Understanding Software Provisioning: An Ontological View

Evgeny Pyshkin[1], Andrey Kuznetsov[2], and Vitaly Klyuev[3]

[1] St. Petersburg State Polytechnic University
Institute of Computing and Control
21, Polytekhnicheskaya st., St. Petersburg 195251, Russia
pyshkin@icc.spbstu.ru
[2] Motorola Solutions, Inc.
St. Petersburg Software Center
Business Centre T4, 12, Sedova st., St. Petersburg 192019, Russia
andrei.kuznetsov@motorolasolutions.com
[3] University of Aizu
Software Engineering Lab.
Tsuruga, Ikki-Machi, Aizu-Wakamatsu 965-8580, Japan
vkluev@u-aizu.ac.jp

Abstract. In the areas involving data relatedness analysis and big data processing (such as information retrieval and data mining) one of common ways to test developed algorithms is to deal with their software implementations. Deploying software as services is one of possible ways to support better access to research algorithms, test collections and third party components as well as their easier distribution. While provisioning software to computing clouds researchers often face difficulties in process of software deployment. Most research software programs utilize different types of unified interface; among them there are many desktop command-line console applications which are unsuitable for execution in networked or distributed environments. This significantly complicates the process of distributing research software via computing clouds. As a part of knowledge driven approach to provisioning CLI software in clouds we introduce a novel subject domain ontology which is purposed to describe and support processes of software building, configuration and execution. We pay special attention to the process of fixing recoverable build and execution errors automatically. We study how ontologies targeting specific build and runtime environments can be defined by using the software provisioning ontology as a conceptual core. We examine how the proposed ontology can be used in order to define knowledge base rules for an expert system controlling the process of provisioning applications to computing clouds and making them accessible as web services.

Keywords: Knowledge Engineering, Ontology Design, Service-Oriented Architecture, Cloud Computing, Software Deployment Automation.

W. Chu et al. (Eds.): DNIS 2015, LNCS 8999, pp. 84–111, 2015.

1 Introduction

Service-oriented software and cloud computing significantly transformed the ways we think about possibilities to provide access to numerous computational resources. Currently there are many efforts about investigating possibilities to use clouds in organizing research and collaborative work. Particularly, in the domain of music information retrieval (MIR) researchers often develop software programs implementing different algorithms. Many of such implementations still remain desktop applications with command-line interface (CLI applications). Originally they are not intended to be executed in networked or distributed environments. With respect to issues of facilitating access to such implementations we have to mention the following problems:

1. A client local machine (where an application is assumed to be executed) might not well suit long lasting resource consuming computations.
2. A local application is hardly accessible for other researchers that wish to use the algorithm.
3. Client algorithms might require access to huge test collections which are hard to download and to allocate on a desktop storage.
4. Some test collections (particularly, in music information retrieval (MIR)) might not be publicly available due to the copyright restrictions.

Research communities often complain that many software implementations remain unpublished. Hence, it is often difficult to compare results achieved by different researchers even if we have access to the test data they used. There is another challenge: one researcher is often unable to reproduce results of others (even if they are published).

In the domain of MIR the MIREX[1] was organized to improve algorithms distribution and evaluation with using wide range of evaluation tasks [1]. However, since the evaluation process is implemented in the form of an annual competition researchers are unable to experiment with MIREX tests at any time they need. Researchers have very limited access to the solutions and test collections of others.

First, let us take a look at the Vamp system[2] [2]. Basically, the Vamp provides a plugin based framework which makes the published MIR research software accessible by the people outside the developer field. Properly speaking, the Vamp defines an application programming interface forcing developers to unify the application interface which has to conform the framework requirements. Despite the Vamp became a popular mean of MIR algorithms distribution, the MIREX still accepts CLI software implementations. Hence, the problem of CLI software provisioning remains to be of current concern.

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

[2] `http://www.vamp-plugins.org`

Second, we have to mention the NEMA environment[3] [3]. The NEMA creates a distributed networked infrastructure providing access to the MIR software and data sets over the Internet. The NEMA architecture supports publishing client algorithms as services implementing the required web interface. The deployment process is partially automated: it is based on using 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. Image modifications are manual, hence developers are expected to have specific knowledge in order to deploy their solutions.

The MEDEA[4] system [4] enables deploying an arbitrary CLI application on an arbitrary cloud platform. Special attention is paid to such cloud services as *Google App Engine*, *EC2*, *Eucalyptus*, and *Microsoft Azure*. 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 runtime environment (e.g. *Python*, *Java*, etc.) and executes a client application. Unfortunately, execution and configuration errors are not analyzed, there is no any automatic mechanism to handle build, execution and configuration errors conditioned not by the client code bugs, but by the deployment process faults.

What are advantages we expect in provisioning research software in clouds? First, clients are able to get quicker access to remote resources, be it a network storage, a virtual machine or an applied software. Second, server side computing decreases requirements for the client side hardware. Third, a public API can be defined for a deployed cloud application: publishing the application source code is not required, neither distributing software binary code. Fourth, test collections stored within a cloud are not necessary to be downloaded to client machines. In turn, clients require no direct access to test collections: there are less problems conditioned by copyright restrictions. Furthermore, clients pay only for the resources rent for short periods of time. Finally, application deployment and installation issues have to be resolved only once.

Despite there are many research efforts aimed to facilitate using cloud services for provisioning research software, there are still many open questions. Let's make closer inspection of the problem. There are applications developed for clouds and there are tools automating the deployment process. However, CLI applications are commonly not intended to be executed in clouds, hence they require different automation tools. Existing tools usually proceed from the assumption that a *virtual platform* (e.g. a PaaS or an IaaS service)[5] is configured properly. It means that all external dependencies (e.g. libraries, external resources, compiler or library versions, etc.) are resolved, and the only problem is to upload and to run the code in a cloud [4]. As a matter of actual practice,

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

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

[5] We refer both a PaaS (platform as a service) and an IaaS (infrastructure as a service) computing node as a *virtual platform* if the difference between them is not important.

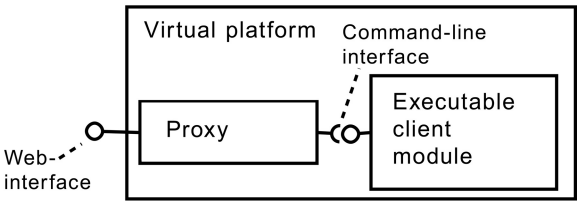

Fig. 1. A web-interface proxy is required in order to deploy CLI applications as web services

this assumption is rarely true, and deploying becomes very complicated: research software developers (being experts in their subject domain) might not be experienced enough in programming and in system administration. They might have no special knowledge on how to transfer their applications to a cloud, and how to configure a cloud runtime environment so as the application works the same way as it works on a local machine.

A reasonable way to deliver a CLI application to a cloud is to define a wrapper (or a proxy) providing a web interface to the application as Figure 1 shows. An automation tool implements mechanisms of transferring a client module (coupled with the related data) to a cloud service. The question is how to discover and to handle software execution errors *automatically*, with special emphasis on the errors conditioned by possible misconfiguration of a virtual platform. In [5] we introduced an idea to use a knowledge driven approach based on an ontology purposed to support processes of software building, its configuration and execution and to describe build and execution errors and actions required to fix recoverable errors automatically. In this work we describe this approach in greater details with a particular focus on ontology design.

2 Introducing the Ontology Usage Domain

As is the case of software testing, there are two major approaches to discover and handle configuration errors and client code external dependencies: source and object modules *static analysis* and *dynamic analysis* (of the execution with using some test data). Static analysis is used in such methods and algorithms as *ConfAnalyzer* [6], *ConfDiagnoser* [7] and *ConfDebugger* [8]. Well-known examples of dynamic analysis tools are *AutoBash* [9] and *ConfAid* [10].

2.1 Sources

In order to formalize error description and the relationships between an error and its resolution procedure, knowledge engineering formalisms are required. Despite there isn't any well-known ontology describing processes of software build, run and runtime environment configuration, we have to cite some general-purpose ontologies used in software engineering.

In [11] the authors describe 19 software engineering ontology types. They give a coat to various aspects of software development, documenting, testing and maintenance. Considering the most relevance to the paper issues, the following ontologies have to be mentioned:

- Software process ontology [12];
- System behavior ontology [13];
- Software artifact ontology [14];
- System configuration ontology [15].

Each of the above listed formalisms operates with a subset of concepts related to the software build, run and environment configuration, but none of these covers the subject in a whole. Beyond subject domain ontologies there are well-known meta-ontologies which potentially fit a wide variety of problems. The process ontologies (like *PSL* [16], *BPEL* [17], *BPMN* [18]) or the general purpose ontology (like *Cyc* [19]) can serve as examples of such formalisms. Unfortunately, meta-ontologies do not contain any concept or relationship which would allow us to describe a specific knowledge area and its specific problems.

In the following sections we define an ontology aimed to describe and resolve problems of CLI software provisioning to a virtual platform. We examine how to construct a production knowledge base used to support client application automatic deployment in clouds. Let us mention again that in this paper we are focused on deployment problems in relation to the special software class – *research software implementing algorithms developed in MIR mostly*. The authors of such programs are usually able to implement an algorithm in the form of CLI console application which transforms input data to the output according to the data formats required by a certain algorithm evaluation system. However it is common that they might not be experienced enough to resolve runtime environment failures or to guarantee that virtual platform requirements are satisfied.

2.2 Target Architecture

It seems impossible and probably useless to try to define an ontology "in the vacuum". In this paper we briefly examine the architectural aspects closely related to the ontology definition. We follow our work [20] where we introduced a target architecture consisting of a virtual platform, a cloud broker and a deployment manager (see Figure 2), the latter being responsible for successful deployment of CLI client console applications on a virtual platform. However that paper doesn't contain any detailed explanation of the software provisioning ontology.

A knowledge base (KB), an inference engine and its working memory are components of an expert system controlling the provisioning process. A pre-installed deployment manager agent provides a web interface to a client application, gathers information about the client application state and environment state. The deployment manager agent uses the knowledge base to resolve deployment problems. The agent interacts with the configuration manager by using high-level commands which describe required actions in order to reconfigure the platform.

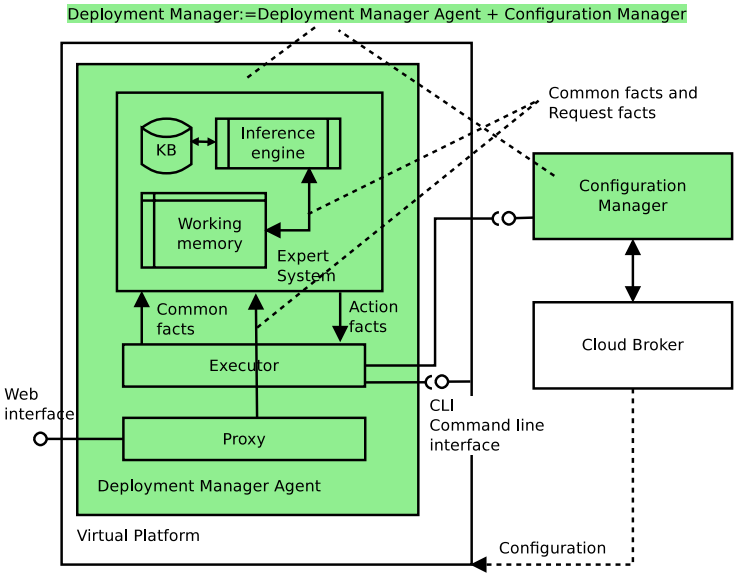

Fig. 2. Managing application deployment in the cloud: an architectural view

In turn, the configuration manager interacts with the cloud broker by using low-level commands. In so doing, it controls the installation of required cartridges[6] as well as the recreation of virtual machines if necessary. Figure 3 represents the general deploying procedure where the expert system is used to resolve possible configuration errors.

There are two major ways to upload a client module on a virtual platform:

- An executable module is uploaded by a client.
- An executable module is generated as a result of a server side building process.

In the latter case the executable module may be considered as a derived artifact of the source code building process. Furthermore, the building process itself can be described by using the same ontology terms. Indeed, the building process is a kind of execution where a compiler (or a building system like *make*, *ant* or *maven*) is an executable entity while a source code is considered to be the input data for a compiler.

3 Constructing a Software Provisioning Ontology

In order to be capable to describe typical scenarios of CLI software automatic deployment we defined a subject domain ontology *"Automated CLI application*

[6] Cartridges encapsulate application components, for example, language runtimes, middleware, and databases, and expose them as services.

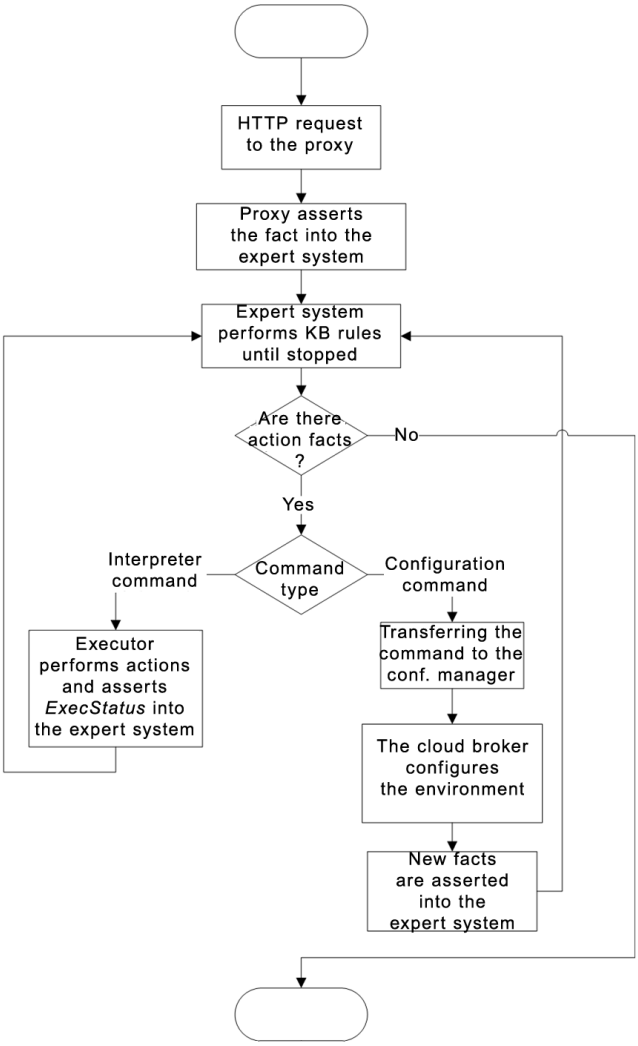

Fig. 3. Main stages of the deployment process

build and run with resolving runtime environment configuration errors" (for further references we use an abridged name *"Software provisioning ontology"*).

3.1 Requirements That Compete: A Case Study

Following Gruber's ontology definition where an ontology is considered as an explicit specification of the conceptualization [21], there are five major ontology design requirements [22]:

1. **Clarity:** An ontology should operate with the intended meanings and with complete (as possible) definitions of introduced terms.
2. **Coherence:** The defined axioms should be logically consistent and all the assertions inferred from the axioms shouldnt contradict to informally given definitions or examples.
3. **Extendibility:** An ontology should be designed so as to allow using shared vocabularies and to support monotonic ontology extension or/and specialization (i.e. new terms might be introduced without revising existing definitions).
4. **Minimal Encoding Bias:** The conceptualization should be specified at the knowledge level without depending on a particular symbol-level encoding.
5. **Minimal Ontological Commitment:** An ontology should require the minimal ontological commitment sufficient to support the intended knowledge sharing activities.

These requirements are competitive: they might contradict to each other. If we consider using ontology term vocabularies in the domain of production knowledge bases, the requirements (1), (3) and (5) are in opposition. If the ontology terms are too common (term commonness being the easiest way to minimize ontological commitment), they are hardly usable to define production rules (since they might produce too much ambiguity). To overcome term ambiguity we have either to revise existing definitions (which contradicts the extendibility principle) or to introduce new terms with complex semantics (which, in turn, contradicts the clarity principle).

Let us consider the following situation: some software module M should be executed triple times (in no particular order) with different arguments $A1$, $A2$, $A3$. Suppose that in order to run the module M with the arguments $A1$ some component K in version V_1 is required (we use the name $K1$ for that). In order to run the module M with the arguments $A2$ the same component K is required, but in version V_2 (the $K2$ component). In order to run the module M with the arguments $A3$ the component K has not to be used at all. The $K1$ and $K2$ configurations are mutually exclusive (i.e the module can not be executed with dependencies on both $K1$ and $K2$). At the beginning (e.g. when the module is being loaded) all the mentioned dependencies are unknown.

Let us introduce facts $NeedK1$ and $NeedK2$ that declare correspondingly that the component configuration $K1$ (for the fact $NeedK1$) or $K2$ (for the fact $NeedK2$) is required in order to successfully execute the module M.

Suppose there are knowledge base rules $R1$ and $R2$ which define how to discover dependencies on $K1$ and $K2$ by analyzing existing installation or execution logs. As a result of executing the production rules $R1$ and $R2$, the facts $NeedK1$ and $NeedK2$ are asserted into the inference engine working memory. Suppose there is also a rule R controlling the module M execution depending on the presence of the facts $NeedK1$ and $NeedK2$ with corresponding dependencies $K1$ and $K2$. Suppose there is a rule R' which is activated when there is no $NeedK1$ nor $NeedK2$ (both facts are absent) in the working memory. Thuswise,

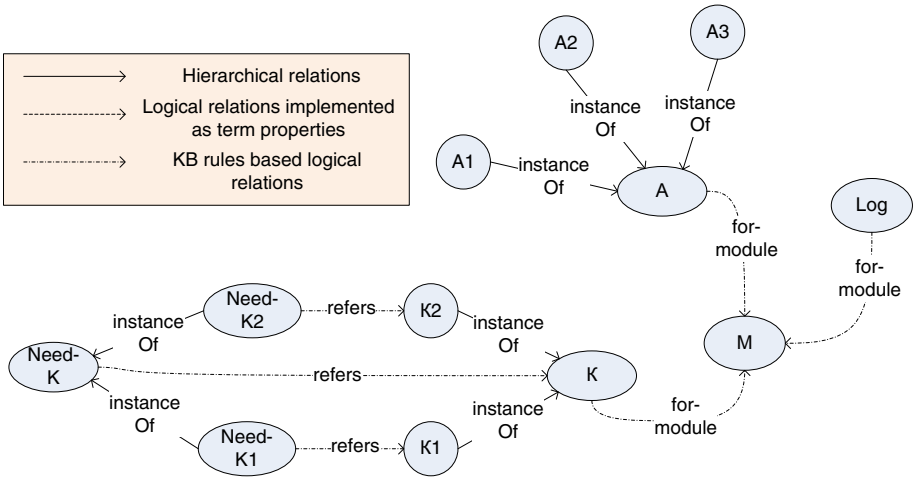

Fig. 4. Running a software module: first attempt to define an ontology

the rule R' controls execution of the module M without dependencies related to the component K.

In fact, such a formal model describes a usual problem of compiling and configuring software projects built from a source code. The different build systems (e.g. *maven* or *ant*) can serve as examples of the module M, target projects can serve as examples of builder arguments A, while third party libraries instantiate dependencies on some component K.

The simple ontology shown in Figure 4 describes the subject domain as follows: there is an A class representing *arguments*. The ontology entities $A1$, $A2$ and $A3$ are instances of A. Class $NeedK$ describes the fact that some component K is required in order to execute the module M. Entities $NeedK1$ and $NeedK2$ are instances of $NeedK$ class containing references to the $K1$ and $K2$ instances of K class. For the reason that the K instances are used with the module M, the K class has an attribute containing a reference to M. The *Log* entity has two attributes: the log *text* and a *reference* to the respective module.

Straightforwardly, Listing 1 demonstrates how to define possible knowledge base rules to be used with this rather naïve ontology.

Listing 1. Knowledge base rules to run a software module (naïve approach)

```
R1:
  if
    Log(text contains ‘‘dependency K1 missed’’)
  then
    assert NeedK1()
end
```

```
R2:
  if
    Log(text contains ‘‘dependency K2 missed’’)
  then
    assert NeedK2()
end

R:
  if
    $dependency: (NeedK1() or NeedK2())
    $arguments: (A1 or A2 or A3)
  then
    run M with $dependency and $arguments
end

R’:
  if
    not NeedK1()
    not NeedK2()
    $arguments: (A1 or A2 or A3)
  then
    run M with $arguments
end
```

It is evident that the ontological model shown in Figure 4 is insufficient to control correct configuration and execution of the module M. There are at least two problems. First, after a sequence of unsuccessful runs (e.g. after an attempt to run the module M with the arguments $A1$ and with no dependencies on K followed by an attempt to run the module M with the arguments $A2$ and with no dependencies on K) the working memory might contain both $NeedK1$ and $NeedK2$ facts. Formally, one can expect that the module shall run with both dependencies ($K1$ and $K2$) which contradicts to the restriction that the components $K1$ and $K2$ are mutually exclusive. Second, the given rules might yield infinite recursion as Table 1 illustrates for one possible series of steps for the subsequently inserted $A1$, $A2$ and $A3$ arguments (in Table 1 the recent items asserted to the working memory being shown bold).

In table 1 rows 7.2 and 7.3 activating the rule R implies executing the module (which has already been successfully run) with beforehand wrong dependencies. Moreover, the step 7 produces recursive rule activation. In the following step 8 the rules $R1$ and $R2$ are inevitably activated again, they assert the facts $NeedK1$ and $NeedK2$, and therefore the rule R is activated four times for every combination of $\{A1, A2\}$ an $\{NeedK1, NeedK2\}$, and then the rules $R1$ and $R2$ are activated all over again.

The latter problem can be fixed by the following improvements of the rules $R1$ and $R2$ (see Listing 2), but this modification doesn't lead to avoiding the unnecessary and irrelevant steps with wrong dependencies.

Table 1. Activating production rules: an issue of infinite recursion

Step	*Rule*	*Facts and Arguments*	*Working Memory*
0	–	–	**A1**
1	R'	A1	A1, **Log("dependency K1 missed")**
2	$R1$	Log	A1, Log("dependency K1 missed"), **NeedK1**
3	R	NeedK1, A1	A1, Log("dependency K1 missed"), NeedK1, **Log("success")**
4	–	–	A1, Log("dependency K1 missed"), NeedK1, Log("success"), **A2**
5	R	NeedK1, A2	Log("dependency K1 missed"), NeedK1, Log("success"), A2, **Log("dependency K2 missed")**
6	$R2$	Log	Log("dependency K1 missed"), NeedK1, Log("success"), A2, Log("dependency K2 missed"), **NeedK2**
7.1	R	NeedK2, A2	Log("dependency K1 missed"), NeedK1, Log("success"), A2, Log("dependency K2 missed"), NeedK2, **Log("success")**
7.2	R	NeedK1, A2	Log("dependency K1 missed"), NeedK1, Log("success"), A2, Log("dependency K2 missed"), NeedK2, Log("success"), **Log("dependency K2 missed")**
7.3	R	NeedK2, A1	Log("dependency K1 missed"), NeedK1, Log("success"), A2, Log("dependency K2 missed"), NeedK2, Log("success"), Log("dependency K2 missed"), **Log("dependency K1 missed")**

Listing 2. Fixing the recursion issue

```
R1:
  if
    Log(text contains ‘‘dependency K1 missed’’)
    not NeedK1()
  then
    assert NeedK1()
end

R2:
  if
    Log(text contains ‘‘dependency K2 missed’’)
    not NeedK2()
  then
    assert NeedK2()
end
```

Furthermore, as Table 2 (steps 9.1 and 9.2) shows, due to the facts $NeedK1$ and $NeedK2$ the rule R' responsible for executing the module M without any external dependency (arguments $A3$) can not be activated. The problem can be solved by adding new rules to the knowledge base in order to allow removing

Table 2. Activating production rules: an issue of irrelevant steps

Step	*Rule*	*Facts and Arguments*	*Working Memory*
8	–	–	Log("dependency K1 missed"), NeedK1, Log("success"), A2, Log("dependency K2 missed"), NeedK2, Log("success"), Log("dependency K2 missed"), Log("dependency K1 missed"), **A3**
9.1	*R*	NeedK1, A3	Log("dependency K1 missed"), NeedK1, Log("success"), A2, Log("dependency K2 missed"), NeedK2, Log("success"), Log("dependency K2 missed"), Log("dependency K1 missed"), A3, **Log("invalid dependency K1")**
9.2	*R*	NeedK2, A3	Log("dependency K1 missed"), NeedK1, Log("success"), A2, Log("dependency K2 missed"), NeedK2, Log("success"), Log("dependency K2 missed"), Log("dependency K1 missed"), A3, Log("invalid dependency K1"), **Log("invalid dependency K2")**

the facts $NeedK1$ and $NeedK2$ from the working memory if necessary. Listing 3 fixes this problem.

Listing 3. Rules for running a module with no external dependencies

```
R11:
  if
    Log(text contains ‘‘invalid dependency K1’’)
    $dependency: NeedK1()
  then
    remove $dependency
end

R12:
  if
    Log(text contains ‘‘invalid dependency K2’’)
    $dependency: NeedK2()
  then
    remove $dependency
end
```

Nonetheless, because of new rules $R11$ and $R12$ appeared, improper rules activation sequences might produce an infinite cycle (consider the sequence $\mathbf{R}'(\mathbf{A1})$, $R1$, $R(A3, NeedK1)$, $R11$, $\mathbf{R}'(\mathbf{A1})$ for instance). Hence, while constructing a rule for inserting facts we have to pay attention to the implementation of other rules. Dependencies between different rules significantly complicate the knowledge base design: rules become unevident and self-contradictory.

The question is how to struggle with this complexity?

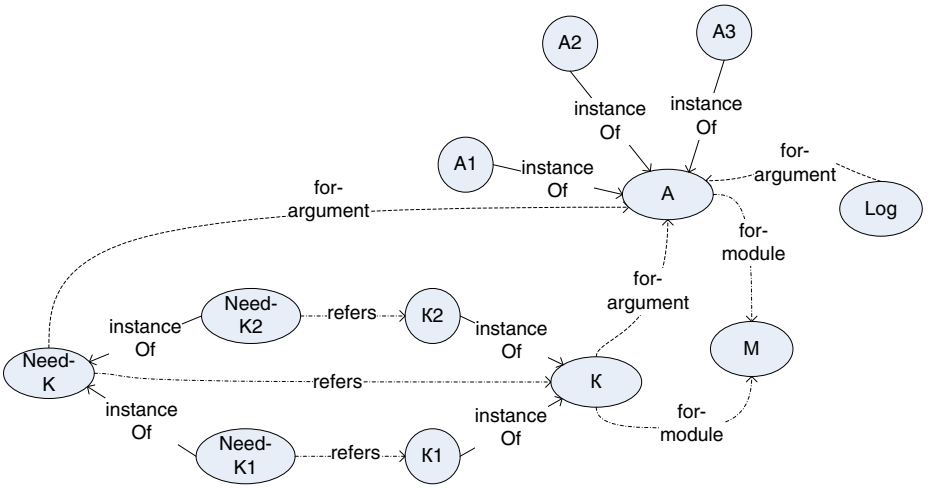

Fig. 5. Running a software module: the revised ontology

Let's turn back to the beginning of the mentioned example and revise the proposed ontology model. We defined the facts $NeedK1$ and $NeedK2$ (and then used them in rules $R1$ and $R2$) as *global scope facts*. It means that they are inserted to the working memory regardless of their connections with the arguments. In reality, the fact say $NeedK1$ has sense only for the concrete arguments. It means that the fact $NeedK1$ should be interpreted not as *"executing the module M with K1 dependency"* but as *"executing the module M with K1 dependency for arguments A1"* or *"executing M with K1 dependency for arguments A2"*, etc. Thus, these facts are **context dependent** and therefore **an ontology should provide concepts supporting the context representation**. It means that we should define an association between the arguments A and the facts $NeedK$ as well as between the *Log* and the class A. The revised sketch is shown in Figure 5.

The rules corresponding to the just mentioned changes are shown in Listing 4.

The class A represents the context: every predicate within the production rule scope refers to the same parameter *$arguments*. Semantics of the classes $NeedK$ and *Log* changed: now there is a relationship linking them to the class A.

Listing 4. Knowledge base rules for the revised ontology

```
R1:
  if
    $arguments: A()
    Log($arguments, text contains ‘‘dependency K1 missed’’)
  then
    assert NeedK1($arguments)
end
```

```
R2:
  if
    $arguments: A()
    Log($arguments, text contains ‘‘dependency K2 missed’’)
  then
    assert NeedK2($arguments)
end

R:
  if
    $arguments: A()
    $dependency: (NeedK1($arguments) or NeedK2($arguments))
  then
    run M with $dependency and $arguments
end

R’:
  if
    $arguments: A()
    not NeedK1($arguments)
    not NeedK2($arguments)
  then
    run M with $arguments
end
```

For the sake of compactness we skip here further analysis of the sample ontology design which allows us to discover the connection between the *Log* class and the component K as well as the rule priority issues.

Nevertheless, even this simplified example implies an important consideration. Paying attention to the competing principles mentioned in the beginning of this section we have to admit that designing subject domain ontology (which is targeted to be used in order to construct ontologies of specific tasks) is a complex problem requiring both solid working experience and much intuition. Indeed, for software engineering purposes one of the most important criteria whether an ontology is successful or not depends on the question whether it is consistent and flexible enough to be used to solve practical problems of software configuration and execution (i.e. only minor extensions of a base ontology are required in order to construct the derived ontologies of specific tasks).

3.2 Core Concepts: Activities, Requests and Related Facts

The software provisioning ontology major concepts are *activities* and activity *requests*. These concepts might aggregate each other: a request might consist of activities, and vice versa, an activity might aggregate requests (being subrequests, in a sense). An *Activity* is a sequence of *actions* provided by an expert or by a user in order to achieve an activity goal, while a *Request* might be considered as a new goal setting.

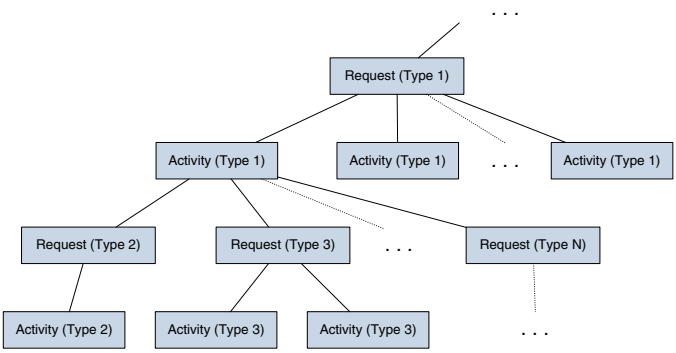

Fig. 6. A hierarchy of activities and requests

Formally, *Activity/Request* hierarchies can be represented by using trees (see Figure 6).

In order to describe activity results, we introduce a concept of an *Activity status* which is twofold: there is an *Activity runtime status* and there is 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*. 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).

The *Request* features a necessary and appropriate condition for starting the activity. For each request instance the activity caused by this request should be known.

Similar to an activity concept, a *Request* might also have its status which is also twofold: there is a *Request runtime status* and there is 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.

Since there might be many objects in the inference engine working memory and these objects might activate many rules at once, assuring rules consistency is a difficult problem. One of the possible ways to solve this problem is to design an ontology in a way that big groups of rules are never activated at once, while all the activated rules are easily computable. In order to support such an approach we introduce the following concepts (see also Figure 7):

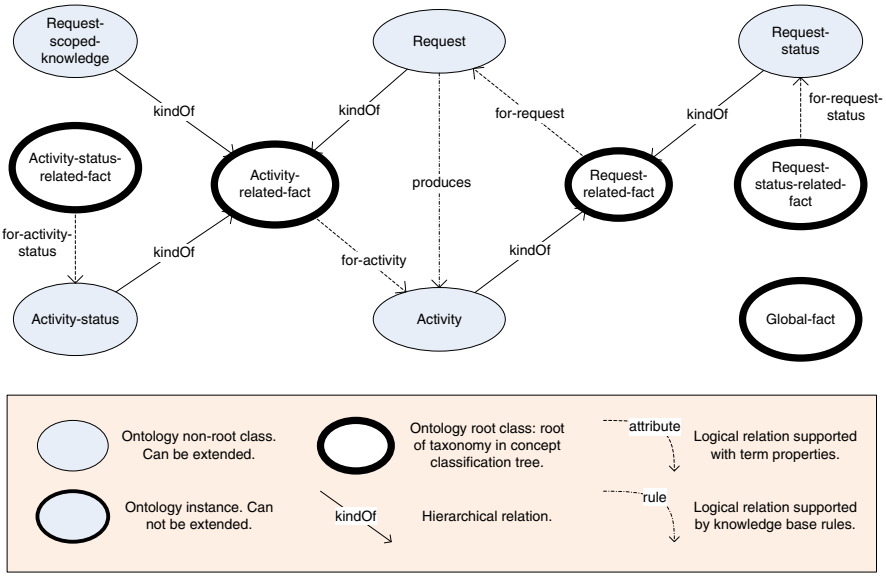

Fig. 7. Activities and requests are major ontology concepts

- An **Activity related fact (ARF)** is a base concept for every fact corresponding to a certain activity. This class has a reference *ARF.activity* to an activity instance. Within the context of executing some activity *Ac* only those facts should be considered which are either global scope facts or the facts related directly to the activity *Ac* (i.e. $ARF.activity = Ac$).
- A **Request related fact (RRF)** is a base concept for every fact corresponding to a certain request. This class has a reference *RRF.request* to a request instance. Within the context of executing some request *Rq* only those facts should be considered which are either global scope facts or the facts related directly to the request *Rq* (i.e. $RRF.request = Rq$).
- An **Activity status related fact(ASRF)** is similar to an *ARF* but related to the activity status.
- A **Request status related fact (RSRF)** is similar to an *RRF* but related to the request status.
- An **Activity life cycle state (ALCS)** is a base class required to describe the activity life cycle states.
- A **Request life cycle state (RLCS)** is a base class required to describe the request life cycle states.

Let us explain the reasons why the activity and request life cycle stages concepts are required. The first reason is production rules simplification. In most cases in order to define rules for the specific tasks (e.g. building with *maven*, running *Java* application, etc.) only rules for stages ALC-W and ALC-A (introduced in Table 3) have to be defined. For other stages the default behavior

provided by the axioms (see Section 3.4) is sufficient. The second one is the correspondence between the life cycle stages and the steps experts usually do while solving technical problems as Table 3 and Table 4 shows.

Table 3. Activity life cycle (ALC) stages and corresponding actions

Code	*Stage*	*Description*
ALC-P	Preparation	Copying facts from the request to the activity
ALC-W	Work	Actions aimed to reach the activity goal (e.g. deploy the package).
ALC-A	Analysis	Final classification of activity results (success or error). Taking actions to configure host platform in the case of activity failure.
ALC-ASP	Activity status preparation	Generating ASRFs to be transferred to the level of activity requests. Normally the transferred facts should contain activity execution results and failure recovery information
ALC-C	Completed	Activity is completed (successfully or with errors). No more actions required for this activity

Table 4. Request life cycle (RLC) stages and corresponding actions

Code	*Stage*	*Description*
RLC-P	Preparation	Copying facts from the parent activity to its subrequest
RLC-M	Main	Performing actions aimed to achieve the request goals: performing an activity and activity status analysis. In case of recoverable errors (which are fixed during ALC-A phase) – re-executing the activity
RLC-RSP	Request status preparation	Generating RSRFs to be transferred to the level of the parent activity. Normally the transferred facts should contain successful request results
RLC-C	Completed	Request is completed. The goal is achieved (probably as a result of several attempts to perform an activity) or not (the request failed). No more actions required for this request

3.3 Aliases, References, and Request Scoped Knowledge

Different activity instances might be (and often should be) *isolated* from each other: analysis of an activity instance doesn't depend on whether other activity instances (of the same or of different types) exist. An activity instance scope is supported by *Activity related facts*: only those facts which are related to this instance should be regarded. Such facts *grouping* allows the *Activity* to represent a fact interpretation context. Nonetheless, there are cases when neither grouping is acceptable, nor isolation. Quite the reverse, it might be required that the facts

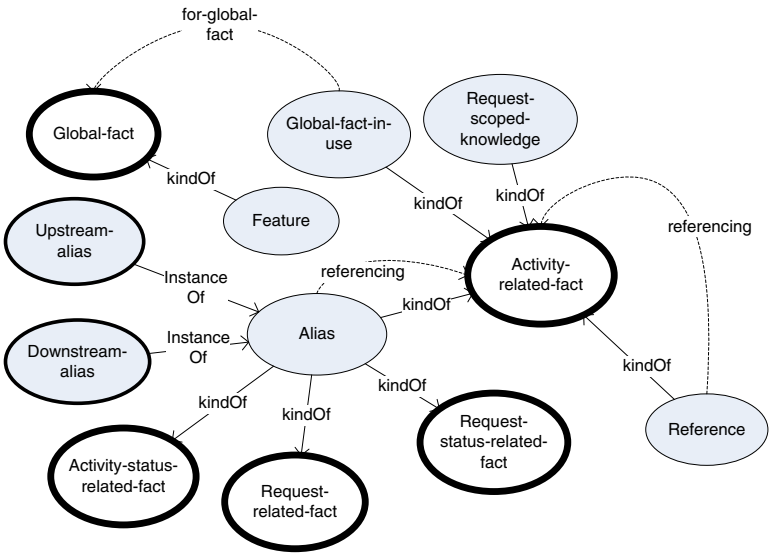

Fig. 8. Aliases, references, and request scoped knowledge representation

are allowed to be transferred from one activity to another. We introduce two ways of such a transfer: *vertical* and *horizontal* transfer (according to the *Activity/Request* graph layout shown in Figure 6). There are four types of vertical facts transfer:

- Transferring facts from the parent *request* to the child *activity* (**RA-down-transfer**): transferring input arguments from the request to the activity.
- Transferring facts from the child *activity* to the parent *request* (**AR-up-transfer**): transferring activity execution results to the parent request.
- Transferring facts from the parent *activity* to the child *request* (**AR-down-transfer**): transferring input arguments from the activity to the request.
- Transferring facts from the child *request* to the parent *activity* (**RA-up-transfer**): transferring (sub)request results to the parent activity.

The horizontal transfer is possible only between the activities within the request scope and implemented by combination of *AR-up-transfer* and *RA-down-transfer*.

To support fact transfers the following concepts are introduced (see also Figure 8):

- **Alias:** A (fact) alias is an object containing a reference to an activity related fact which, in turn, is either a request related fact or an activity related fact or a request status related fact or an activity status related fact. Hence, if an activity needs the fact to be up-transferred, this activity should create an *Upstream alias* for this fact and "attach" the *alias* to an activity status. If an activity needs the fact to be down-transferred (to the subrequest), a

Downstream alias should be created and "attached" to the activity request. The similar thing is valid for requests and request statuses.

- **Reference:** A reference is a connection to an activity related fact. At the same time a reference is an activity related fact itself. It provides a way for facts labeling for any purposes. Since the *Reference* itself constitutes an activity related fact, it is possible to transfer the label vertically and horizontally.
- **Request Scoped Knowledge:** A request scoped knowledge is an information obtained as an activity execution result. It has sense only within the context of the parent request and should be available for all the child activities. The *Request scoped knowledge* class extends the *Activity related fact* class without adding any new attributes or relationships.
- **UseGlobalFact** provides capability to bind a global fact to an activity context

The fact transfer is a standard procedure controlled by the rules introduced in Section 3.4.

3.4 Ontology Axioms

For the introduced concept model we defined the following axiomatic rules:

1. **Activity Life Cycle Axioms**
   (a) For any *activity* there is always only one *fact* describing the *activity life cycle* from the set {*ALC-P,ALC-W,ALC-A,ALC-ASP,ALC-C*}.
   (b) For any *activity* its life cycle stages follow the only allowed sequence: *ALC-P, ALC-W, ALC-A, ALC-ASP, ALC-C*. The transitions are possible only in the moments when there is no active knowledge base rules, and no other operation on action facts is performed and no incomplete subrequests (i.e. subrequests with a status different from *RLC-C*).
2. **Request Life Cycle Axioms**
   (a) For any *request* there is always only one *fact* describing the *request life cycle* from the set {*RLC-P,RLC-M,RLC-RSP,RLC-C*}.
   (b) For any *request* its life cycle stages follow the only allowed sequence: *RLC-P, RLC-M, RLC-RSP, RLC-C*. The transitions are possible only in the moments when there is no active knowledge base rules, and no other operation on action facts is performed and no incomplete child activities (i.e. child activities with a status different from *ALC-C*).
3. **Vertical Fact Transfer Axioms**
   (a) **RA-Down-Transfer Axiom.** For any *request* being in the stage *RLC-M* and any *child activity* being in the stage *ALC-P* the following is required: for any *Downstream alias* "attached" to the *request* the object that this *alias* refers to should be copied and "attached" to the *activity*.
   (b) **AR-Up-Transfer Axiom.** For any *request* being in the stage *RLC-M* and any *child activity* being in the stage *ALC-ASP* the following is required: for any *Upstream alias* "attached" to the *activity status* this *alias* should be copied and "attached" to the *request status*.

(c) **AR-Down-Transfer Axiom.** For any *activity* being in the stage *ALC-W* or *ALC-A* and any *child request* being in the stage *RLC-P* the following is possible: for any *fact* "attached" to the *activity* a *Downstream alias* can be created and "attached" to the *request*.

(d) **RA-Up-Transfer Axiom.** For any *activity* being in the stage *ALC-W* or *ALC-A* and any *child request* being in the stage *RLC-RSP* the following is required: for any *Upstream alias* "attached" to the *request status* the object that this *alias* refers to should be copied and "attached" to the *activity*.

4. **Horizontal Fact Transfer Axiom:** For any *request* being in the stage *RLC-M* and any *child activity* being in the stage *ALC-ASP* the following is required: for any *request scoped knowledge* fact created in the activity stages ALC-W or ALC-A and "attached" to the *activity* the *Downstream alias* should be created and attached to the *request*. Note that in contrast to the *AR-up-transfer* axiom, the *alias* is "attached" to the *request* itself, not to the *request status*. With using the *RA-down-transfer* axiom, it means that the *request scoped knowledge* is transferred to all new *activities*.
5. **Axiom about Creating an Activity in Response to the Request:** For any *request* if there is no child activity at all or if all the existing *child activities* are in the stage *ALC-C* and all the *child activities* are completed with error, and for every *child activity* at least one *ErrorFixed* fact exists, then a new *activity* should be created and "attached" to the current *request*.
6. **Request Status Axioms**
   (a) **Successful Request Axiom:** For any *request* if all the *child activities* are in the stage *ALC-C* and there is at least one *child activity* with status *activity succeeded*, then the *request* is completed with status *request succeeded*.
   (b) **Failed Request Axiom:** For any *request* if all the *child activities* are in the stage *ALC-C*, there is no any *child activity* with status *activity succeeded*, and conditions of the axiom (5) are not met, the *request* is completed with status *request error*.

The ontology axioms define rules of transferring facts between requests and activities. The axioms guarantee that an activity is re-performed (within the context of the same request) until either the activity goal is reached or all known error resolving procedures are examined.

3.5 Actions, Common Facts and Global Facts

The action facts are used to arrange indirect communications between a knowledge base and an execution environment. The environment executes the required actions and asserts the action status facts back into the working memory.

The important point is that action fact definitions allow retaining the *declarative* (non-imperative) form of the knowledge base rules. For example, if it is required to execute an application, the knowledge base simply asserts an *ExecAction* fact to be performed by the executor (instead of running the application

by itself). However, asserting facts into the inference engine working memory does affect neither an environment nor an expert system except the case that new rules might be activated in the knowledge base. In the earlier described target architecture (see Figure 2) there is a special component aimed to process action facts – the executor. The implementation of how the executor interacts with the expert system can affect the implementation of the knowledge base rules. For our implementation we consider a model of deferred action execution (where action facts are executed if and only if there is no more any active rule left in the inference engine schedule). In the moment when there is no more any rule scheduled for the inference engine, the executor takes control. The executor checks whether there are the action facts in the working memory. As soon as the action is executed, the new facts representing execution results are asserted into the working memory. The processed facts are then removed from the working memory.

We introduced several common actions as Figure 9 shows. Major action facts in the software provisioning ontology are the following:

1. **ExecAction:** This fact represents a shell command to be executed. The execution result is asserted into the working memory in the form of an *ExecStatus* fact.
2. **AddFeatureAction:** This fact means that the execution environment needs to be changed (some external components should be added or deleted). The *AddFeatureAction* is an abstract entity. Its subclasses should be defined (for example, a subclass *AddJDK7*) in order to communicate properly with the configuration manager.
3. **UserAction:** This fact represents a requirement that some action has to be performed by a user

Common facts (see also Figure 9) are interpreted within the context of some activity or some request (i.e. common facts are related fact instances). In contrast, global facts are context independent. Major common facts are represented by the $Artifact$, $ExecCommand$ and $UserInfo$ classes. Here is the explanations of major common facts:

1. **Artifact** is an artifact in the local file system, e.g. a file ($FileArtifact$) or a folder ($FolderArtifact$).
2. **ArtifactRef** is an artifact (specified by a URL) in the remote file system.
3. **ExecCommand** describes a command line interpreter command. The command format includes a program name and positional and key/value arguments. The command is executed under control of the deployment manager agent in response to the action fact *ExecAction*.
4. **UserInfo** is an arbitrary information to be shown to a user.
5. **FileArtifactList** represents a list of $FileArtifact$ facts.

We assume that within the ontologies of specific tasks new common fact types may be added similar to the example as we demonstrate in section 4.

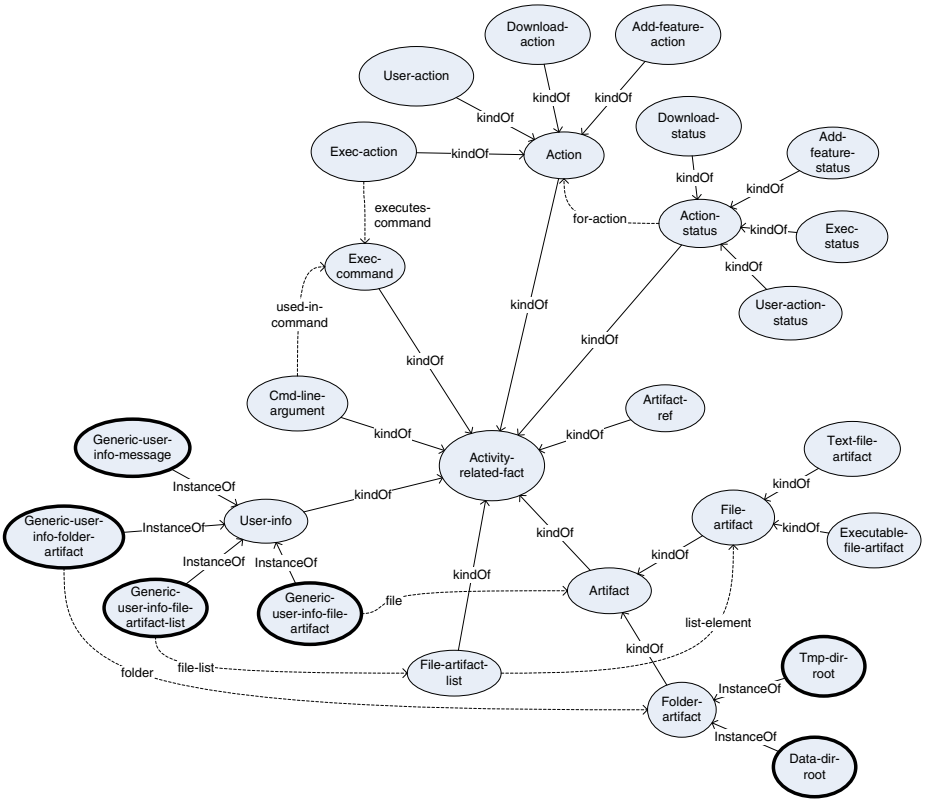

Fig. 9. Actions, artifacts and user information facts

4 Designing an Ontology of Specific Tasks

Subject domain ontologies are rarely used in expert systems directly. The reason is that such formalisms 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 base entities of the core ontology.

4.1 Running a Module Example Revisited

With respect to the introduced concepts of the core ontology, the ontology of specific tasks for the above mentioned example of running some software module M can be revised as Figure 10 shows.

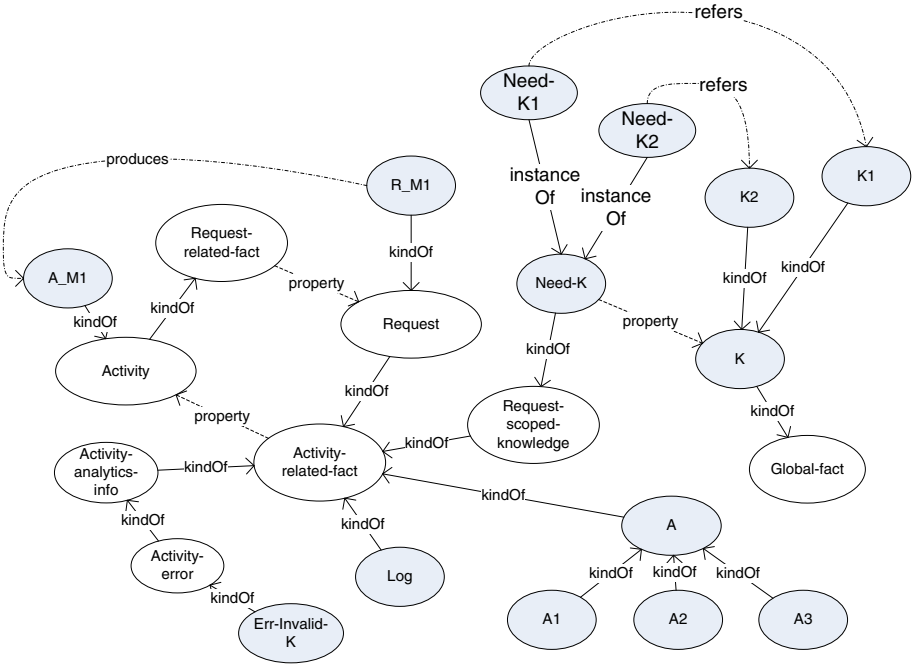

Fig. 10. Ontology of specific tasks (example)

The corresponding rules can be rewritten with the use of the redefined ontology (see Appendix).

4.2 Example of *maven* Build and *Java* Programs Execution

Figure 11 and Figure 12 demonstrate how to use the software provisioning ontology to describe the task of building an application with using *maven* build system and the task of executing a Java application

As you can see, the core ontology concepts are descriptive enough to serve as a foundation for definition of relatively complex derived specific ontologies without introducing many new terms and without revision of existing concepts. Let us note that for the sake of paper readability we skip the further detailed demonstration on 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.

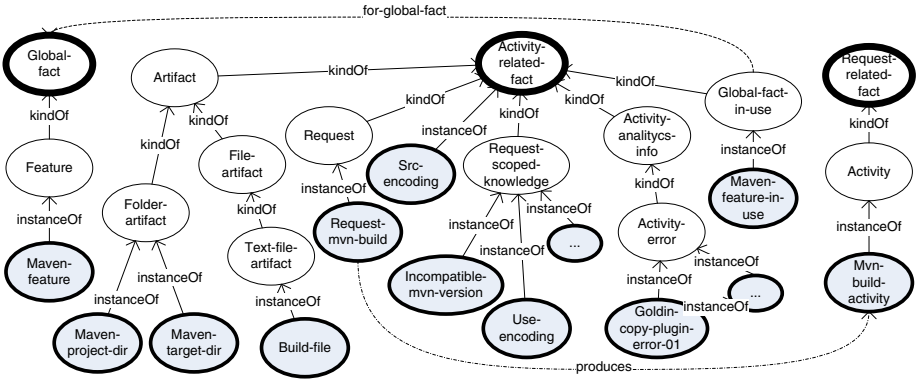

Fig. 11. An ontology of specific tasks: building applications with using *maven*

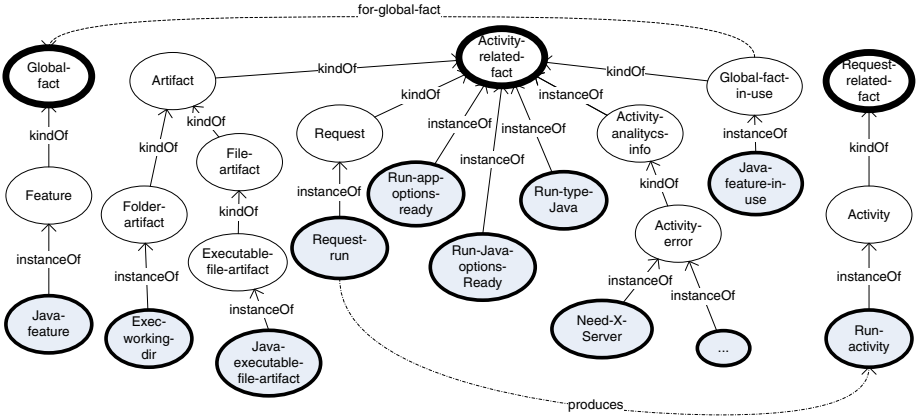

Fig. 12. An ontology of specific tasks: executing *Java* applications

5 Conclusion

In this work we defined an ontology which provides a conceptual core for building a networked environment for provisioning software applications to computing clouds and for resolving execution environment configuration errors automatically. We introduced an architecture for application deployment automation in computing clouds. By using a series of examples related to the software engineering practices and with respect to the requirements of research software we examined a question why designing an ontology is a complex problem and analyzed major iterations in the process of ontology construction.

Since the proposed general-purpose ontology can't be used directly to define expert system knowledge base rules, we demonstrated how the ontologies of specific tasks can be designed within the subject domain of building software projects with *maven* and their execution in the Java runtime environment. The general schema

is applicable to different target languages, building systems and execution environments. Unlike most of existing ontology models used in software engineering (which are focused on process *description*), the software provisioning ontology is focused on process *execution* (e.g. software build, run and environment configuration) from the perspective of a command line interpreter. It allows using the proposed ontology as a conceptual model for software execution automation.

The above mentioned ontologies of specific tasks have been formally defined[7] by using Java language constructions and used as a subject domain model for developing an expert system controlling the process of CLI applications automatic deployment. We developed the expert system which uses the knowledge base containing information about possible deployment errors and error resolution rules. Using the software provisioning ontology as a core model we defined the production rule templates describing typical tasks to be solved in order to execute the required software and to identify possible execution and configuration errors. It is important that the approach allows further modifications of the knowledge base by an expert with taking new situations discovered during the deployment stage into consideration. After adding new rules to the knowledge base such newly detected errors can be resolved automatically.

On the base of the core ontology we implemented a method for CLI software automatic deployment in *PaaS* and *IaaS* clouds. Unlike to traditional scenarios, our approach doesn't require platform *pre-configuration* nor platform configuration *description* (by using descriptor files or scripts), but allows installing necessary modules automatically or guided by a user in interactive mode (i.e. a user is able to choose one of several possible actions suggested by the deployment system).

We tested the approach in a prototype deployment system by using series of model examples and two research projects [23,24] which proved suitability of our approach to support automatic CLI-based software provisioning to computing clouds. In further works we plan to describe the prototype system architecture and a deployment manager implementation in more details, as well as to arrange a series of experiments with a selection of MIR research projects.

References

1. Mirex home, `http://www.music-ir.org/mirex/wiki/mirex_home`
2. Cannam, C., Benetos, E., Mauch, M., Davies, M.E., Dixon, S., Landone, C., Noland, K., Stowell, D.: Mirex 2014: Vamp plugins from the centre for digital music
3. West, K., Kumar, A., Shirk, A., Zhu, G., Downie, J., Ehmann, A., Bay, M.: The networked environment for music analysis (nema). In: 2010 6th World Congress on Services (SERVICES-1), pp. 314–317 (July 2010)
4. Bunch, C.: Automated Configuration and Deployment of Applications in Heterogeneous Cloud Environments. PhD thesis, Santa Barbara, CA, USA, AAI3553710 (2012)
5. Kuznetsov, A., Pyshkin, E.: 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 2(193), 110–125 (2014)

[7] `https://github.com/andrei-kuznetsov/fpf4mir/tree/master/fpf4mir-core`

6. Rabkin, A., Katz, R.: Precomputing possible configuration error diagnoses. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 193–202 (November 2011)
7. Zhang, S.: Confdiagnoser: An automated configuration error diagnosis tool for java software. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 1438–1440. IEEE Press, Piscataway (2013)
8. Dong, Z., Ghanavati, M., Andrzejak, A.: Automated diagnosis of software misconfigurations based on static analysis. In: 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 162–168 (November 2013)
9. Ya-Yunn, S., Attariyan, M., Flinn, J.: Autobash: Improving configuration management with operating system causality analysis. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles, pp. 237–250. Stevenson (2007)
10. Attariyan, M., Flinn, J.: Automating configuration troubleshooting with dynamic information flow analysis. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI 2010, pp. 1–11. USENIX Association, Berkeley (2010)
11. Zhao, Y., Dong, J., Peng, T.: Ontology classification for semantic-web-based software engineering. IEEE Trans. Serv. Comput. 2(4), 303–317 (2009)
12. Liao, L., Qu, Y., Leung, H.: A software process ontology and its application. In: First Intl. Workshop on Semantic Web Enabled Software Eng. (November 2005)
13. Caralt, J., Kim, J.W.: Ontology driven requirements query. In: 40th Annual Hawaii International Conference on System Sciences, HICSS 2007, pp. 197c–197c (January 2007)
14. Ambrosio, A., de Santos, D., de Lucena, F., da Silva, J.: Software engineering documentation: an ontology-based approach. In: Proceedings of the WebMedia and LA-Web, pp. 38–40 (October 2004)
15. Shahri, H.H., Hendler, J.A., Porter, A.A.: Software configuration management using ontologies (2007)
16. Psl core, `http://www.mel.nist.gov/psl/psl-ontology/psl_core.html`
17. Web services business process execution language version 2.0 (oasis standard April 11, 2007), `http://docs.oasis-open.org/wsbpel/2.0/os/wsbpel-v2.0-os.html`
18. Business process model and notation version 2.0 (bpmn 2.0) (omg standard January 02, 2011), `http://www.omg.org/spec/bpmn/2.0/pdf/`
19. Aitken, S.: Process representation and planning in cyc: From scripts and scenes to constraints (2001)
20. Pyshkin, E., Kuznetsov, A.: A provisioning service for automatic command line applications deployment in computing clouds. In: IEEE Proceedings of the 16th IEEE Conference on High-Performance Computing and Communications, pp. 526–529 (2014)
21. Gruber, T.R.: A translation approach to portable ontology specifications. Knowl. Acquis. 5(2), 199–220 (1993)
22. Gruber, T.R.: Toward principles for the design of ontologies used for knowledge sharing. Int. J. Hum.-Comput. Stud. 43(5-6), 907–928 (1995)
23. Glazyrin, N.: Audio chord estimation using chroma reduced spectrogram and self-similarity. In: Proceedings of the Music Information Retrieval Evaluation Exchange, MIREX (2012)
24. Khadkevich, M., Omologo, M.: Large-scale cover song identification using chord profiles. In: de Souza Britto Jr., A., Gouyon, F., Dixon, S. (eds.) ISMIR, pp. 233–238 (2013)

Appendix: Knowledge Base Rules Revised According to the Edited Ontology of Specific Tasks

```
EI:
  if  // Error identification
    $activity : A_M1()
    ALCWork(activity == $activity)
    Log(activity == $activity, text contains ‘‘dependency K[0-9] missed’’)
  then
    assert ErrInvalidK($activity)
    assert ActivityFailed($activity)
    logger.print(‘‘Activity failed because of missed component K’’)
end

SI:
  if  // Success identification
    $activity : A_M1()
    ALCWork(activity == $activity)
    Log(activity == $activity, text contains ‘‘success’’)
  then
    assert ActivitySucceeded($activity)
    logger.print(‘‘Activity succeeded’’)
end

R1:
  if  // need K1
    $activity : A_M1()
    ALCAnalyze(activity == $activity)
    $err : ErrInvalidK(activity == $activity)
    Log(activity == $activity, text contains ‘‘dependency K1 missed’’)
  then
    assert NeedK1($activity)
    assert ActivityErrorFixed($activity, $err)
    logger.print(‘‘Problem fixed - using K1 next time’’)
end

R2:
  if  // need K2
    $activity : A_M1()
    ALCAnalyze(activity == $activity)
    $err : ErrInvalidK(activity == $activity)
    Log(activity == $activity, text contains ‘‘dependency K2 missed’’)
  then
    assert NeedK2($activity)
    assert ActivityErrorFixed($activity, $err)
```

```
    logger.print(‘‘Problem fixed - using K2 next time’’)
end

R’:
  if  // run with no dependencies
    $activity : A_M1()
    ALCWork(activity == $activity)
    $a : A(activity == $activity)
    not NeedK(activity == $activity)
  then
    //run M with $arguments
    $cmd := ExecCommand($activity, M, $arguments)
    assert ExecAction($activity, $cmd)
end

R:
  if  // run with dependencies
    $activity : A_M1()
    ALCWork(activity == $activity)
    $a : A(activity == $activity)
    $dependency: NeedK(activity == $activity)
  then
    //run M with $dependency and $arguments
    $cmd := ExecCommand($activity, M, $dependency, $arguments)
    assert ExecAction($activity, $cmd)
end
```