kuznetsov, and V. Klyuev, "Understanding software provisioning: An ontological view," in *Databases in Networked Information Systems*, ser. Lecture Notes in Computer Science, W. Chu, S. Kikuchi, and S. Bhalla, Eds. Springer International Publishing, 2015, vol. 8999, pp. 84–111. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16313-0_7

[10] K. West, A. Kumar, A. Shirk, G. Zhu, J. Downie, A. Ehmann, and M. Bay, "The networked environment for music analysis (nema)," in *Services (SERVICES-1), 2010 6th World Congress on*, July 2010, pp. 314–317.

[11] J. C. Giarratano and G. Riley, *Expert systems: principles and programming*. Brooks/Cole Publishing Co., 1989.

[12] Y. Duan, J. S. Edwards, and M. Xu, "Web-based expert systems: benefits and challenges," *Information & Management*, vol. 42, no. 6, pp. 799–811, 2005.

[13] N. Dunstan, "An interactive webbased expert system degree planner," in *The Second International Conference on Informatics Engineering & Information Science (ICIEIS2013)*. The Society of Digital Information and Wireless Communication, 2013, pp. 302–308.

[14] I. M. Dokas, "Developing web sites for web based expert systems: A web engineering approach." in *ITEE*, 2005, pp. 202–217.

[15] N. Dunstan, "A hybrid architecture for web-based expert systems," *International Journal of Artificial Intelligence and Expert Systems*, vol. 3, no. 4, pp. 70–79, 2012.

[16] R. A. Harrington, S. Banks, and E. Santos Jr, "Development of an intelligent user interface for a generic expert system," in *Online Proceedings of the Seventh Midwest Artificial Intelligence and Cognitive Science Conference*, 1996.

[17] M. Nofal and K. M. Fouad, "Developing web-based semantic expert systems," *IJCSI International Journal of Computer Science Issues*, vol. 11, no. 1, pp. 103–110, Jan. 2014.

[18] B. P. Knijnenburg, M. C. Willemsen, Z. Gantner, H. Soncu, and C. Newell, "Explaining the user experience of recommender systems," *User Modeling and User-Adapted Interaction*, vol. 22, no. 4-5, pp. 441–504, 2012.

[19] I. M. Dokas, "Developing web sites for web based expert systems: A web engineering approach," in *In Proceedings of the Second International ICSC Symposium on Information Technologies in Environmental Engineering (Magdeburg*. Shaker Verlag, 2005, pp. 202–217.

[20] R. Morales-Chaparro, M. Linaje, J. Preciado, and F. Sánchez-Figueroa, "Mvc web design patterns and rich internet applications," *Proceedings of the Jornadas de Ingenieria del Software y Bases de Datos*, 2007.

[21] P. Gupta and M. C. Govil, "Mvc design pattern for the multi framework distributed applications using xml, spring and struts framework," *Int J Comput Sci Eng*, vol. 2, no. 4, pp. 1047–1051, 2010.

[22] G. E. Krasner, S. T. Pope *et al.*, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[23] M. Potel, "Mvp: Model-view-presenter the taligent programming model for c++ and java," *Taligent Inc*, 1996.

[24] R. Kazman, L. Bass, M. Webb, and G. Abowd, "Saam: A method for analyzing the properties of software architectures," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 81–90. [Online]. Available: http://dl.acm.org/citation.cfm?id=257734.257746

[25] C.-C. Lin, S.-C. Chen, and Y.-M. Chu, "Automatic price negotiation on the web: An agent-based web application using fuzzy expert system," *Expert Systems with Applications*, vol. 38, no. 5, pp. 5090–5100, 2011.

[26] B. Roy and T. N. Graham, "Methods for evaluating software architecture: A survey," *School of Computing TR*, vol. 545, p. 82, 2008.

[27] A. Kipkebut, "An evaluation of web based expert system as a catalyst for maize production in kenya," *Computer Engineering and Intelligent Systems*, vol. 5, no. 3, pp. 86–97, 2014.

[28] K. Tutuncu and M. Koklu, "A new expert system shell in turkish language for training," in *Proceedings of the International Conference on challenges in IT, Engineering and Technology*, ser. ICCIET'2014, 2014, pp. 26–30.

[29] A. Kuznetsov and E. Pyshkin, "An ontology of software building, execution and environment configuration and its application for software deployment in computing clouds," *St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems*, no. 2(193), pp. 110–125, 2014.

[30] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing," *Int. J. Hum.-Comput. Stud.*, vol. 43, no. 5-6, pp. 907–928, Dec. 1995. [Online]. Available: http://dx.doi.org/10.1006/ijhc.1995.1081

[31] B. McFee, T. Bertin-Mahieux, D. P. W. Ellis, and G. R. G. Lanckriet, "The million song dataset challenge." in *WWW (Companion Volume)*, A. Mille, F. L. Gandon, J. Misselis, M. Rabinovich, and S. Staab, Eds. ACM, 2012, pp. 909–916. [Online]. Available: http://dblp.uni-trier.de/db/conf/www/www2012c.html#McFeeBEL12

[32] D. L. Donoho, "An invitation to reproducible computational research," *Biostatistics*, vol. 11, no. 3, pp. 385–388, 2010.