УДК 004.41:004.822

А.Н. Кузнецов, Е.В. Пышкин

ОНТОЛОГИЯ СБОРКИ, КОНФИГУРАЦИИ ОКРУЖЕНИЯ И ЗАПУСКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И ЕЕ ПРИМЕНЕНИЕ ПРИ АВТОМАТИЗАЦИИ РАЗВЕРТЫВАНИЯ КЛИЕНТСКИХ ПРОГРАММ В ВЫЧИСЛИТЕЛЬНЫХ ОБЛАКАХ

A.N. Kuznetsov, E.V. Pyshkin

ONTOLOGY OF SOFTWARE BUILDING, EXECUTION AND ENVIRONMENT CONFIGURATION AND ITS APPLICATION IN SOFTWARE DEPLOYMENT IN COMPUTING CLOUDS

Рассмотрены аспекты сборки, конфигурации, развертывания и запуска приложений в облачной среде. Уделено внимание проблемам, с которыми сталкиваются разработчики, будучи специалистами в определенной предметной области, но не являясь при этом экспертами в области программирования. На примере проблем, возникающих при тестировании алгоритмов в области информационного поиска, проиллюстрированы основные подходы к организации исполнения консольных клиентских программ в облачной платформе. Предложена онтология предметной области, описывающая процессы построения программного кода и его запуска, а также ошибки, возникающие во время построения и запуска, и действия, необходимые для устранения возникших ошибок. Приведены примеры определения онтологий конкретных задач для построения исходного кода средствами maven и запуска java приложений, сформулированы рекомендации по использованию предложенной онтологии для описания правил базы знаний экспертной системы.

СЕРВИСНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА; ОБЛАЧНЫЕ ВЫЧИСЛЕНИЯ; ОНТОЛОГИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ; ИНФОРМАЦИОННЫЙ ПОИСК; СЕМАНТИЧЕСКАЯ СЕТЬ.

This paper is focused on software applications build, configuration and execution in cloud frameworks. We pay special attention to the case when developers being experts in their subject domain aren't experienced enough in programming. By examples of problems existing in the area of testing information retrieval algorithms we examine major approaches to support CLI client software execution in clouds. We introduce a subject domain ontology describing processes of software code building, its execution, build and execution errors as well as actions which are necessary to fix recognized errors. We cite examples of concrete tasks ontology definition to describe code building activities using maven and software execution activities using java. The recommendations on how to use the proposed ontology to define expert system knowledge base rules are given.

SERVICE-ORIENTED ARCHITECTURE; CLOUD COMPUTING; SOFTWARE ONTOLOGY; INFORMATION RETRIEVAL; SEMANTIC NETWORKS.

Как известно, облачные вычисления реализуют концепцию сервисно-ориентированного подхода к распределенным вычислениям. Она позволяет клиентам быстро получать ресурсы, такие как виртуальные машины или сетевые хранилища, и возвращать их, оплачивая только время или количество использованных ресурсов. До настоящего времени облачные вычисления рассматривались, преимущественно, в контексте проектирования и предоставления web-сервисов. С удешевлением облачных ресурсов различные научные сообщества начинают исследовать возможность при-

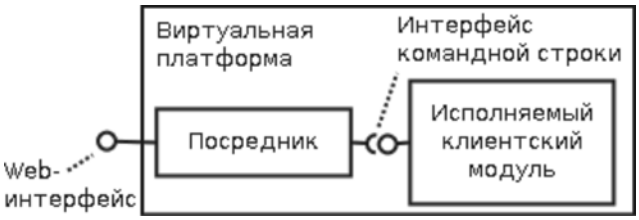

Рис. 1. Применение посредника при развертывании консольного ПО в облачном окружении

менения облачных сервисов в своих областях [1].

Традиционно многие исследователи разрабатывают ПО, реализующее исследуемые методы, с целью проведения сравнений разрабатываемого алгоритма с уже существующими. Поскольку в ряде случаев исполнение программного кода занимает длительное время, целесообразно выполнять такие испытания в облаке, арендовав нужное количество ресурсов на короткое время. Другое преимущество облачного приложения заключается в том, что его можно сделать публичным, предоставив API для доступа к приложению, обеспечивая таким образом доступ к исследуемому алгоритму другим членам научного сообщества.

В целях упрощения использования облачных сервисов в научных исследованиях разрабатываются различные платформы и инструменты для развертывания приложений. Эти инструменты делятся на две группы:

автоматизирующие развертывание приложений, разработанных для исполнения в вычислительных кластерах;

автоматизирующие развертывание приложений, разработанных для исполнения на ПК.

Предметом нашего исследования является вторая группа ПО. Основное допущение, которое делается в рамках работ по автоматизации развертывания приложений этой группы (см., например, [1]), заключается в том, что требуется правильно сконфигурировать облачную инфраструктуру. При этом сами вычислительные среды, предоставленные PaaS или IaaS сервисом (далее эти среды будем называть *виртуальной платформой*, когда различия между PaaS и IaaS несущественны), уже сконфигурированы нужным образом, т. е. все зависимости развертываемого ПО от сторонних компонентов удовлетворены. Часто это предположение оказывается неверным, что значительно усложняет процесс развертывания, равно как и сам механизм использования облачных сервисов для научных исследований.

Особенно ярко указанная проблема проявляется в областях науки, где исследователи, являясь экспертами в своей области, не являются экспертами в разработке компьютерных программ. Примером может служить информационный поиск (ИП). Традиционно, для сравнения алгоритмов, исследователи в области ИП используют метрики, вычисленные для комбинации «алгоритм-корпус». Такой подход предполагает реализацию алгоритма, причем разработанные программные модули чаще всего являются обычными консольными приложениями[1], разработанными для настольных ПК. Для развертывания такого приложения в облаке требуется *посредник*, предоставляющий web-интерфейс к консольному приложению, как показано на рис. 1.

В данной статье мы рассматриваем проблемы автоматического выявления и устранения ошибок запуска приложений, вызванных неправильной конфигурацией *виртуальной платформы*, на которой производится запуск клиентского модуля, а также предлагаем онтологию для описания и решения возникающих проблем с помощью баз знаний.

[1] То есть речь идет о приложениях с интерфейсом командной строки. Такие приложения не предполагают интерактивное взаимодействие с пользователем, а используют аргументы командной строки как основной метод коммуникации с пользователем.

Обзор предметной области

Хотя настоящее исследование посвящено онтологии процесса сборки, запуска ПО и конфигурации среды исполнения (для краткости будем называть эту онтологию *Запуск ПО*), с учетом контекста работы, а именно автоматического развертывания консольного ПО в облачных инфраструктурах и платформах (далее для краткости будем называть их *облачными структурами*, когда различия между PaaS и IaaS несущественны), необходимо обратить внимание на исследования, которые не имеют прямого отношения к онтологиям, но связаны с задачами нашей предметной области.

NEMA (Networked Environment for Music Analysis) [2]. Эта платформа предоставляет распределенную сетевую инфраструктуру для исследований в области информационного поиска музыки (MIR, Music Information Retrieval). Предлагаемая архитектура позволяет авторам публиковать реализации своих алгоритмов в виде сервисов с определенным интерфейсом. Развертывание производится в ручном режиме, поэтому авторы развертываемых программных модулей должны обладать определенными экспертными знаниями и навыками для публикации своего алгоритма в данной инфраструктуре.

MEDEA (Message, Enqueue, Dequeue, Execute, Access) [1]. Данная система позволяет развернуть произвольное консольное приложение в облаке. Основное внимание уделено развертыванию приложений в различных облачных сервисах (Google App Engine, EC2, Eucalyptus, MS Azure), при этом преследуются две основные цели: оценить производительность различных сервисов при решении одной и той же задачи; предоставить возможность развертывать произвольные приложения с интерфейсом командной строки в произвольном облачном сервисе. Для развертывания приложения пользователь системы должен предоставить дополнительную информацию, в т. ч. требуемое окружение, где под «требуемым окружением» понимается строка, описывающая среду исполнения, например «Java» или «Python». При развертывании приложения в виртуальной платформе будет установлено необходимое окружение. Главным недостатком MEDEA является то, что ошибки, возникающие в процессе исполнения клиентского кода (в т. ч. и те, которые связаны с неправильной конфигурацией среды исполнения), не анализируются.

Обнаружение и диагностика ошибок конфигурации приложения, а также зависимостей программного модуля от сторонних библиотек и компонентов обычно выполняется двумя способами: путем статического анализа кода и объектных модулей или путем динамического анализа результатов исполнения программы на тестовом наборе данных. Для первого подхода разработано множество методов и алгоритмов, например ConfAnalyzer [3], ConfDiagnoser [4], ConfDebugger [5] и пр. Известными инструментами, реализующими второй подход, являются AutoBash [6] и ConfAid [7]. Для выявления ошибочных ситуаций используются предикаты (реализованные в виде скриптов), возвращающие значение *ложь*, если обнаружена ошибка конфигурации. AutoBash и ConfAid интенсивно используют модифицированное ядро Linux и программные отладчики бинарного кода, что усложняет использование этих утилит в облачных структурах из-за ограничений политик безопасностей поставщика облачных ресурсов. Необходимость модифицированного ядра и отладчиков можно отнести к недостаткам реализации, в то время как другой, более существенный, недостаток заключается в том, что предметная область диагностирования ошибок конфигурации ПО определена в указанных работах неявно. Ключевыми понятиями являются *ошибка* (определяется с помощью предиката) и *решение* (определяется с помощью скрипта, исправляющего ошибку). Ошибка вызвана *неправильной конфигурацией*, однако само понятие *неправильной конфигурации* и атрибуты этого понятия явно не описаны, при этом они тесно связаны с реализацией утилит и форматом данных, которые эти утилиты собирают и обрабатывают.

Для формального описания связей между ошибкой и решением, а также формального описания ошибки требуется примене-

ние развитых формализмов, используемых в области инженерии знаний. К таким формализмам можно отнести языки описания процессов (BPEL, BPMN, PSL, Cyc и пр.), онтологии, а также базы знаний.

На настоящий момент нет широко известных онтологий[2], описывающих процесс сборки, запуска ПО и конфигурации среды исполнения (для краткости будем называть эту онтологию *Запуск ПО*). Тем не менее существует несколько широко известных онтологий, описывающих область программной инженерии. В работе [9], посвященной классификации онтологий в области программной инженерии, выявлено 19 типов онтологий, покрывающих различные аспекты проектирования, документирования, тестирования и поддержки программных продуктов. Наиболее релевантными данной работе являются следующие типы онтологий: онтология процесса разработки (software process ontology) [10], онтология поведения системы (system behavior ontology) [11], онтология программных артефактов (software artifact ontology) [12], онтология конфигурации системы (system configuration ontology) [13]. Каждая из перечисленных онтологий оперирует определенным подмножеством понятий, необходимых для описания предметной области *Запуск ПО*, но ни одна из представленных онтологий не описывает предметную область целиком.

Помимо онтологий предметной области программной инженерии, широко известны метаонтологии, способные описывать обширный круг задач. Наиболее релевантной нашей области является метаонтология процессов (process ontology), включающая в себя такие понятия, как процесс (process), деятельность (activity) и пр. Примерами подобных онтологий являются языки PSL (process specification language) [14] и Cyc [15]. Кроме того, существуют языки описания процессов, ориентированных на формальное исполнение (например, BPEL [16] и BPMN [17]). Особенность подобного рода метаонтологий заключается в том, что, стремясь описать широкий круг задач, они не содержат конкретики для описания понятий и связей в узкой предметной области, поэтому часто требуется разработка онтологии предметной области и конкретных задач на базе существующей метаонтологии.

Постановка задачи

Рассмотрим трехзвенную архитектуру, состоящую из *виртуальной платформы*, брокера облака и менеджера автоматических развертываний консольных приложений. На *виртуальной платформе* имеется предустановленный компонент – агент менеджера развертываний, – который предоставляет web-интерфейс к консольному приложению, собирает информацию о состоянии консольного приложения и окружения и содержит базу знаний для решения проблем развертывания клиентского ПО. Агент взаимодействует с менеджером посредством высокоуровневых команд, описывающих необходимые действия по реконфигурации платформы (например, «добавить JDK1.7»). Менеджер взаимодействует с брокером посредством низкоуровневых команд, обеспечивающих инсталляцию в платформу необходимых картриджей, или пересоздание виртуальных машин из других образов.

Требуется разработать такую онтологию для использования в базе знаний агента, которая описывает процессы автоматического построения, конфигурирования и запуска клиентского приложения в *виртуальной платформе*, а также ошибки, возникающие во время построения и запуска, и действия, необходимые для устранения выявленных ошибок.

Онтология запуска программ

Цель менеджера – успешное развертывание приложения в *виртуальной платфор-*

[2] «Онтология – формальное явное описание понятий в рассматриваемой предметной области (классов (иногда их называют понятиями)), свойств каждого понятия, описывающих различные свойства и атрибуты понятия (слотов (иногда их называют ролями или свойствами)), и ограничений, наложенных на слоты (фацетов (иногда их называют ограничениями ролей))» [8].

Рис. 2. Семантическая сеть, описывающая онтологию сборки и запуска консольного ПО и конфигурации окружения

ме. Под успешным развертыванием будем понимать успешный запуск клиентского ПО на пробном наборе входных данных. Для успешного запуска менеджеру требуются следующее:

наличие исполняемого модуля клиентского ПО;

тестовый набор данных;

описание формата вызова клиентского модуля из командной строки;

описание ожидаемого результата (набор файлов и их формат) для данного тестового набора данных.

Исполняемый модуль клиентского ПО загружается в *виртуальную платформу* одним из двух возможных способов. Первый способ состоит в том, что исполняемый модуль непосредственно загружается разработчиком клиентского ПО. Во втором случае исполняемый модуль является производным артефактом процесса сборки клиентского ПО из исходных текстов. При этом построение исходного кода можно описать теми же понятиями, что и запуск клиентского ПО, поскольку процесс построения можно рассматривать как запуск консольного приложения (компилятора, или более развитой системы построения кода как make, ant, maven и т. п.), а входными данными являются исходные коды клиентского ПО.

Результатами исполнения консольного приложения (клиентского модуля или системы построения) являются:

сгенерированные файлы;

выходные потоки (stdout, stderr);

код возврата.

Результат запуска может быть признан успешным, если код возврата равен нулю, а сгенерированные файлы и содержимое stdout/stderr соответствует ожиданиям. В противном случае запуск признается неуспешным.

В случае неуспешного запуска менеджер развертываний анализирует причины возникших ошибок (путем разбора выходных потоков, отчетов среды исполнения и т. д.) и пытается их решить самостоятельно путем переконфигурации *виртуальной платформы* и повторного запуска консольного приложения (клиентского ПО или системы построения исходного кода). Если автоматически решить проблему не удается, создается и рассылается уведомление заинтересованным лицам (автору клиентского ПО, администратору системы и т. п.) о невозможности автоматического развертывания.

Для описания подобного рода сценариев и с учетом устоявшейся терминологии в существующих онтологиях, разработана онтология автоматического построения и запуска консольных приложений и устранения ошибок конфигурации среды исполнения. Семантическая сеть, описывающая онтологию предметной области, представлена на рис. 2.

Все множество понятий онтологии можно разбить (по назначению) на три группы: *обычные* факты, факты-действия (action), направленные на внешние, по отношению к машине вывода, компоненты, и факты-запросы (request), описывающие действия, направленные на машину вывода.

Обычные факты описывают текущее состояние системы. Ключевыми фактами являются следующие: **Деятельность** (Activity), **Артефакт** (Artifact) и **Команда** (для интерпретатора командной строки, ExecCommand). Дадим краткое описание ключевых *обычных* фактов:

1. **ИсходныйАртефакт** (OriginalArtifact) – это артефакт, предоставленный разработчиком *клиентского модуля*, представляющий собой файл, каталог, архив или URL. В процессе работы машины вывода он является причиной добавления производного артефакта **Файл** или **Каталог**, представляющего исходный артефакт в виде файла или каталога в локальной файловой системе соответственно после скачивания, копирования или распаковки исходного артефакта.

2. **Артефакт** (Artifact) – Артефакт файловой системы. Это всегда либо один файл (**Файл**), либо один каталог (**Каталог**) в локальной файловой системе.

3. **Деятельность** (Activity) – действие эксперта (или экспертной системы) для достижения цели: **Сборка**, **Запуск** клиентского модуля. **Деятельность** является агрегирующим понятием (при помощи отношения **описывать**), и всегда имеет своей целью

запуск какой-либо **Команды**, результат исполнения которой описывается **СтатусомИсполнения**. В зависимости от кода возврата команды и, возможно, некоторых других факторов, принимается решение об **УспехеДеятельности** или **ОшибкеДеятельности**.

4. **Команда** (ExecCommand) – команда для интерпретатора командной строки. Формат команды состоит из пяти элементов, как показано на рис. 3: из имени программы и четырех групп **Аргументов** командной строки: **Позиционных{1,2}** и именованных **КлючЗначение{1,2}**. Такой формат обусловлен тем, что в качестве среды исполнения часто выступает виртуальная машина (например, Java), и первая пара аргументов – это ключи виртуальной машины, а вторая – ключи исполняемой программы. Команда исполняется в облачном окружении агентом менеджера развертываний в ответ на факт-действие **ДействиеИнтерпретатора**. По завершении исполнения агент добавляет в рабочую память **СтатусИсполнения** с информацией о коде возврата, а также содержимым выходных потоков.

Необходимость выполнить какое-либо действие отражается в онтологии фактом-действием. Такой подход позволяет сохранить правила базы знаний декларативными (а не императивными). Однако добавление фактов в рабочую память машины вывода само по себе не оказывает воздействие ни на окружающую среду, ни на экспертную систему, кроме возможной активации новых правил в базе знаний. Для обработки фактов-действий требуется специальный компонент – *исполнитель* (см. рис. 4). Взаимодействие *исполнителя* с экспертной системой может быть организовано различными способами, а реализация взаимодействия может оказывать влияние на реализацию правил в экспертной базе знаний. В рамках прототипа разрабатываемой системы автоматического развертывания консольных приложений мы предлагаем использовать позднее исполнение и прямой доступ в рабочую память машины вывода. Этот подход отличается простотой реализации и толерантностью к добавлению, удалению и изменению факта-действия вплоть до момента исполнения действия агентом.

Взаимодействие экспертной системы и *исполнителя* изображено на рис. 4.

В тот момент, когда в расписании машины вывода больше нет активных правил (т. е. правил, которые могут сработать), управление передается *исполнителю*, который анализирует рабочую память на наличие в ней фактов-действий. Выполнив необходимое действие, *исполнитель* добавляет в рабочую память машины вывода новые факты, вызванные этим действием. Например, в результате исполнения действия **ДействиеИнтерпретатора**, *исполнитель* добавит в рабочую память **СтатусИсполнения**. В целях упрощения реализации агента, обработанные факты-действия удаляются из рабочей памяти машины вывода.

В рамках онтологии предметной области мы выделили следующие основные факты-действия:

1. **ДействиеИнтерпретатора** (Exec Action) – указание на необходимость исполнить команду. Результат исполнения команды будет добавлен в рабочую память в виде факта **СтатусИсполнения**.

2. **КонфигурацияОкружения** (ConfigAction) – указание на необходимость изменить состояние окружающей среды (добавить или удалить сторонние компоненты). Не предполагается прямое использование этого класса. Вместо этого должны быть определены подклассы данного понятия (например, «ДобавитьJDK7»), которые могут быть интерпретированы менеджером развертываний.

3. **Уведомление** (UsrNfAction) – указание на необходимость отправить пользователю сообщение (например, о невозможности автоматического развертывания приложения). Аналогично предыдущему действию,

```
<имя_прог> [[<поз1>][<ключ1 зн1>]]… [[<поз2>][<ключ2 зн2>]]…
java –verbose:class –jar ./prog.jar -progArg1 val1 -progArg2
```

Рис. 3. Формат командной строки

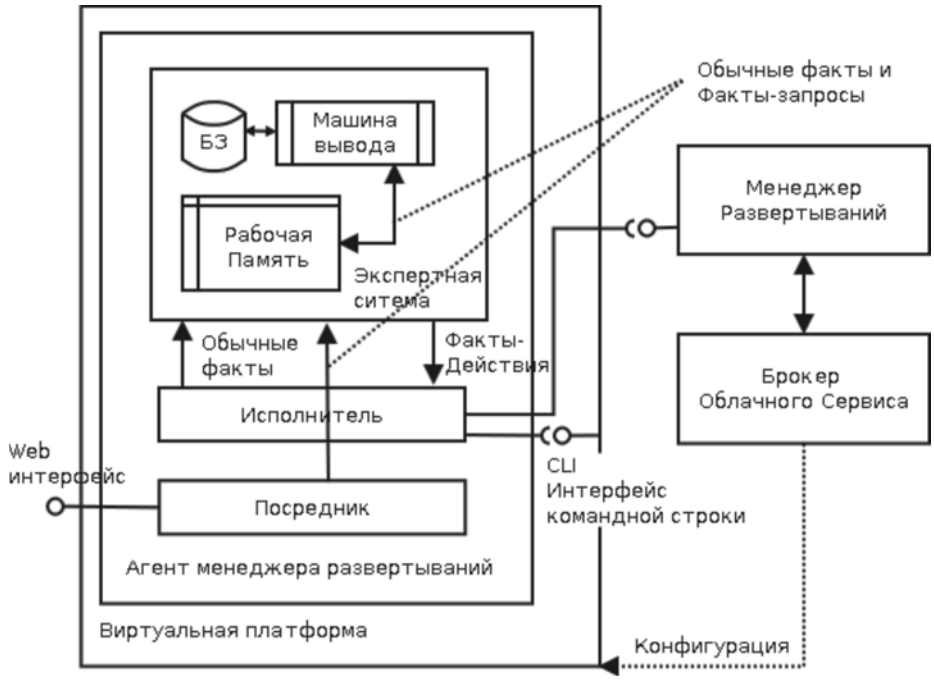

Рис. 4. Взаимодействие экспертной системы (ЭС) и менеджера развертываний

не предполагается прямое использование этого класса.

Факты-запросы по своей сути похожи на факты-действия, однако их принципиальным отличием является то, что запросы направлены на саму экспертную систему и обрабатываются машиной вывода как обычные факты. Запросы могут быть добавлены в систему как извне (например, *посредником*), так и самой системой. Например, если пользователь системы предоставил исходные коды в качестве **ИсходногоАртефакта**, система может сгенерировать сама себе новый **ЗапросСборки**. В предлагаемой онтологии предметной области основным и единственным типом запроса является **ЗапросДеятельности**: **ЗапросЗапуска** или **ЗапросСборки**. Также предполагается, что в онтологиях конкретных задач, базирующихся на онтологии предметной области, могут появиться дополнительные типы запросов.

Онтологии конкретных задач

Онтология предметной области редко используется в экспертных системах без изменений, поскольку она не обладает нужным уровнем детализации для решения конкретных задач. Поэтому на основе онтологии предметной области создается онтология конкретной задачи, обладающая нужным уровнем детализации. В этом разделе мы продемонстрируем расширение предложенной онтологии для описания конкретных задач, на двух примерах:

1) сборка исходных кодов системой maven;

2) запуск Java приложения.

Для описания конкретной задачи мы расширим базовые понятия (классы), введенные в онтологии предметной области, так, как показано на рис. 5.

В частности, расширим понятие **Аргумент** подклассами **JАргумент** и **M2Аргумент**. Некоторые экземпляры этих понятий приведены на рисунке в качестве иллюстрации, однако список приведенных экземпляров неполный. Список экземпляров ограничен набором всех возможных ключей для Java и maven соответственно. Приведенные в качестве иллюстраций понятия имеют следующий смысл:

1. **JOptHeadless** («-Djava.awt.headless=true») – запуск Java машины в AWT Headless режиме;

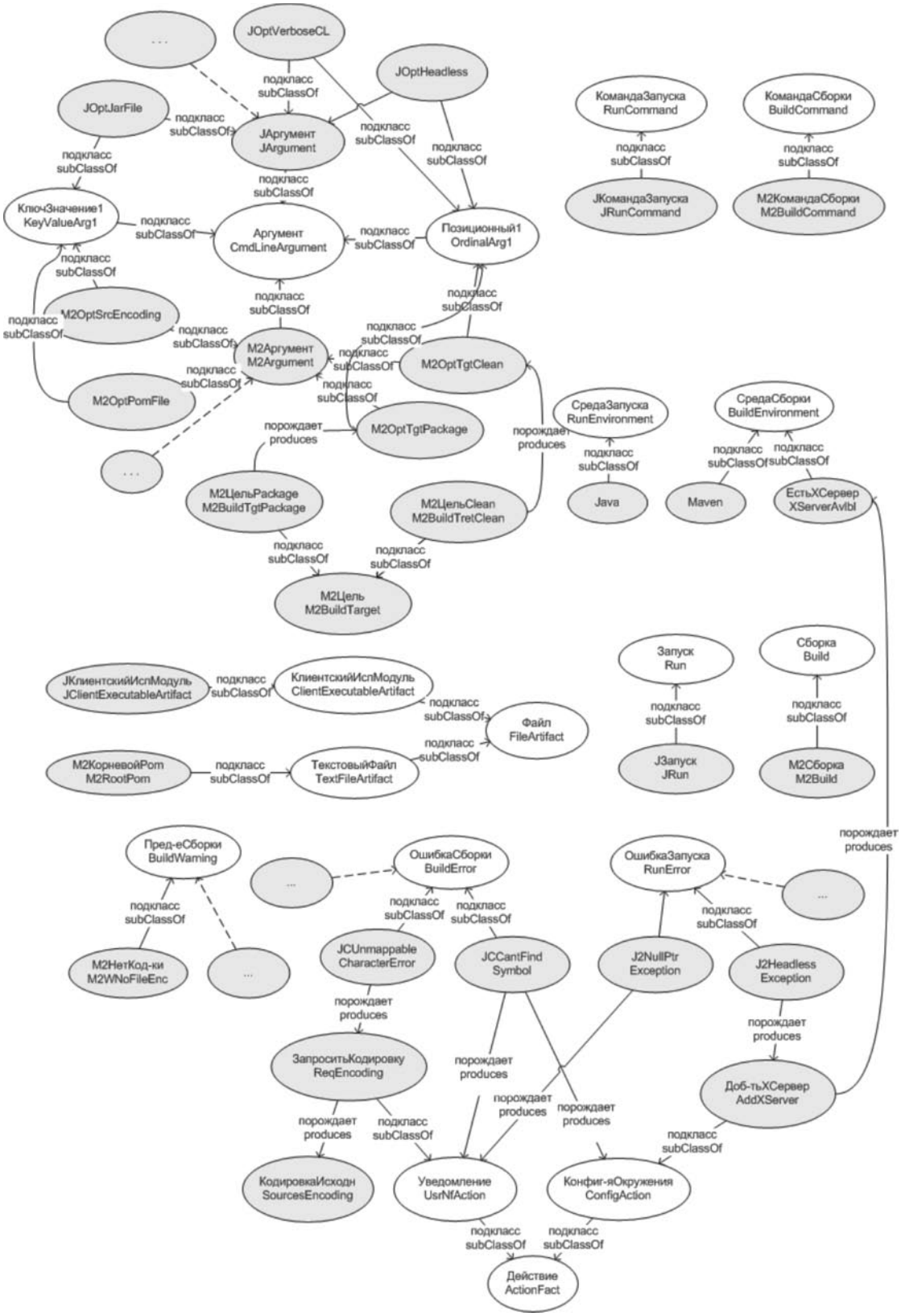

Рис. 5. Фрагмент семантической сети, описывающий онтологию сборки maven и запуска Java приложений

2. **JOptVerboseCL** («-verbose:class») – выводить лог Java компонента Class Loader;

3. **JOptJarFile** («-jar <file>») – местоположение Jar файла;

4. **M2OptTgtClean** («clean»), **M2OptTgtPackage** («package») – maven цели: clean и package соответственно;

5. **M2OptPomFile** («-f <file>») – местоположение pom файла;

6. **M2OptSrcEncoding** («-Dproject.build.sourceEncoding=<encoding>») – кодировка ресурсов maven (в т. ч. исходных файлов).

Другое важное расширение онтологии заключается во введении понятий **JЗапуск** (запуск Java программы) и **M2Сборка** (сборка maven), расширяющие базовые понятия **Запуск** и **Сборка**, а также введение конкретных типов ошибок и предупреждений maven и Java. Аналогично расширению аргументов, перечисленные экземпляры ошибок являются иллюстрацией, поскольку полный набор экземпляров измеряется десятками сотен. В примере на рис. 5 приведены следующие типы maven, Java и Javac (компилятор Java) ошибок:

1. **M2НетКодировки** – предупреждение maven сборки «WarnNoFileEncoding» о том, что для проекта не указана исходная кодировка. Это предупреждение можно игнорировать.

2. **J2HeadlessException** – ошибка запуска Java кода: невозможно исполнение в Headless режиме. В ответ на эту ошибку экспертная система обычно добавляет факт-запрос на реконфигурацию среды: **ДобавитьXСервер**.

3. **J2NullPointerException** – ошибка запуска Java кода: NullPointerException. Обычно эту ошибку невозможно исправить автоматически, т. к. она часто связана с ошибками в клиентском коде. Необходимо отправить уведомление о невозможности продолжения запуска.

4. **JCCantFindSymbol** – ошибка Java компилятора при сборке: невозможно найти символ. Часто связана либо с ошибками компиляции других классов этого же проекта, либо с отсутствующим jar файлом, содержащим необходимые определения. Эту ошибку можно диагностировать по полному имени отсутствующего символа. Для широко известных имен классов (org.apache.*, например) эта ошибка может быть исправлена автоматически путем добавления необходимых jar файлов в classpath во время компиляции. Для нестандартных имен классов эту ошибку автоматически исправить невозможно, поэтому будет создано соответствующее уведомление пользователям системы.

5. **JCUnmappableCharacterError** – ошибка Java компилятора при сборке: невозможно разобрать входной файл. Обычно возникает из-за проблем с кодировками. Например, если в java файле есть комментарии на русском языке и файл сохранен в кодировке cp1251, а javac пытается читать этот же файл как UTF8. Обычно проблема решается путем явного указания кодировки исходного файла.

Приведенный выше список классов, описывающих аргументы и ошибочные ситуации Java и maven, не является полным, а является иллюстрацией того, каким образом можно расширять онтологию предметной области для решения конкретных задач. В следующем разделе мы продемонстрируем как использовать расширенную онтологию при описании правил в продукционных моделях баз знаний.

Применение онтологии в продукционных базах знаний

В этом разделе мы опишем основные шаблоны правил, покрывающие процесс построения, запуска исходного кода и диагностирование ошибок, возникающих в процессе построения и запуска, на примере процесса сборки кода системой maven. Эти шаблоны можно считать рекомендациями по использованию онтологии. Представленный набор правил является упрощенным подмножеством правил, который был опробован на нескольких реальных maven проектах в рамках исследования. Для описания правил мы будем использовать язык Drools [18], задающий правила в форме «когда-тогда» («when-then», «если-то»), который ориентирован на понимание неэкспертами в области программирования.

В общем виде причиной какой-либо деятельности является запрос этой деятельности. Если ЭС определила, что в ка-

честве **ИсходногоАртефакта** были предоставлены исходные коды программы, то ЭС может сама себе создать **ЗапросПостроения**. В результате такого запроса будет создана соответствующая деятельность (Листинг 1).

ReqNewActivity::newActivityInstance является «синтаксическим сахаром». Фактически, если запрос был **ЗапросСборки**, то в рабочую память добавится **Сборка**, а если **ЗапросЗапуска**, то **Запуск**. Не представляет труда составить такое правило, которое, в зависимости **СредыСборки**, будет преобразовывать деятельность **Сборка** в соответствующий подкласс, например, **M2Сборка**.

Выполнение деятельности всегда направлено на выполнение **Команды** интерпретатора командной строки. В общем виде правило для составления команды выглядит как показано в листинге 2.

В рассматриваемом примере команду для построения кода с помощью maven можно составить как в листинге 3.

```
rule "Activity_New"
  when
    $rr : ReqNewActivity(  )
    // если есть запрос новой деятельности
    forall ( $b : Activity( request == $rr )
            // деятельность для данного запроса
            // либо не существует, либо
            // для каждой деятельности справедливо:
            ExecStatus( activity == $b, succeeded == false )
            // предыдущие попытки либо еще не завершились,
            // либо завершились с ошибками
            ActivityErrorFixed( activity == $b ) )
            // и хотя бы одна ошибка исправлена
  then
    insert( $rr.newActivityInstance() );
end
```

Листинг 1: Пример правила базы знаний, добавляющего новую **Деятельность** *в ответ на* **ЗапросДеятельности**

```
rule "SomeCommand" when
  $options : java.util.LinkedList() from collect ( CmdLineArgument() )
then
  insertLogical( new ExecCommand("some_command", $arguments) );
end
```

Листинг 2: Шаблон правила базы знаний, составляющего **Команду** *из* **Аргументов**

```
rule "Maven_BuildCommand" when
    $b : M2Build(  )
    // для каждой M2Сборки
    $options : java.util.LinkedList( size > 0 ) from
    // опции сборки - это коллекция M2Option
            collect ( M2Argument( activity == $b ) )
            // для данной деятельности
  then
    insertLogical( new M2BuildCommand($b, $options) );
end
```

Листинг 3: Пример правила базы знаний, составляющего **M2КомандуСборки** *из* **M2Аргументов** *maven*

На примере этого правила можно продемонстрировать целесообразность определения **Аргументов** как отдельных сущностей, а не атрибутов **Команды**, поскольку зависимость ключей командной строки от внешних факторов (например, кодировки исходных файлов) в этом случае можно декларативно определить так, как это показано в листинге 4.

В случае представления аргументов как атрибутов **Команды** такой декларативности достичь не удается, а в части правила, описывающего действие, пришлось бы изменять состояние атрибутов **Команды**, что ухудшает читаемость правил и производительность ЭС.

Нетрудно записать правило, формирующее **ДействиеИнтерпретатора** для **Команды** (листинг 5).

Агент менеджера развертываний выполнит соответствующее действие и добавит в рабочую память **СтатусИсполнения**, который можно проанализировать, и принять решение об успешности или неуспешности завершившейся **Деятельности** (листинг 6).

Для неуспешной деятельности можно разработать правила, диагностирующие возникшие ошибки. Например, ошибку кодировки исходного файла можно диагностировать следующим правилом (см. листинг 7).

```
rule "Maven_Option_SrcEncoding" when
    $build : M2Build()
    // для каждой M2Сборки
    $srcEnc : SrcEncoding( )
    // Если известна кодировка ресурсов
  then
    insert( new M2OpSrcEncoding($build, $srcEnc.getSrcEncoding()) );
    // Добавить соответствующую опцию M2
end
```

*Листинг 4: Пример правила базы знаний, добавляющего опцию сборки (***M2OpSrcEncoding***) в ответ на выявление нового факта о конфигурации проекта (***SrcEncoding***)*

```
rule "CommandExecutor"
  when
    $cmd : ExecCommand( )
    // Для каждой команды
    not ExecStatus( execCommand == $cmd )
    // для которой нет результата исполнения
  then
    insertLogical( new ExecAction($cmd) );
    // Создать соответствующий факт-действие
end
```

Листинг 5: Пример правила базы знаний, формирующего **ДействиеИнтерпретатора** *для* **Команды**

```
rule "BuildStatus"
  when
    $build : Build( )
    ExecStatus( activity == $build, succeeded == true )
  then
    insertLogical( new BuildSucceeded($build));
end
```

Листинг 6: Пример правила базы знаний, принимающего решение об успешности сборки

```
rule "Maven_Err_UnmappableCharacter"
  when
    $build : Build()
    // для каждой Сборки
    ExecStatus( activity == $build,
                succeeded == false,
                // Если статус указывает на ошибку
                $errLine : getSingleLine(«error: unmappable character for encoding»),
                // и в stdout или stderr есть строка «error: unmappable character...»
                $errLine != null )
  then
    insert( new JCUnmappableCharacterError($build, $errLine) );
    // то имеется проблемы с кодировкой
end
```

Листинг 7: Пример правила базы знаний, диагностирующего ошибку сборки (ошибка кодировки исходного файла)

Предложена онтология предметной области, описывающая процессы построения программного когда, его запуска, ошибки, возникающие во время построения и запуска, а также действия, необходимые для устранения возникших ошибок. Поскольку онтология предметной области оперирует слишком общими понятиями для того, чтобы на ее основе формулировать правила экспертной системы, требуются онтологии конкретных задач. В статье приведены примеры определения онтологий конкретных задач для построения исходного кода средствами maven и запуска Java программ. Также сформулированы рекомендации по использованию предложенной онтологии для описания правил базы знаний экспертной системы.

Рассматриваемые онтологии (предметной области и две онтологии конкретных задач) формально описаны на языке Java и использованы в качестве модели предметной области, применяемой при разработке экспертной системы, являющейся частью системы автоматического развертывания консольных приложений в качестве веб-сервисов. Для описания базы знаний экспертной системы использовался язык Drools и машина вывода Drools Expert 5.0.0.Final [18]. Разработанная система позволяет *автоматически* развертывать консольные приложения в облачных структурах благодаря использованию базы знаний о возможных ошибках развертываний. База знаний, разработанная в рамках исследования, не является полной (т. е. не содержит знаний о решении всех возможных проблем, возникающих в ходе развертывания ПО). Поэтому важным является не то, что развертывание выполняется автоматически с *текущим* набором правил базы, а то, что развертывание возможно выполнить в автоматическом режиме, *дополнив* базу знаний необходимыми правилами. С точки зрения эксплуатации системы, поддержкой базы знаний должен заниматься эксперт в соответствующей области. Эксперт добавляет новые правила в базу, опираясь на конкретные случаи невозможности автоматического развертывания. Добавление новых правил должно обеспечивать возможность автоматического развертывания приложения, тем самым происходит «тиражирование» знаний эксперта.

Прототип системы автоматического развертывания, использующий предлагаемую онтологию, испытывался на нескольких модельных примерах (в которых намеренно были внесены ошибки), а также на двух реальных проектах: [19] и [20]. Проведенные эксперименты не выявили ситуаций невозможности описания ошибок построения или запуска ПО в рамках предлагаемой онтологии и с помощью продукционной модели базы знаний. В настоящее время продолжается испытание прототипа на других реальных проектах, не основанных на maven и Java.

СПИСОК ЛИТЕРАТУРЫ

1. **Bunch C.** Automated Configuration and Deployment of Applications in Heterogeneous Cloud Environments // Ph.D. Dissertation. University of California at Santa Barbara, Santa Barbara, CA, USA. 2012. 245 p.

2. **West K., Kumar A, Shirk A, Guojun Zhu, Downie J.S., Ehmann A., Bay M.** The Networked Environment for Music Analysis (NEMA) // 6th World Congress on Services, IEEE Computer Society. 2010. Pp. 314–317.

3. **Rabkin A., Katz R.** Precomputing possible configuration error diagnoses // In Proc. of the 26th IEEE/ACM Internat. Conf. on Automated Software Engineering. 2011. Pp. 193–202.

4. **Zhang S.** ConfDiagnoser: An Automated Configuration Error Diagnosis Tool for Java Software // In Proc. of the Internat. Conf. on Software Engineering, IEEE Press Piscataway. 2013. Pp. 312–321.

5. **Dong Z., Ghanavati M., Andrzejak A.** Automated Diagnosis of Software Misconfigurations Based on Static Analysis // In Proc. of the 2013 IEEE Internat. Symp. on Software Reliability Engineering Workshops. 2013. Pp. 162–168.

6. **Su Y., Attariyan M., Flinn J.** AutoBash: Improving Configuration Management with Operating System Causality Analysis // In Proc. of the 21st ACM Symp. on Operating Systems Principles. 2007. Pp. 237–250.

7. **Attariyan M., Flinn J.** Automating Configuration Troubleshooting with ConfAid // Usenix ;login: Magazine. 2011. Vol. 36. No. 1.

8. **Noy N.F., Mcguinness D.L.** Ontology Development 101: A Guide to Creating Your First Ontology // Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880. 2001.

9. **Zhao Y., Dong J., Peng T.** Ontology Classification for Semantic-Web-Based Software Engineering // IEEE Trans. Serv. Comput. 2009. Vol. 2. No. 4. Pp. 303–317.

10. **Liao L., Qu Y., Leung H.K.N.** A Software Process Ontology and Its Application // In Proc. of the First Int'l Workshop Semantic Web Enabled Software Eng. 2005.

11. **Caralt J.C., Kim J.W.** Ontology Driven Requirement Query // In Proc. of the 40th Ann. Hawaii Int'l Conf. System Sciences. 2007. 197 p.

12. **Ambrosio A.P., de Santos D.C., de Lucena F.N., de Silva J.C.** Software Engineering Documentation: An Ontology-Based Approach // In Proc. of the WebMedia and LA-Web Joint Conf. 10th Brazilian Symp. Multimedia and the Web Second Latin Am. Web Congress. 2004. Pp. 38–40.

13. **Shahri H.H., Hendler J.A., Porter A.A.** Software Configuration Management Using Ontologies // In Proc. of the 3rd Int'l Workshop Semantic Web Enabled Software Eng. 2007.

14. PSL Core [электронный ресурс] / URL: http://www.mel.nist.gov/psl/psl-ontology/psl_core.html (дата обращения 06.04.2014)

15. **Aitken S.** Process Representation and Planning in Cyc: From Scripts and Scenes to Constraints. AIAI, University of Edinburgh, 2001 [электронный ресурс] / URL: http://www.aiai.ed.ac.uk/~stuart/Papers/plan01-ws.pdf (дата обращения 06.04.2014)

16. Web Services Business Process Execution Language Version 2.0 (OASIS Standard 11 April 2007) [электронный ресурс] / URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (дата обращения 06.04.2014)

17. Business Process Model and Notation Version 2.0 (BPMN 2.0) (OMG Standard 02 January 2011) [электронный ресурс] / URL: http://www.omg.org/spec/BPMN/2.0/PDF/ (дата обращения 06.04.2014)

18. Drools Expert User Guide [электронный ресурс] / URL: http://docs.jboss.org/drools/release/5.5.0.Final/drools-expert-docs/html_single/index.html (дата обращения 14.04.2014)

19. **Glazyrin N.** Audio chord estimation using chroma reduced spectrogram and self-similarity // In Proc. of the Music Information Retrieval Evaluation Exchange. 2012.

20. **Khadkevich M., Omologo M.** Large-Scale Cover Song Identification Using Chord Profiles // In Proc. of the 14th Internat. Society for Music Information Retrieval Conf. 2013. Pp. 233–238.

REFERENCES

1. **Bunch C.** Automated Configuration and Deployment of Applications in Heterogeneous Cloud Environments, *Ph.D. Dissertation*, University of California at Santa Barbara, Santa Barbara, CA, USA, 2012, 245 p.

2. **West K., Kumar A, Shirk A, Guojun Zhu, Downie J.S., Ehmann A., Bay M.** The Networked Environment for Music Analysis, *6th World*

Congress on Services, IEEE Computer Society, 2010, Pp. 314–317.

3. **Rabkin A., Katz R.** Precomputing possible configuration error diagnoses, *In Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, Pp. 193–202.

4. **Zhang S.** ConfDiagnoser: An Automated Configuration Error Diagnosis Tool for Java Software, *In Proceedings of the International Conference on Software Engineering*, IEEE Press Piscataway, 2013, Pp. 312–321.

5. **Dong Z., Ghanavati M., Andrzejak A.** Automated Diagnosis of Software Misconfigurations Based on Static Analysis, *In Proceedings of the 2013 IEEE International Symposium on Software Reliability Engineering Workshops*, 2013, Pp. 162–168.

6. **Su Y., Attariyan M., Flinn J.** AutoBash: Improving Configuration Management with Operating System Causality Analysis, *In Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007, Pp. 237–250.

7. **Attariyan M., Flinn J.** Automating Configuration Troubleshooting with ConfAid. *Usenix; login: Magazine*, 2011, Vol. 36, No. 1.

8. **Noy N.F., Mcguinness D.L.** Ontology Development 101: A Guide to Creating Your First Ontology, *Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880*, 2001.

9. **Zhao Y., Dong J., Peng T.** Ontology Classification for Semantic-Web-Based Software Engineering, *IEEE Trans. Serv. Comput.*, 2009, Vol. 2, No. 4, Pp. 303–317.

10. **Liao L., Qu Y., Leung H.K.N.** A Software Process Ontology and Its Application, *In Proceedings of the First Int'l Workshop Semantic Web Enabled Software Eng.*, 2005.

11. **Caralt J.C., Kim J.W.** Ontology Driven Requirement Query. *In Proceedings of the 40th Ann. Hawaii Int'l Conference System Sciences*, 2007, 197 p.

12. **Ambrosio A.P., de Santos D.C., de Lucena F.N., de Silva J.C.** Software Engineering Documentation: An Ontology-Based Approach, *In Proceedings of the WebMedia and LA-Web Joint Conf. 10th Brazilian Symp. Multimedia and the Web Second Latin Am. Web Congress*, 2004, Pp. 38–40.

13. **Shahri H.H., Hendler J.A., Porter A.A.** Software Configuration Management Using Ontologies, *In Proceedings of the 3rd Int'l Workshop Semantic Web Enabled Software Eng.*, 2007.

14. *PSL Core*. Available: http://www.mel.nist.gov/psl/psl-ontology/psl_core.html (Accessed 06.04.2014)

15. **Aitken S.** *Process Representation and Planning in Cyc: From Scripts and Scenes to Constraints*. AIAI, University of Edinburgh, 2001, 4 p. Available: http://www.aiai.ed.ac.uk/~stuart/Papers/plan01-ws.pdf (Accessed 06.04.2014)

16. *Web Services Business Process Execution Language Version 2.0* (OASIS Standard 11 April 2007). Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (Accessed 06.04.2014)

17. *Business Process Model and Notation Version 2.0* (BPMN 2.0) (OMG Standard 02 January 2011) Available: http://www.omg.org/spec/BPMN/2.0/PDF/ (Accessed 06.04.2014)

18. *Drools Expert User Guide*. Available: http://docs.jboss.org/drools/release/5.5.0.Final/drools-expert-docs/html_single/index.html (Accessed 14.04.2014)

19. **Glazyrin N.** Audio chord estimation using chroma reduced spectrogram and self-similarity, *In Proceedings of the Music Information Retrieval Evaluation Exchange*, 2012.

20. **Khadkevich M., Omologo M.** Large-Scale Cover Song Identification Using Chord Profiles. *In Proceedings of the 14th International Society for Music Information Retrieval Conference*, 2013, Pp. 233–238.

КУЗНЕЦОВ Андрей Николаевич – *аспирант Санкт-Петербургского государственного политехнического университета.*

195251, Санкт-Петербург, ул. Политехническая, д. 29.

E-mail: andrei.n.kuznetsov@gmail.com

KUZNETSOV, Andrey N. *St. Petersburg State Polytechnical University.*

195251, Politekhnicheskaya Str. 29, St. Petersburg, Russia.

E-mail: andrei.n.kuznetsov@gmail.com

ПЫШКИН Евгений Валерьевич – *доцент Санкт-Петербургского государственного политехнического университета, кандидат технических наук.*
195251, Санкт-Петербург, ул. Политехническая, д. 29.
E-mail: evgeny.pyshkin@gmail.com

PYSHKIN, Evgeniy V. *St. Petersburg State Polytechnical University.*
195251, Politekhnicheskaya Str. 29, St. Petersburg, Russia.
E-mail: evgeny.pyshkin@gmail.com