УДК 004.6+ 004.4

Ресурсы и инструменты для преподавания методов и средств Semantic Web

Апанович З.В. (Институт систем информатики СО РАН, Новосибирский государственный университет)

За последние годы создано значительное количество структурированных данных как научными так и коммерческими организациями. На базе этих данных разрабатываются многочисленные приложения, что делает более важным и нужным знакомство с этим направлением современных ИТ-специалистов. В данной работе обсуждаются как важные аспекты эволюции направления Semantic Web, так и опыт преподавания методов и средств Semantic Web.

Ключевые слова: *Открытые Связанные Данные, RDF, RDFS, OWL, SPARQL.*

1. Введение

С момента возникновения идеи Semantic Web в 2001 это направление стремительно развивалось. За последние годы размер Облака Связанных Данных[1] значительно возрос, и в настоящее время насчитывает порядка 10 000 наборов данных, содержащих в общей сложности более 150 миллиардов триплетов. К наиболее часто используемым наборам связанных данных относятся [1]:

- Europeana, объединяющая метаданные для цифровых объектов из музеев, архивов и аудиовизуальных архивов по всей Европе;
- Linked Data Service Библиотеки Конгресса США, использующая более 50 словарей;
- OCLC WorldCat Linked Data, каталог из более чем 370 миллионов библиографических записей, экспериментально доступных в форме Связанных данных;
- VIAF, Виртуальный международный авторитетный файл OCLC, объединяющий более 40 авторитетных файлов из разных стран и регионов. К этим наборам данных осуществляется более 100 000 запросов в день.

---

[1] http://linkeddata.org/

Значительное количество структурированных данных, не являющихся открытыми, создано также коммерческими организациями. Идея Semantic Web была подхвачена гигантами Интернета, такими как Google, Microsoft, Facebook, в результате чего возникли огромные графы знаний, такие как Google Knowledge graph[2] , Microsoft Satori[3], извлекающие информацию из источников данных различной природы, и позволяющие значительно улучшить качество поиска информации. В таблице 1 [2] приведена информация о размерах наиболее известных в настоящее время графах знаний. Экземпляры (instances) означают экземпляры классов (концепты А-боксов), под количеством фактов понимается количество триплетов RDF, количество типов - это количество различных типов или классов, определенных в схеме, количество отношений – количество различных отношений, определенных в схеме.

Помимо этого, публикация структурированных данных в HTML контенте коммерческих Web-сайтов стала мэйнстримом. Множество коммерческих сайтов встраивают структурированные данные в свои html-страницы при помощи таких форматов как RDFa и JSON-LD и таких словарей как schema.org и GoodRelations, рассчитывая на улучшение видимости их сайтов. В настоящее время более 540 миллионов HTML страниц имеют встроенные структурированные описания данных. На Рис. 1 показан график, отражающий рост использования классов словаря schema.org коммерческими Интернет-сайтами для публикации структурированных данных о товарах и услугах [3].

Таблица 1 [2]. Объемы общеизвестных графов знаний

Название	Кол-во экземпляров	Кол-во фактов	Кол-во типов	Кол-во отношений
DBpedia (English)	4 806 150	176 043 129	735	2 813
YAGO	4 595 906	25 946 870	488 469	77
Freebase	49 947 845	3 041 722 635	26 507	37 781
Wikidata	15 602 060	65 993 797	23 157	1,673
NELL	2 006 896	432 845	285	425

[2] http://googleblog.blogspot.co.uk/2012/05/introducing-knowledge-graph-things-not.html
[3] http://blogs.bing.com/search/2013/03/21/understand-your-world-with-bing/

OpenCyc	118 499	2 413 894	45 153	18 526
Google's Knowledge Graph	570 000 000	18 000 000 000	1 500	35 000
Google's Knowledge Vault	45 000 000	271 000 000	1 100	4 469
Yahoo! Knowledge Graph	3 443 743	1 391 054 990	250	800

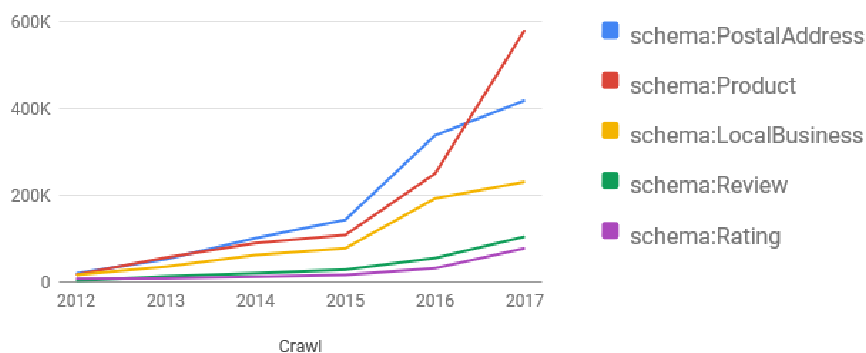

Рис. 1. Рост использования классов словаря schema.org коммерческими Интернет-сайтами для публикации структурированных данных о товарах и услугах [3]

Еще одним немаловажным аспектом является то, что направление Semantic Web предлагает действенные подходы к решению проблемы разнородности данных, являющейся одним из современных вызовов направления Big Data.

Все выше перечисленные факты делают более важным и нужным знакомство с этим направлением современных ИТ-специалистов. В Новосибирском университете на ММФ и Факультете Информационных Технологий читается спецкурс «Принципы, методы и средства

связывания данных в приложениях Semantic Web»[4], предназначенный для магистрантов, специализирующихся в области разработки программного обеспечения. В данной работе обсуждаются как важные аспекты эволюции направления Semantic Web, так и методы преподавания дисциплины «Принципы, методы и средства связывания данных в приложениях Semantic Web».

Работа выполнена при финансовой поддержке РАН (проект № 15/10).

2. Знакомство с принципами связанных данных на примере существующих наборов данных

Идея о том, что Интернет – это не только связанные документы, но и связанные данные, была высказана Тимом Бернесом Ли еще в 1994 году[5]. Затем в 2001 году появился термин «Semantic Web» [4], который обозначал расширение Всемирной паутины, позволяющее людям использовать общий контент, преодолевая границы отдельных приложений и веб-сайтов. Первоначальная идея состояла в том, что люди публикуют данные в Интернете, онтологии используются для того, чтобы все используемые термины понимались одинаково, а затем люди создают интеллектуальные приложения на основе доступных данных.

Реальный количественный скачок в развитии этого направления начался в 2006 году, когда были сформулированы принципы связанных данных[6], и началась деятельность, направленная на создание Облака Связанных данных. Поэтому знакомство с топологией Облака связанных данных является первым шагом при знакомстве с Semantic Web. Последняя визуализация этого набора, сделанная в феврале 2017 года, находится по адресу http://lod-cloud.net/. В облако входят наборы данных, разбитые по следующим тематикам: кросс-доменные, такие как DBPedia и Wikidata, географические (Geonames), библиотечные (Europeana, Worldcat, VIAF), наборы, посвященные наукам о жизни (Bio2RDF, Uniprot), лингвистические (BabelNet), средства массовой информации (BBC, New York Times), социальные сети. Хотя правительственные данные и упоминаются в этом облаке, основная информация о них находится на отдельных порталах. С момента своего создания в 2007 году, Облако Связанных данных значительно расширилось, и в настоящее время, его элементы рассредоточены по нескольким каталогам, таким как Datahub.io, publicdata.eu, data.gov, open.canada.ca. Все эти наборы данных используют CKAN платформу с открытым исходным кодом, которая является по факту стандартом для Открытых данных. Имеется

[4] http://fit.nsu.ru/data_/docs/mag/program/2013-2015/SWeb.pdf.
[5] http://www.w3.org/Talks/WWW94Tim/
[6] http://www.w3.org/DesignIssues/LinkedData.html

специальный набор данных LODStats [5], который поддерживает статистику о текущем состоянии облака данных.

Все наборы облака LOD созданы в соответствии с базовыми принципами Linked Data:

использовать URIs для определения сущностей;

- использовать HTTP URIs таким образом, чтобы на эти сущности можно было ссылаться и чтобы они могли быть найденными человеком и программным клиентом;
- при разыменовании URI, предоставлять полезную информацию о соответствующей сущности, используя такие стандарты, как RDF и SPARQL;
- при публиковании данных в веб включать в описание ссылки на другие наборы данных.

Наборы данных, удовлетворяющие всем принципам связанных данных, соответствуют пятизвездочной модели связанных данных[7].

В контексте изучения принципов Связанных Данных очень важным является осознание того, что URI (IRI) являются, прежде всего, инструментом *именования* (создания уникальных глобальных идентификаторов) как для информационных объектов, таких как HTML-страницы, так и для объектов реального мира. Эти объекты реального мира могут иметь описания как в человеко-читаемом формате, таком как страницы HTML, так и в формате, понятном компьютеру (различные синтаксические формы RDF), и что выбор подходящего описания объекта реального мира осуществляется при помощи такой процедуры, как *обсуждение контента*. Все эти понятия легко продемонстрировать на примере произвольного набора данных, хранящегося в облаке связанных данных. Например, объект реального мира Москва имеет URI <dbpedia.org/resource/Moscow> в наборе данных Dbpedia.org, html-страница этого ресурса имеет URI <http://dbpedia.org/page/Russia>, а URI <http://dbpedia.org/data/Moscow> соответствует файлу в формате RDF/XML. Получение наиболее релевантного ресурса осуществляется при помощи механизма обсуждения контента. Поэтому одно из первых практических заданий предлагаемых студентам, состоит в самостоятельном ознакомлении с произвольным набором, входящим в Облако Связанных данных, и составлении краткого отчета, отвечающего на такие вопросы как: По каким правилам формируются URI данного набора данных? Какая онтология используется для структурирования этого набора данных? С какими наборами данных связан данный набор?

[7] http://5stardata.info

Какие способы доступа имеются к данному набору данных? В соответствии с какими лицензиями он используется? И т.д.

3. Знакомство с моделью данных RDF

Модель данных RDF является стандартом, разработанным W3C, и предназначена для интегрированного представления информации, которая происходит из нескольких источников и гетерогенно структурирована (представлена с использованием различных схем).

В этой модели граф RDF является множеством триплетов вида <subject, predicate, object>, где субъект и предикат всегда являются URI, а объект может быть как URI, так и литералом (строкой). При этом предикат указывает на отношение, существующее между субъектом и объектом. Благодаря тому, что URI является глобальным уникальным идентификатором, отдельные триплеты с одинаковыми субъектами могут «склеиваться» образуя ориентированный граф с помеченными ребрами. Условно, предикаты, используемые для описания связанных данных можно разбить на три группы:

Предикаты отношений используются для связывания сущностей из разных наборов данных. Именно благодаря этим предикатам осуществляется связь между наборами, разнесенными по разным адресам. Например, следующий триплет представляет факт, состоящий в том, что Михаил Васильевич Ломоносов является автором полного собрания сочинений:

<http://www.worldcat.org/oclc/2899645> *schema:creator* <http://viaf.org/viaf/46896023>.

При этом URI субъекта этого триплета принадлежит пространству имен набора данных Worldcat, URI объекта принадлежит пространству имен набора данных VIAF, а предикат schema: creator описан в словаре schema.org.

Предикаты идентичности на уровне экземпляра указывают на URI, используемые другими источниками данных для идентификации одних и тех же объектов реального мира или абстрактных понятий (*owl:sameAs* и *rdfs:seeAlso*). Свойство *owl:sameAs* используется для выражения того, что два URI соответствуют одному и тому же объекту реального мира. Например, следующий триплет утверждает, что субъект и объект данного триплета соответствуют одному и тому же объекту реального мира:

<http://www.geonames.org/524894> *owl:sameAs* <http://dbpedia.org/resource/Moscow>. Свойство *rdfs:seeAlso* показывает, что более актуальную информацию про данный субъект можно найти, пройдя по URI объекта триплета. Автоматическое создание триплетов этого типа называется «установлением идентичности сущностей». Linked Data опирается на

решение проблемы идентичности сущностей в эволюционной и распределенной манере. Эволюционность состоит в том, что со временем создается все больше связей *owl:sameAs*, а распределенность связана с тем, что этим могут заниматься параллельно много поставщиков и потребителей данных.

Словарные предикаты связывают данные с описаниями словарных терминов, которые используются для представления данных, а также эти определения с определениями связанных терминов в других словарях. Словарные связи делают данные само-описываемыми и позволяют приложениям Linked Data понимать и интегрировать данные, описанные при помощи разных словарей. Например, предикат *rdf:type* позволяет описать принадлежность ресурса классу выбранного словаря (онтологии). Стало быть, триплет <http://dbpedia.org/resource/Moscow> *rdf:type dbo:Place* связывает сущность Moscow из набора данных dbpedia.org с классом *dbo:Place*, описанным в онтологии http://dbpedia.org/ontology. Имеется также большая группа предикатов, предназначенная для описания связей между классами и свойствами разных словарей (онтологий). Свойства из словаря RDFS *rdfs:subPropertyOf* и *rdfs:subClassOf* могут быть использованы, для того чтобы декларировать отношения между двумя свойствами или двумя классами из разных словарей. Словарь OWL предоставляет предикаты *owl:equivalentClass* и *owl:equivalentProperty* для утверждений, что два класса или два свойства, имеют один и тот же смысл.

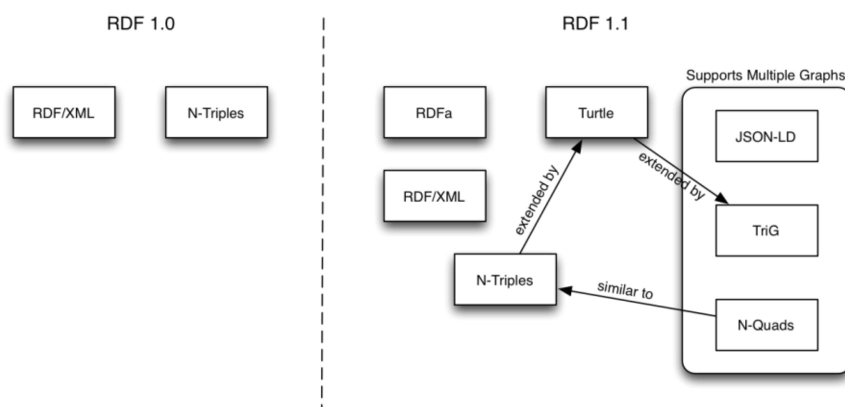

Рис. 2. Увеличение множества синтаксических форм для модели данных RDF1.1 по сравнению с моделью RDF1.0 [6]

Чтобы опубликовать граф RDF в Интернете, он сначала должен быть сериализован при помощи одной из синтаксических форм RDF. Следует заметить, что развитие проекта по созданию и практическому использованию облака Связанных данных оказало большое влияние на разработку новых версий всех основных форматов, используемых в Semantic

Web. В 2012 году появился OWL2, в 2013 году появился SPARQL 1.1, а в 2014 году – RDF 1.1. На Рис 2. показаны синтаксические формы RDF 1.0 и RDF 1.1. Одним из существенных отличий RDF 1.1 по сравнению с RDF 1.0 является появление понятия *RDF Dataset*, определяемого как множество RDF-графов. Помимо таких форм как RDF/XML, Turtle и N-Triples, появились три синтаксические формы, ориентированные на описание нескольких именованных графов Trig, N-Quads и JSON-LD. Также форматы RDFa и JSON-LD позволяют встраивать структурированные данные в HTML- страницы. Поскольку вновь появившиеся форматы активно используются в реальных приложениях Semantic Web, в курсе демонстрируются все указанные варианты синтаксических форм [7], а также осуществляется знакомство с инструментами, позволяющими конвертацию данных между этими представлениями. Помимо этого, рассматриваются такие понятия модели RDF, как пустые узлы, типизированные литералы, абсолютный и относительный URI, идентификатор фрагмента, понятие базы и т.д. Все основные примеры рассматриваются параллельно в форматах RDF/XML и Turtle.

Задание, позволяющее быстро познакомиться с простейшими этапами создания Связанных данных, выглядит следующим образом. Студентам предлагается ознакомиться с основными классами и свойствами словаря FOAF (Friend Of A Friend)[8], а затем создать описание собственной персоны в формате RDF/XML при помощи приложения FOAF-a-matic[9]. После этого они должны вручную добавить к сгенерированному файлу в формате RDF/XML пять синтаксически правильных триплетов, использующих в качестве предикатов различные свойства, описанные в разных наборах данных облака LOD. Студенты должны проверить синтаксическую правильность полученного файла при помощи валидатора RDF[10].

После знакомства с основными понятиями RDF проводится вводная лекция по SPARQL, знакомящая с его основными конструкциями. Эта вводная часть воспринимается студентами достаточно легко, поскольку SPARQL имеет много схожих черт с SQL. На последующих занятиях теоретические вопросы Связанных данных рассматриваются параллельно с углубленным изучением SPARQL 1.1. и использованием запросов SPARQL для ознакомления с понятиями Связанных данных. В частности, знакомство с форматами RDFS и OWL сопровождается знакомством с классами и свойствами конкретных словарей, описанными в облаке связанных данных. Их исследование базируется на запросах SPARQL 1.1.

---

[8] http://xmlns.com/foaf/spec/
[9] http://www.ldodds.com/foaf/foaf-a-matic.html
[10] http://www.w3.org/RDF/Validator/

4. Языки описания словарей RDFS и OWL и словари, используемые в облаке связанных данных

RDF Schema (RDFS) и язык описания онтологий Web Ontology Language (OWL) являются наиболее известными языками описания словарей (онтологий) в области Semantic Web. RDFS – это минимальный язык описания онтологии, построенный поверх RDF, который позволяет определить:

- классы индивидуальных ресурсов;
- свойства, связывающие два ресурса;
- иерархии классов;
- иерархии свойств;
- ограничения на область определения и область значений свойств (*rdfs:domain*, *rdfs:range*).

OWL обеспечивает разработчиков онтологий гораздо более сложными и полезными конструкциями, чем RDFS, благодаря чему популярность OWL постоянно растет. Как и в RDFS, основными элементами OWL являются классы, свойства и индивиды, которые являются членами классов. Свойства OWL являются бинарными отношениями, среди них принято выделять *owl:ObjectProperty* и *owl:DataTypeProperty*. Свойства типа *owl:ObjectProperty* связывают двух индивидов, тогда как *owl:DataTypeProperty* связывают индивида с литералом.

В курсе по Semantic Web делается акцент на том, что стандарты RDFS и OWL, связаны между собой, что основной синтаксической формой OWL2 является RDF/XML, и, стало быть, к схемам, описанным в форматах rdfs и owl, можно писать запросы SPARQL. Например, в том, что *owl:Class* является подклассом *rdfs:Class*, а *owl:DatatypeProperty* и *owl:ObjectProperty* являются подклассами *rdf:Property* легко убедиться, при помощи простого запроса SPARQL:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:< http://www.w3.org/2000/01/rdf-schema#>
SELECT ?s ?o
FROM <http://www.w3.org/2002/07/owl>
WHERE   {?s rdfs:subClassOf ?o }
```

Одним из важнейших свойств Semantic Web является возможность интегрировать данные из различных источников, описанных при помощи разных словарей (схем, онтологий). Словарь состоит из классов, свойств и типов данных, которые определяют смысл данных. RDF словари сами выражаются и публикуются в соответствии с принципами

связанных данных. В настоящее время в Интернете опубликовано значительное количество словарей и онтологий. Например, набор данных LODStats дает информацию о 2593 словарях, имеющихся в Облаке Связанных данных[11]. Web of Data применяет двойственный подход к такой разнородности данных. Во-первых, рекомендуется уменьшать уровень разнородности за счет использования терминов из широко используемых словарей. Во-вторых, в случае использования проприетарных терминов, они должны быть как можно более *само-описываемыми*, то есть каждый словарный термин должен быть связан со своим описанием. Кроме этого, рекомендуется публиковать файлы соответствий между терминами из разных словарей в виде RDF.

Таким образом, словари являются важной частью облака связанных данных, и знакомство с наиболее популярными словарями необходимо в рамках курса по Semantic Web [8]. Для того чтобы иметь возможность повторно использовать имеющиеся словари, надо знать, где их найти. Большое количество словарей может быть обнаружено при помощи поискового движка Watson[12]. Более 628 различных словарей (данные указаны на январь 2018 года) представлены в специализированном наборе данных Linked Open Vocabularies (LOV) [9].

Набор данных LOV строит экосистему словарей, поддерживающую их повторное использование. Он подсчитывает популярность каждого термина словаря, а также устанавливает отношения между словарями, используя словарь VOAF. На уровне словарей LOV извлекает три типа информации для каждой версии словаря: метаданные, входные связи/входные словари, выходные связи/выходные словари, связанные с каждым словарем. Метаинформация, определенная внутри словаря, предоставляет контекст и полезные данные о словаре. Для этого обычно повторно используются такие словари, как DublinCore[13], описывающий создателей, публикаторов, и других персон, внесших вклад в создание и развитие словаря, CreativeCommons[14], описывающий лицензии словаря и др. Входные связи позволяют в явном виде указать, что термины данного словаря ссылаются на термины других словарей. Аналогичным образом выходные связи позволяют в явном виде указать, что термины других словарей ссылаются на термины данного словаря.

В словаре VOAF введены такие типы отношений между словарями как *voaf:metadata*, *voaf:specializes*, *voaf:extends*, *voaf:hasEquivalencesWith*, *voaf:generalizes*,

---

[11] http://stats.lod2.eu/vocabularies
[12] http://watson.kmi.open.ac.uk/WatsonWUI/
[13] http://dublincore.org/documents/dc-rdf/
[14] https://creativecommons.org/ns

voaf:hasDisjunctionsWith. Также используется отношение импорта между словарями *owl:import*. На Рис. 3 показано изображение входных и выходных связей словаря schema.org с другими словарями LOV. Такое изображение генерируется автоматически для каждого словаря, включенного в LOV, при помощи запросов SPARQL.

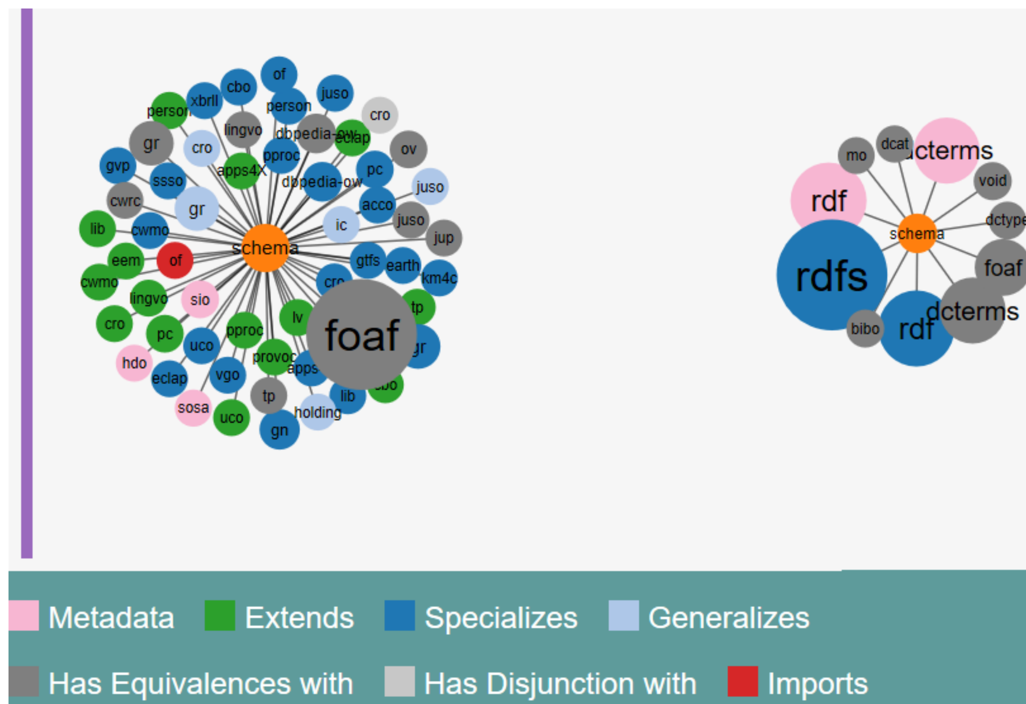

Рис. 3. Связи словаря schema.org с другими словарями LOV [9].

Для того чтобы существующий словарь был включен в набор данных LOV, должны выполняться следующие требования:

— словарь должен быть написан на RDF (RDFS, OWL/RDF) и быть разыменовываемым;

— словарь должен быть синтаксически правильным;

— все словарные термины (классы, свойства и типы данных) словаря должны иметь свойство *rdfs: label;*

— словарь должен ссылаться и повторно использовать релевантные существующие словари;

— словарь должен предоставлять некоторые метаданные о самом словаре (по крайней мере, название).

Для того чтобы поддержать повторное использование словарей, LOV поддерживает поиск словарей на основе терминов (класс, свойство, тип данных) или предметной области. Он ранжирует все термины входящих в него словарей на основе количества использований каждого термина в наборах данных облака LOD. Так самыми популярными терминами являются *rdf:type* (рейтинг1.000), *rdfs:label* (рейтинг1.000), *owl:sameAs* (рейтинг 0.5401).

Помимо того, что LOV является весьма полезным вспомогательным средством при поиске онтологий, он является также прекрасным инструментом для использования в образовательных целях. В частности, благодаря имеющейся конечной точке SPARQL, можно знакомить учащихся со многими вопросами устройства онтологий, как это будет показано в разделе 6.

Большим достоинством LOV является то, что он хранит локальные копии онтологий, с которыми можно знакомиться разными способами. Так, например, страничка LOV, посвященная словарю schema.org, указывает локальное URI этого словаря (schema.org). Но для того, чтобы получить описание этого словаря в формате RDFS, необходимо извлечь его из RDFa описания HTML страницы[15] при помощи инструмента RDFa Distiller and Parser[16]. Все словари, знакомство с которыми осуществляется в данном курсе (VOID, Dublin Core, FOAF, goodRelations, schema.org, dbpedia.org/ontology, owl, rdfs), присутствуют в наборе данных LOV.

5. Запросы SPARQL – важный инструмент работы с наборами данных

Запросы SPARQL [10] позволяют не только получить исчерпывающую информацию о любом незнакомом наборе данных, но также копировать данные, создавать новые данные, конвертировать данные, осуществлять контроль качества данных, не только проверяя выполнение таких ограничений как корректность типов данных, но и соответствие бизнес правилам.

При знакомстве с новым набором данных, прежде всего, надо понять, что это за данные. Самый первый запрос SPARQL, который следует задать к незнакомому набору данных это

SELECT * WHERE { ?s ?p ?o . }#LIMIT 50

Поскольку это запрос к графу по умолчанию, а многие хранилища триплетов хранят данные в именованных графах, вторым уместным запросом будет запрос, который покажет URI всех именованных графов, входящих в данный набор:

SELECT DISTINCT ?g WHERE { GRAPH ?g {?s ?p ?o } }

В частности, при запуске этого запроса на конечной точке SPARQL словаря LOV выдается список всех словарей (онтологий) имеющихся в настоящий момент в этом наборе данных, а также URI всех словарей, по которым можно строить запросы SPARQL к любому

---

[15] https://schema.org/docs/schema_org_rdfa_html

[16] http://www.w3.org/2007/08/pyRdfa

из описанных словарей. На примере этого набора данных достаточно просто проиллюстрировать запросами SPARQL занятия, посвященные именованным графам, языкам описания RDFS и OWL. Например, запуск следующего запроса на конечной точке http://lov.okfn.org/dataset/lov/sparql позволяет студентам быстро почувствовать разницу в доступе к именованным графам и графам по умолчанию:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?g ?s  ?o
FROM  NAMED <http://www.w3.org/ns/mls>
FROM  <http://purl.org/vocab/aiiso/schema>
WHERE {{?s rdfs:label ?o.}
UNION
{GRAPH ?g {?s rdf:type ?o}}
}
```

Также, в курсе рассматриваются такие типы запросов, как: «Какие классы декларированы в данном наборе данных? Сколько экземпляров имеется у каждого класса заданного набора данных? Какие классы используют определенное свойство? Какие свойства декларированы? Какие значения имеет данное свойство? Какие классы используются?». В контексте RDF вопрос про то, какие классы используются, следует понимать как синоним вопроса « у какого класса имеются экземпляры»?

Следует отметить, что как классы, так и свойства в словарях часто декларируются неявно, как подклассы или подсвойства других классов. Поэтому для полного ответа на вопрос обо всех декларированных классах необходимо проверить также наличие триплетов, имеющих в качестве предиката *rdfs:subClassOf*, а также поискать транзитивное замыкание по этому предикату (rdfs:subClassOf+).

Таким образом, все статистические данные, которые обычно сопровождают набор данных в облаке связанных данных, можно вычислить самостоятельно, грамотно пользуясь запросами SPARQL. Ответы на все выше перечисленные базовые вопросы студенты учатся находить для произвольного набора данных, который они видят первый раз в жизни.

Помимо вопросов, направленных на исследование незнакомого набора данных, имеется большая группа запросов, предназначенных для работы с наборами данных на протяжении всего их жизненного цикла. Поэтому в режиме практических занятий рассматриваются следующие вопросы:

— как при помощи запроса CONSTRUCT объединить данные из нескольких разрозненных наборов данных,
— как сконструировать новые данные на основе имеющихся,
— как преобразовать данные, описанные при помощи одной онтологии в данные, основанные на другой онтологии,
— как при помощи запросов SPARQL можно автоматизировать решение задачи идентификации сущностей в разных наборах данных (генерацию отношений *owl:sameAs*),
— как устанавливать соответствие между классами и свойствами разных онтологий.

Опыт показал, что благодаря знакомству с языком SQL, студенты легко справляются с формой запроса SELECT, при этом у них возникают проблемы с применением таких форм, как ASK и CONSTRUCT. Поэтому данные формы запросов требуют большего количества упражнений. Еще один вопрос, требующий внимательного рассмотрения, связан с построением запросов к отдельным файлам при помощи ключевого слова FROM или же доступа к удаленной конечной точке при помощи ключевого слова SERVICE. Наконец, в режиме практических занятий следует явно показывать, что запросы SPARQL будут формулироваться по-разному, в зависимости от типа конечной точки SPARQL (специализированной, такой как dbedia.org/sparql, или же точки произвольного доступа, такой как http://uriburner.com/sparql).

6. Жизненный цикл связанных данных и инструменты, используемые на разных этапах жизненного цикла.

Процесс предоставления связанных данных часто характеризуется как *жизненный цикл*, в котором данные создаются, повторно используя другие источники данных, преобразуются, а затем делаются доступными для использования. Вновь созданные данные могут стать одним из ряда источников данных, используемых при подготовке дальнейших данных. Таким образом, создание и публикация связанных данных имеет циклический характер. Известно несколько разных моделей жизненного цикла связанных данных, наиболее подробная модель жизненного цикла [11] включает такие этапы как извлечение данных, хранение и поддержка запросов, авторская разработка, связывание данных, семантическое обогащение, анализ качества, эволюция и исправление ошибок, поиск, просмотр и разведка. В упрощенной формулировке жизненного цикла связанных данных можно выделить три основные стадии. Эти стадии также могут быть сопоставлены с принципами связанных данных.

Создание связанных данных: Извлечение данных, создание HTTP URI в качестве имен, и выборе словарей для описания предикатов и для классификации сущностей. Этот этап относится к принципам связанных данных 1 и 2, поскольку предполагает нахождение или создание HTTP URI имен для вещей.

Связывание данных: Поиск и выражение связей между сущностями-синонимами из разных наборов данных. Этот этап относится к принципу связанных данных 4, так как предполагает обеспечение ссылок на другие сущности.

Публикация связанных данных: Создание метаданных о наборе данных и обеспечение доступа к набору данных. Относится к принципу связанных данных 3 в том, что обеспечивает возвращение полезной информации о наборе данных.

Инструменты, применяемые на каждой стадии жизненного цикла, быстро эволюционируют. Каждый год появляются новые инструменты, а некоторые инструменты устаревают и перестают использоваться. Поэтому эта часть курса нуждается в ежегодном обновлении. Тем не менее, есть некоторые базовые инструменты, знакомство с которыми важно для понимания курса.

Наиболее распространенные форматы, на основе которых осуществляется генерация данных RDF – это таблицы или табличные данные, реляционные базы данных и текстовые данные. Для разных видов данных существуют различные стратегии и инструменты, позволяющие преобразовать их в RDF. Реляционные БД обычно отображаются в RDF при помощи правил установления соответствия и инструментов, таких как D2R [12]. В этих случаях правила отображения пишутся вручную, что весьма несложно, так как схема реляционной БД обычно явно задана. В рамках данного курса, студенты знакомятся с языком R2RML, который является рекомендацией W3C для определения отображения между реляционными базами данных и связанными данными. Он позволяет определить шаблоны, по которым названия таблиц отображаются в названия классов выбранной онтологии, первичные ключи таблицы отображаются в глобальные URI в выбранном пользователем пространстве имен, а названия столбцов таблиц отображаются в предикаты выбранной пользователем онтологии. Полу-структурированные данные, такие как веб – таблицы, обычно существуют в больших количествах, но без явной семантики. Их автоматическое преобразование в RDF является не простой задачей [13]. В рамках данного курса, студенты знакомятся с простым инструментом OpenRefine[17], который позволяет решать эту задачу в интерактивном режиме. Ряд инструментов существует для поддержки извлечения данных из

[17] http://openrefine.org/

свободного текста, включая Open Calais[18] и DBpedia Spotlight [14]. На этапе связывания данных решается задача установления связей идентичности между отдельными индивидами, а также установления словарных связей между свойствами и классами, описанных в разных онтологиях словарях. Для автоматизированного решения первой задачи часто используется инструмент SILK [15], с которым осуществляется знакомство в данном курсе. Дополнительно, в курсе изучается вопрос, как триплеты с предикатом *owl:sameAs* могут быть созданы в результате логического вывода. Генерация триплетов с предикатом *owl:sameAs* при помощи запросов SPARQL рассматривается на практических занятиях. Что касается задачи создания словарных связей, то автоматическое решение этой задачи остается активной областью исследования, знакомиться с которой более уместно в форме научных семинаров[19]. В рамках данного курса демонстрируется, как можно решать эту задачу при помощи запросов SPARQL.

После того, как наборы данных RDF был созданы и взаимосвязаны, процесс публикации включает следующие задачи:

1. Создание метаданных для описания набора данных;
2. Предоставление доступа к набору данных;
3. Валидация набора данных.

Метаданные создаются в соответствии с такими словарями как VOID и Dublin Core. Оба эти словаря представлены в наборе данных LOV. В качестве основной формы предоставления связанных рассматриваются каталог DataHub, основанный на платформе с открытым исходным кодом CKAN, а также хранилища триплетов, такие как Jena[20] и Virtuoso[21].

В качестве инструментов валидации данных можно использовать RDF Triple-Checker[22], который помогает находить опечатки и распространенные ошибки в данных RDF, OntOlogy Pitfall Scanner![23] , который помогает обнаружить некоторые наиболее распространенные ошибки, возникающие при разработке онтологий, а также инструмент RDFAlerts[24].

7. Приложения Связанных данных

[18] http://www.opencalais.com/opencalais-demo/
[19] http://oaei.ontologymatching.org/
[20] jena.apache.org
[21] https://virtuoso.openlinksw.com/
[22] http://graphite.ecs.soton.ac.uk/checker/
[23]http://oops.linkeddata.es/ OOPS!
[24] http://swse.deri.org/RDFAlerts/

Приложения связанных данных являются тем, что подтверждает ценность этого направления. Среди существующих приложений можно выделить следующие три группы.

1) *Навигаторы Связанных данных.* Разыменовывают URI, чтобы получить описание ресурса. Типичным примером навигатора Связанных данных являются DBPedia.org.

2) *Поисковые системы связанных данных.* Позволяют посылать запросы к связанным данным. В отличие от обычных поисковых систем, которые в основном рассматривается как средство для локализации человеко-читаемого контента, семантическая поисковая система используется для поиска онтологий, словарей и документов RDF. К таким системам относятся Watson и LOV. Системы Семантического поиска, встроенные в поисковые системы Google и Bing, опираются на внутренние Графы Знаний и позволяют помимо поиска по ключевым словам выдавать дополнительную информацию о сущностях, отображенных на эти Графы Знаний. Поисковые системы, основанные на Графах Знаний, все чаще используются в промышленности. Появляется все больше коммерческих приложений, использующих интеграцию данных о продуктах, услугах, предложениях работы и т.д. [16, 17].

3) *Приложения Связанных данных*, ориентированные на конкретную предметную область. Эти приложения создаются для решения конкретного круга проблем в указанном домене. Подавляющее большинство приложений Связанных данных попадают в эту третью категорию.

Примером приложения семантических технологий являются многочисленные разделы портала bbc.com. Использование семантических веб-технологий не видно пользователю, который взаимодействует с порталом как с обычным веб-сайтом. Для управления контентом этого сайта реализована специальная архитектура, которая называется «Динамическая Семантическая Публикация» и направлена на автоматизацию агрегации или публикацию взаимосвязанного контента в рамках портала BBC[25].

Еще одно интересное приложение, которое рассматривается в курсе - Open Pharmacology Space[26] - семантическая исследовательская среда для фармакологии. Исследование и разработка новых лекарств требует от ученых извлечения знаний из множественных источников информации от баз данных белков и химических соединений до моделей биологических путей. Интеграция данных в таких системах является серьезной проблемой. Например, источник ChemSpider содержит информацию о химических соединениях, где они

---

[25] http://www.bbc.co.uk/blogs/bbcinternet/2012/04/sports_dynamic_semantic.html

[26] http://www.openphacts.org/open-phacts-discovery-platform

были получены, CheMBL дополняет эту информацию данными о биоактивности молекул, похожих на лекарства, Drugbank дает информацию о клиническом использовании лекарств, содержащих указанную молекулу. Поскольку эти источники по-разному подходят к представлению структуры молекулы, они могут выдавать разные соединения для одного и того же химического препарата. Open Pharmacology Space [17] идентифицирует сущности, описанные в разных источниках данных, и увязывает их между собой.

Еще один интересный проект GRAVITATE[27] направлен на создание инструментов, позволяющих археологам реконструировать разрушенные культурные объекты, части которых хранятся в разных коллекциях. Помимо примеров конкретных приложений, в курсе рассматривается также архитектура приложений, работающих со Связанными данными.

8. Заключение

В данной работе кратко представлены основные части курса, который все еще находится в процессе развития, поскольку в процессе развития находится научное направление Semantic Web. С одной стороны, данный аспект повышает актуальность представляемого курса, с другой стороны, создает трудности в его преподавании, поскольку каждый год курс приходится обновлять. Наблюдается процесс интеграции знаний из Облака Открытых Связанных данных с данными, встроенными в html- страницы, методы машинного обучения используются для улучшения качества данных Semantic Web, и одновременно методы Semantic Web используются для поддержки открытия новых знаний [18]. Это значит, что потребность в специалистах, способных работать в этом направлении, будет и дальше возрастать.

Список литературы

1. Smith-Yoshimura K. Analysis of International Linked Data Survey for Implementers// D-Lib Magazine. July/August 2016. Vol. 22, №7/8.
2. Paulheim H., Automatic Knowledge Graph Refinement: A Survey of Approaches and Evaluation Methods// Semantic Web. 2016.
3. Bizer Ch., Meusel R., Primpeli A. Web Data Commons - RDFa, Microdata, and Microformat Data Sets, 2018 http://webdatacommons.org/structureddata/
4. Berners-Lee T., Hendler J., Lassila O. The Semantic Web: Scientific American: Feature Article May 2001

---

[27] http://gravitate-project.eu/

5. Ermilov I., Lehmann J., Martin M., Auer S. LODStats: The Data Web Census Dataset //ISWC 2016, pp. 38-46.
6. Wood D., 3 Round Stones Inc. What's New in RDF 1.1 URL:http://www.w3.org/TR/rdf11-new/ (дата обращения 05.01.2018)
7. Schreiber G., Raimond Y., RDF 1.1 Primer, W3C Working Group Note 24 June 2014 URL: http://www.w3.org/TR/rdf11-primer/ (дата обращения 05.01.2018)
8. d'Aquin M., Noy N. F. Where to publish and find ontologies? a survey of ontology libraries// Web Semantics:Science, Services and Agents on the World Wide Web, 2012. P. 96 – 111.
9. Vandenbusschea P-Y, Atemezingb G. A., Poveda-Villalónc M., Vatantd B., Linked Open Vocabularies (LOV): a gateway to reusable semantic vocabularies on the Web// Semantic Web, 2014. P. 1–5.
10. DuCharme B. Learning SPARQL, Second Edition, O'Reilly Media, Inc. URL http://it-ebooks.info/book/2574/ (дата обращения 25.12.2017).
11. Auer S., Lehmann J., A-C. Ngonga Ngomo, Zaveri A.Introduction to Linked Data and its Lifecycle on the Web// Reasoning Web. Semantic Technologies for the Web of Data - 7th International Summer School 2011, Galway, Ireland, August 23-27, 2011, Tutorial Lectures http://www.jens-lehmann.org/files/2013/reasoning_web_linked_data.pdf (дата обращения 05.01.2018).
12. Spanos D.-E., Stavrou P., Mitrou N., Bringing relational databases into the semantic web: A survey, Semant. Web. 2012. Vol. 3, №2. P. 169–209. http://www.semantic-web-journal.net/sites/default/files/swj121_1.pdf (дата обращения 05.01.2018).
13. Mulwad V., Finin, T. Joshi A., Semantic message passing for generating linked data from tables, in: Proceedings of the 12th International Semantic Web Conference, Springer, 2013.
14. Mendes P.N., Jakob M., García-Silva A., Bizer C., Dbpedia spotlight: Shedding light on the web of documents// Proceedings of the 7th International Conference on Semantic Systems, I-Semantics'11, ACM, New York, NY,USA, 2011, P. 1–8.
15. Isele R., Jentzsch A., Bizer Ch. Silk Server - Adding missing Links while consuming Linked Data// 1st International Workshop on Consuming Linked Data (COLD 2010), Shanghai, November 2010.
16. Ristoski P., Mika P., Enriching Product Ads with Metadata from HTML Annotations// ESWC 2016, LNCS 9678, 2016. P. 151–167.
17. Gray A. J.G., Groth P., Loizou A., Askjaer S., Brenninkmeijer C., Burger K., Chichester C., Evelo C. T., Goble C., Harland L., Pettifer S., Thompson M., Waagmeester A., Williams A. J. Applying linked data approaches to pharmacology: Architectural decisions and implementation//Semantic Web . 2014. P. 101–113.
18. Ristoski P. Paulheim H. SemanticWeb in data mining and knowledge discovery: A comprehensive survey//Web Semantics: Science, Services and Agents on theWorldWideWeb. 2016. P.1–22.

УДК 004.43

Реализация оптимизирующих трансформаций в системе предикатного программирования

Каблуков И.В. (Институт систем информатики СО РАН),

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описываются оптимизирующие трансформации склеивания переменных, замены хвостовой рекурсии циклом, открытой подстановки, упрощения и оформления. Результатом трансформаций является эффективная императивная программа. Используется потоковый анализ, включающий построение графа вызовов и определение области жизни переменных программы.

Ключевые слова: *функциональное программирование, трансформации программ, потоковый анализ, склеивание переменных, хвостовая рекурсия, открытая подстановка.*

1. Введение

Программа на языке предикатного программирования P [6] является предикатом, описывающим алгоритм решения математической задачи в форме исполняемого оператора. Имея общие особенности с функциональным и императивным программированием, предикатное программирование, тем не менее, принципиально отличается от них.

Эффективность предикатных программ достигается применением при трансляции следующих оптимизирующих трансформаций:

- склеивания переменных, реализующие замену нескольких переменных одной;
- замены хвостовой рекурсии циклом;
- подстановки определения предиката на место его вызова;
- кодирования алгебраических типов (списков, деревьев) через массивы и указатели.

Данные трансформации реализуют *оптимизацию среднего уровня* с переводом предикатной программы в эффективную императивную программу. Построение алгоритмов такой оптимизации – новая задача, характерная для предикатного, но не функционального программирования. Оптимизация среднего уровня принципиально отличается от

традиционной оптимизации для императивных языков. Трансформации склеивания переменных и кодирования объектов алгебраических типов аналогов не имеют.

Эффективность программы после применения трансформаций обеспечивается оптимизацией, реализуемой программистом при построении предикатной программы. Фактически, набор применяемых трансформаций планируется при построении программы. И задача трансформатора – «угадать», найти этот набор по программе. Иначе говоря, ставится задача разработки алгоритмов трансформаций не для произвольной программы, а для программы, ориентированной на трансформацию.

В настоящей работе описываются алгоритмы следующих трансформаций: склеивания переменных, замены хвостовой рекурсии циклом, открытой подстановки программ на места их вызовов. Реализация трансформаций использует результаты потокового анализа предикатной программы. Дополнительно осуществляются простые оптимизирующие преобразования, упрощающие программу.

Во втором разделе настоящей работы дается общее представление о системе предикатного программирования. В третьем разделе описывается структура реализуемого блока трансформаций. В последующих разделах описываются составные части этого блока. В четвертом разделе приводятся алгоритмы потокового анализа. В разделах с пятого по восьмой описываются трансформации подстановки определения предиката на место вызова, замены хвостовой рекурсии циклом и склеивания переменных, а также упрощения и оформления. В девятом разделе приводятся примеры трансформаций программ. Десятый раздел – обзор смежных работ по оптимизирующим трансформациям.

2. Система предикатного программирования

Программа на языке P [6] является предикатом, описывающим алгоритм решения математической задачи в форме исполняемого оператора.

Предикатная программа состоит из набора рекурсивных программ (определений предикатов) на языке P следующего вида:

<имя программы>(<описания аргументов>**:** <описания результатов>)

pre <предусловие>

{ <оператор> }

post <постусловие>

Необязательные конструкции предусловие и постусловие являются формулами на языке исчисления предикатов; они используются для улучшения понимания программ и для дедуктивной верификации [3, 9, 12, 20]. Ниже представлены основные конструкции языка P:

оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов и результатов предиката и локальных переменных.

<переменная> = <выражение>

{<оператор1>; <оператор2>}

<оператор1> || <оператор2>

if (<логическое выражение>) <оператор1> **else** <оператор2>

<имя программы>(<список аргументов>**:** <список результатов>)

<тип> <пробел> <список имен переменных>

В предикатном программировании запрещены такие языковые конструкции, как циклы и указатели, серьезно усложняющие программу. Вместо циклов используются рекурсивные функции, а вместо массивов и указателей – списки.

В наборе предикатов программы существует главный предикат, с которого начинается исполнение программы и который вызывает остальные предикаты по цепочкам вызовов.

Трансформации. Получение эффективной программы для функциональных языков проблематично даже с применением изощренной оптимизации в процессе трансляции. После отладки программы на функциональном языке для достижения требуемой эффективности ее приходится переписывать на императивный язык.

К предикатной программе применяется набор трансформаций с получением эффективной программы на императивном расширении языка P, после чего программа может конвертироваться на любой из императивных языков: C, C++, ФОРТРАН и др. Трансформация программ с внутреннего представления на императивное расширение происходит в 4 этапа, на каждом из которых реализуется одна из базовых трансформаций:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование алгебраических типов: списков, деревьев низкоуровневыми структурами с использованием массивов и указателей.

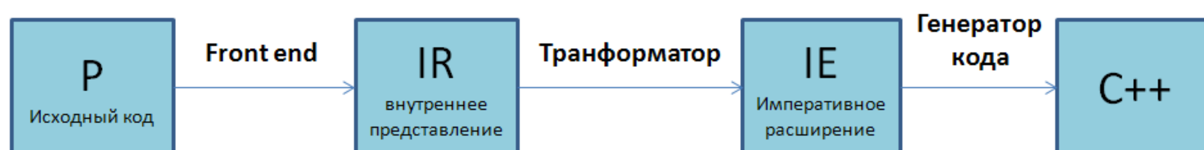

Рис. 2.1. Схема реализации языка P.

Эффективность программы после применения трансформаций обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи [3, 9, 12, 20]. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче [1, 4, 5, 8, 19]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [18]) оптимизация программы полностью возлагается на транслятор, в частности, обеспечивается автоматическое приведение рекурсии к хвостовому виду. Разумеется, функциональное программирование существенно уступает в эффективности, поскольку даже применением изощренных методов оптимизации невозможно автоматически воспроизвести серию оптимизаций, совершаемых программистом вручную.

Императивное расширение языка P определяет следующие дополнительные языковые конструкции:

break

for (<тип> <параметр цикла> = <выражение>; <условие завершения>; <пересчет параметра>) { <оператор> }

while (<выражение>) { <оператор> }

loop { <оператор> }

if (<выражение>) { <оператор> }

#<метка>

<метка>**:** <оператор>

Семантика циклов **for** и **while** соответствует языку C++. Оператор вида #<метка> реализует переход на оператор, помеченный указанной меткой.

Приведенные выше конструкции возникают в программе в результате проведения трансформаций предикатной программы. Использование этих конструкций в исходной предикатной программе недопустимо.

3. Структура блока трансформаций

Блок трансформаций реализует оптимизацию среднего уровня, преобразуя программу с языка предикатного программирования на язык императивного расширения. Производятся следующие трансформации:

- склеивание переменных;

- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование алгебраических типов.

Для проведения трансформаций требуется предварительный потоковый анализ программы, который строит граф вызовов и определяет аргументы, результаты и живые переменных операторов. После проведения трансформаций потоковая информация о программе может стать недействительной. Поэтому перед каждой трансформацией необходимая потоковая информация обновляется: перед трансформацией подстановки определения предиката граф вызовов строится заново (т.к. замена хвостовой рекурсии циклом сделала некоторые предикаты нерекурсивными, и они становятся пригодны для подстановки на место вызова), а перед трансформацией кодирования алгебраических типов обновляется информация о живых аргументах операторов (т.к. склеивание переменных изменило их время жизни). В конце после применения трансформаций реализуется серия упрощений программы.

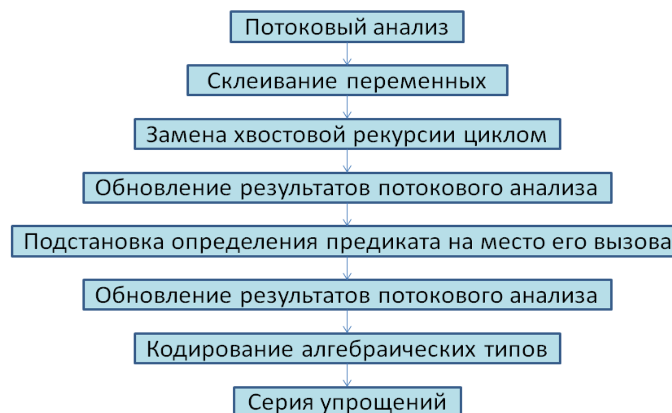

Рис. 3.1. Схема блока трансформаций.

Граф вызов предикатной программы — это ориентированный граф (орграф) с предикатами в качестве вершин. Вызов предиката A в теле предиката B определяет дугу в графе B→A.

Понятия живости переменной определяется по отношению к ее вхождению в операторе. Переменная является *живой*, если ее значение используется при исполнении программы после данного оператора.

Обоснование порядка трансформаций. Для рекурсивных предикатов нецелесообразно проводить подстановку тела на место вызова. Поэтому замена хвостовой рекурсии циклом производится перед подстановкой тел предикатов. Трансформацию склеивания можно было бы реализовать после замены хвостовой рекурсии и подстановки определения предиката.

4. Потоковый анализ

Для проведения трансформаций подстановки тела предиката на место вызова и склеивания переменных требуется построение графа вызовов программы.

Построение графа вызовов. Для каждой переменной предикатного типа находятся ее *заместители* — это предикаты, являющиеся возможными значениями переменной. Определим массив S. Его индексы — имена переменных предикатного типа, а значения – множества заместителей. Вначале набор индексов пустой. Алгоритм построения множества заместителей реализуется многократным просмотром программы. На очередном просмотре набор индексов и множества заместителей пополняются. Работа алгоритма завершается, когда на очередном просмотре программы множества заместителей останутся неизменными.

Алгоритм построения графа вызовов следующий. Именами A, B, C будут обозначаться предикаты, именами a, b, c — переменные предикатного типа. В программе ищутся вхождения имен предикатов.

Правило 1**.** Имеется вызов B(…) в теле предиката A. Тогда в граф вызовов добавляется дуга A→B.

Правило 2**.** Пусть в теле предиката A находится вызов B(…, C, …), и есть определение предиката B(…, a, …) {…}. Тогда в множество S[a] добавляется C.

Правило 3**.** Имеется присваивание a = C. Тогда в множество S[a] добавляется C.

Правило 4. Имеется вызов a(…) в теле предиката A. Тогда в граф добавляются дуги A→C для каждого C из множества S[a].

Правило 5**.** Пусть в теле предиката A находится вызов B(…, a, …: …), и есть определение предиката B(…, c, …: …) {…}. Тогда в множество S[c] добавляется множество S[a].

Правило 6**.** Имеется присваивание b = a. Тогда в множество S[b] добавляется множество S[a].

Правило 7**.** Пусть в теле предиката A находится вызов B(…**:** …, a, …), где a — результат вызова. Пусть есть определение предиката B(…**:** …, c, …) {…}. Тогда всякий заместитель c является заместителем a. Поэтому в множество S[a] добавляется множество S[c].

Компоненты сильной связности. Две вершины s и t орграфа *сильно связаны*, если существует ориентированный путь из s в t и ориентированный путь из t в s. Компонента сильной связности есть максимальное множество сильно связанных вершин. Будем называть *рекурсивным слоем* набор предикатов, чьи вершины принадлежат одной компоненте сильной связности графа вызовов. *Рекурсивный вызов* — вызов предиката из того же рекурсивного слоя, в котором находится вызывающий предикат.

Компоненты сильной связности графа вызовов находятся по алгоритму Тарьяна [7, разд. 3.5.3], который имеет линейную сложность. Алгоритм можно считать вариацией алгоритма поиска в глубину. Для каждой вершины строятся пути из нее, и если один из путей вернулся в начальную вершину, то все вершины на пути объединяются в одну компоненту связности.

Пусть в теле предиката A есть вызов B(…, c, …), где c — переменная предикатного типа со значением A. Пусть есть определение B(…, d, …) {… d(…); …}. Тогда A вызывает B, а B неявно вызывает A. Такая форма рекурсии с неявным вызовом через предикатную переменную запрещена в языке P. Для каждой дуги A→B, порожденной вызовом через переменную проверяется, что A и B не принадлежат одному рекурсивному слою.

Определение порядка предикатов. Необходимо определить порядок предикатов, в котором проводится трансформация склеивания. Склеивания, которые были произведены на аргументах и результатах – формальных параметров предиката, должны быть перенесены на аргументы и результаты, являющиеся фактическими параметрами во всех вызовах данного предиката. Поэтому склеивание в теле предиката должно быть проведено раньше, чем склеивания в теле другого предиката, содержащего вызов первого.

Граф приводится к ациклическому виду стягиванием компонент сильной связности в одну вершину. Орграф становится ациклическим, и на нем можно определить порядок с помощью алгоритма обратной топологической сортировки [7, разд. 3.1.2]. Обратный топологический порядок на орграфе это порядок, при котором для каждой дуги A→B вершина B находится раньше A. Важно отметить, что граф вызовов предикатной программы связный: существует главный предикат, с которого начинается исполнение программы и который вызывает остальные предикаты по цепочкам вызовов. Поэтому все используемые предикаты будут упорядочены.

Нахождение живых переменных операторов. Для трансформации склеивания переменных необходимо определение аргументов, результатов и живых переменных операторов в телах предикатов. Нахождение аргументов и результатов операторов происходит просмотром программы во внутреннем представлении.

Живыми (активными) аргументами оператора являются переменные, чьи значения будут использоваться после завершения исполнения этого оператора. Вхождения живых переменных нельзя склеивать с другими переменными, поскольку значения переменных используются далее. Склеиванию подлежат переменные, не являющиеся живыми. Отметим, что формальные параметры – результаты предиката являются живыми для каждого

оператора его тела, так как они предположительно будут использоваться после завершения исполнения предиката.

Описываемый ниже алгоритм является адаптацией алгоритма [1, разд. 9.2.5]. Для каждого оператора определяются преемники – операторы, исполняющиеся сразу после него. Пусть Op – произвольный исполняемый оператор предиката F. Определим множества In[Op] и Out[Op] – множества переменных, живых перед и после исполнения оператора, соответственно. Пусть args[Op] – множество аргументов, а results[Op] – множество результатов оператора Op; Exit – фиктивный пустой оператор, вставляемый после каждого последнего оператора на каждой ветви тела предиката.

Алгоритм имеет следующий вид:

In[Exit] = формальные параметры – результаты предиката F
while (внесены изменения в любое In)
    for (каждый исполняемый оператор Op предиката) {
        Out[Op] = $\bigcup_{S \text{ – преемник } Op}$ In[S];
        In[Op] = args[Op] ∪ (Out[Op] \ results[Op]);
    }

Присваивание In[Exit] результатов предиката F необходимо для определения в качестве живых переменных результатов предиката для каждого его оператора по цепочкам преемников. После исполнения оператора Out[Op] = $\bigcup_{S \text{ – преемник } Op}$ In[S] определяются в качестве живых все переменные, которые живы перед исполнением какого-либо преемника этого оператора. После исполнения оператора In[Op] = args[Op] ∪ (Out[Op] \ results[Op]) определяются в качестве живых все переменные, которые либо используются в качестве аргумента в этом операторе, либо используются после этого оператора и не переопределяются в нем. Таким образом, переменная жива после исполнения оператора, если ее текущее значение используется в одной из цепочек преемников этого оператора или она является результатом предиката.

В трансформации склеивания используются множества переменных Out[Op], живых после исполнения оператора.

5. Подстановка определения предиката на место вызова

Пусть A(x: y) { S } – определение предиката в программе, а A(e: z) – вызов предиката в теле некоторого другого предиката B. Здесь x, y, z обозначают списки переменных, а e – список выражений. В соответствии с операционной семантикой языка предикатного

программирования [13] *подстановка определения предиката на место вызова* A(e: z) есть замена вызова следующей композицией:

| x | = | e |; { S }; | z | = | y | . (5.1)

Здесь конструкции | x |, | z | и | y | являются мультипеременными, а | e | – мультивыражением. Оператор присваивания вида | x | = | e | называется *групповым оператором присваивания*.

Можно убрать присваивание | z | = | y |, заменив вхождения y на z в теле S. Если среди e есть переменные, то их также можно использовать в теле S вместо соответствующих переменных в x при выполнении определенных ниже условий. Тогда вызов заменяется следующей композицией:

| x' | = | e' |; { S' } . (5.2)

S' — тело предиката S с заменой z на y и заменой части параметров x на соответствующие переменные в e. Пусть в списке e встречаются переменная a, которой в списке x соответствует переменная b. Если a не используются после вызова или b не перевычисляется в S, то можно заменить b на a в S. x' и e' — оставшиеся части списков x и e.

Возможно, что локальные имена в теле S могут также встречаться в теле предиката B. Это не вызывает проблем, поскольку эти переменные создаются как разные переменные во внутреннем представлении. Однако при обратной трансляции в текст (посредством инструмента pretty-printer) реализуется переименование переменных.

Гиперфункция - предикат с несколькими ветвями [6, 10]. Рассмотрим определение гиперфункции A(x: y #1 : z #2) { S } с двумя ветвями и вызов этой гиперфункции с последующими обработчиками переходов по ветвям:

A(e: g #M1 : f #M2) case M1: { S1 } case M2: { S2 }; N

Здесь N – следующий за обработчиками оператор. После исполнения оператора вызова управление переходит к оператору S1, если предикат завершился по первой ветке, или S2, если по второй. Далее управление переходит к оператору N.

При открытой подстановке тела предиката A вызов заменяется композицией | x' | = | e' |; { S' } { M1: S1; #M3 } { M2: S2: #M3 } , где M3 - метка оператора N. S' - это оператор S, в котором операторы выхода #1 и #2 заменены на #M1 и #M2 соответственно, результаты предиката y и z заменены на результаты вызова g и f, а также часть параметров x заменены на соответствующие переменные в e как описано выше для обычного предиката.

В соответствии с операционной семантикой [13] вычисление выражений, входящих в набор e в групповом операторе присваивания | x | = | e |, проводится параллельно. Реализация такого параллелизма на процессорах с обычной архитектурой не эффективна. Поэтому проводится *раскрытие* группового оператора присваивания его заменой набором обычных операторов присваивания. В общем случае раскрытие группового оператора возможно при использовании дополнительных промежуточных переменных. Например, раскрытие оператора | a, b | = | b, a | реализуется операторами t = a; a = b; b = t с использованием дополнительной переменной t. В большинстве случаев раскрытие возможно без использования дополнительных переменных; иногда достаточно поменять местами операторы присваивания. Например, раскрытие | a, b | = | c, a | реализуется присваиваниями: b = a; a = c.

6. Замена хвостовой рекурсии циклом

Существует специальный случай рекурсии, называемый *хвостовой рекурсией*, когда можно заменить рекурсию циклом. Рекурсивный вызов предиката определяет хвостовую рекурсию, если:

- имя вызываемого предиката совпадает с именем определяемого предиката, в теле которого находится вызов;
- вызов является последней исполняемой конструкцией в определении предиката, содержащем вызов.
- результаты вызова совпадают с соответствующими формальными параметрами – результатами определяемого предиката.

Последняя исполняемая конструкция оператора S принадлежит множеству last(S), которое определим для различных видов операторов S: last(A; B) = last(B), last(**if** (C) A **else** B) = last(A) ∪ last(B), last(D(e: z)) = {D(e: z)}, last(A || B) = ∅. Однако если параллельный оператор A || B реализуется последовательным исполнением A и B, например, как B; A, то last(A || B) = last(A).

Понятие хвостовой рекурсии проиллюстрируем на примерах. В программе умножения через сложение

Умн(**nat** a, b**:** **nat** c) { **if** (a = 0) c = 0 **else** c = b + Умн(a – 1, b) }

рекурсивный вызов Умн(a – 1, b) не является хвостовым, поскольку после завершения исполнения вызова исполняется операция “+”. Хвостовой является рекурсия для каждого из двух рекурсивных вызовов в программе вычисления наибольшего общего делителя:

```
D(nat a, b: nat c)
{ if (a = b) c = a else if (a < b) D(a, b – a: c) else D(a – b, b: c) }
```

Хвостовая рекурсия может быть заменена циклом, что существенно повышает эффективность программы; в частности, дает возможность использовать открытую подстановку. Определение хвостовой рекурсии представлено далее как специальный случай подстановки определения предиката на место вызова (см. разд. 5).

Пусть имеется рекурсивное определение предиката A(x: y) { S } и хвостовой рекурсивный вызов A(e: y) внутри оператора S. Подставим определение предиката A на место вызова A(e: y). Результатом подстановки является фрагмент: | x | = | e |; { S }. Обозначим через S′ оператор, полученный заменой в S вызова A(e: y) на данный фрагмент. Очевидно, можно заменить в S′ второе вхождение S передачей управления на начало оператора S′. В итоге определение предиката A преобразуется к виду: A(x: y) {M: S″ }, где S″ получается из S′ заменой вызова A(e: y) парой операторов: | x | = | e |; **goto** M. Данное преобразование есть трансформация *замены хвостовой рекурсии циклом*.

Алгоритм трансформации. Пусть есть рекурсивный предикат D(**nat** a, b**: nat** c) { S }. В теле S ищутся последние исполняемые конструкции по определенным выше правилам для управляющих операторов суперпозиции, условных и параллельных операторов. Если в найденном множестве есть рекурсивные вызовы, проверяется, что результаты этих вызовов совпадают с соответствующими формальными результатами предиката. Для выявленных хвостовых рекурсий производится замена вызова на присваивание аргументам предиката аргументов вызова и переход в начало тела предиката.

Если в рекурсивном определении предиката A все рекурсивные вызовы имеют вид хвостовой рекурсии, то применение трансформации ко всем этим вызовам преобразует тело предиката в цикл, а определении предиката A – в нерекурсивное определение. После этой трансформации становится эффективной постановка определения предиката A на место вызова A в теле другого предиката.

Применение трансформации замены хвостовой рекурсии циклом покажем для программы вычисления наибольшего общего делителя, приведенной выше. Трансформация двух рекурсивных вызовов предиката D дает следующую программу:

```
D(nat a, nat b: nat c) {
M:     if (a = b) c = a
       else   if (a < b) |a, b| = |a, b – a|; goto M
              else |a, b| = |a – b, b|; goto M
```

```
}
```

Раскроем групповые операторы присваивания, а также заменим фрагмент с операторами перехода на цикл **loop**. Получим:

```
D(nat a, nat b: nat c) {
        loop {
                if (a = b) {c = a; break; }
                if (a < b) b = b – a
                else a = a – b
        }
}
```

Замена фрагмента с операторами перехода на цикл **loop** не меняет процесса исполнения. Преобразование такого рода, улучшающее структуру программы, называется *оформлением.*

7. Склеивание переменных

Трансформация *склеивания переменных* a ← x есть замена в предикатной программе всех вхождений каждой переменной из списка переменных x на переменную a. Например, склеивание переменных a ← b, c представляет замену всех вхождений в программе имен b и c на имя a.

В отличие от задачи экономии памяти [4], склеиванию подлежат результаты программы с аргументами или локальные переменные с результатами, между которыми имеется информационная связь. Задача склеивания переменных не актуальна для оптимизации функциональных программ: там можно было бы рассматривать лишь склеивание аргументов функций и локальных переменных конструкций **let** и **where**. Эта задача не возникает также для императивных программ, поскольку практически все рассматриваемые здесь склеивания обычно проведены программистом в императивной программе.

В языке P запрещено присваивание вида: x:=op(x, y). Вместо этого используется присваивание x:=op(x1, y) в предположении, что переменная x1 должна быть склеена с переменной x при трансляции на императивное расширение языка P. Например, при склеивании переменных c и d оператор c:=d+1 будет преобразован в оператор присваивания c:=c+1, а оператор a:=b при склеивании a и b превратится в оператор a:=a, удаляемый из программы. Склеивание переменных, массивов или списков, предотвращает копирование больших структур данных.

Склеивание переменных может задаваться в предикатной программе. Если имеется результирующая переменная, имя которой завершается штрихом, при наличии аргумента с тем же именем, то результирующая переменная должна быть склеена с аргументом.

Например, результат a' при наличии аргумента a неявно определяет трансформацию $a \leftarrow a'$. Набор склеиваний может быть частично задан программистом. Необходимо проверить его корректность и дополнить.

Эквивалентность программы до и после проведения трансформации склеивания достигается при выполнении ряда условий. Одно из них: аргумент a, склеенный с результатом b, не является живой после присваивания переменной b.

В общем случае задача склеивания очень сложна. В действительности, набор склеиваний планируется программистом при написании предикатной программы. Поэтому нет задачи нахождения наилучшего варианта склеивания. Необходимо воспроизвести вариант склеивания, запланированный программистом.

Общая схема реализации. *Регион склеивания* для оператора G есть пара <x: y>, где x — подмножество аргументов оператора, а y — один из его результатов, все переменные имеют одинаковый тип, а переменные x не являются живыми переменными оператора G. Неживую переменную можно склеить с другой, так как она не используется далее в программе. **Пример**. Пусть имеется оператор F с аргументами a, b, c, d, e и результатами f, g, h. Пусть переменные a, b, d и g, h имеют натуральный тип, а переменные c, e и f — массивы. Тогда для оператора F регионы склеивания могут быть такими: <a, b: g>, <d: h> и <c, e: f>.

Регион склеивания вида <a**:** g> , где a и g — переменные, является *командой склеивания*, определяющей замену переменной a на g.

Запрет на склеивание для оператора G есть пара вида {a**:** g} , где a и g — переменные. Он запрещает склеивание переменной g с переменной a.

Алгоритм склеивания для программы в целом реализуется перебором всех предикатов программы в порядке, определенном потоковым анализатором. Он гарантирует, что склеивания для формальных параметров предиката будут перенесены на фактические параметры в вызовах этого предиката, что необходимо в случае открытой подстановки. Для рекурсивных вызовов перенос склеиваний на аргументы вызова необходим лишь для вызовов с хвостовой рекурсией.

Склеивание переменных предиката реализуется с помощью алгоритма уточнения регионов. Дерево тела предиката обходится снизу вверх; строятся начальные регионы склеивания, которые далее уточняются, проверяя выполнение ограничений. По построенным регионам склеивания для предиката выбираются команды склеивания. Применение команд склеивания к программе реализуется независимым просмотром программы. Там же реализуется замена параллельных операторов операторами суперпозиции, о которой сказано ниже.

Построение регионов склеивания. Для построения регионов склеивания дерево тела предиката обходится снизу вверх. В дереве тела предиката нижними операторами являются операторы присваивания и вызовов предикатов. Для этих операторов строятся начальные регионы склеивания определенным ниже образом. Регионы операторов суперпозиции, параллельных и условных операторов строятся на основе уже построенных регионов подоператоров. Построенные таким способом регионы являются корректными, т.е. при проведении склеиваний по этим регионам сохраняется эквивалентность программы.

Для оператора **присваивания** вида a = b строится регион <b: a>, при условии, что b не является живой переменной этого оператора. Склеивание превратит этот оператор в тождественный оператор присваивания b = b, который далее будет удален из тела предиката. Для оператора присваивания с несколькими аргументами $a = b_1 + \ldots + b_n$ регион строится только если тип переменной присваивания — список, для которого операция конкатенации '+' не вычисляет новое значение, а добавляет к левому аргументу операции правый. Для оператора такого вида строиться регион <b_1: a>. В таком случае при склеивании исходный оператор заменится на оператор $b_1 = b_1 + \ldots + b_n$, в котором b_1 не будет копироваться.

Для определения предиката, вызываемого нерекурсивным оператором **вызова**, команды склеивания уже построены, что достигается выбранным порядком обработки предикатов по графу вызовов. Пусть есть определение предиката F(…, a, …**:** …, b, …) {…}, в котором переменные a и b склеиваются, и есть нерекурсивный вызов F(…, d, …**:** …, e, …). При последующей трансформации подстановки определения предиката на место вызова переменные a и b будут заменены на, соответственно, d и e. Так как в предикате F переменные a и b склеились, то переменные d и e тоже можно склеить. Поэтому строится регион <d: e>.

Если соответствующий аргумент вызова – не переменная, то регионов не строится, за исключением случая конкатенации строк $b_1 + \ldots + b_n$, описанного выше для оператора присваивания.

Для хвостовых рекурсий регионы строятся после просмотра тела рекурсивного предиката и построения команд склеивания. Пусть имеется определение предиката F(…, a, …**:** …, b, …) {…}, для которого была построена команда склеивания <a: b>. Пусть в теле предиката есть хвостовой рекурсивный вызов F(…, d, …: …, b, …). Поскольку замена хвостовой рекурсии циклом интерпретируется как открытая подстановка тела на место вызова, соответствующие переменные d и b, позиции которых в вызове соответствуют позициям a и b в списке формальных параметров, должны быть склеены. Поэтому набор команд склеивания предиката F дополняется командой <d: b>.

Для нехвостовых рекурсивных вызовов регионов склеивания не строится, так как такие вызовы не будут заменяться определениями предикатов.

В подоператорах **параллельного оператора** все результаты различны, а аргументы делятся на уникальные — участвующие только в одном подоператоре, и общие — участвующие более чем в одном подоператоре. Уникальные аргументы можно склеить с результатами, а общие аргументы склеивать нельзя, т.к. они одновременно используются в других подоператорах.

В предикатном программировании возможны две реализации параллельного оператора: через последовательное исполнение и через параллельное. Параллелизм на уровне простых операторов допускает эффективную реализации лишь для архитектуры широких команд Эльбрус-3. Для прочих архитектур параллельный оператор в подавляющем большинстве случаев целесообразно реализовать последовательным исполнением. Алгоритм склеивания зависит от способа реализации параллельного оператора. При параллельной реализации оператора G(d, e: a) || F(d, e: b) склеивание d с a невозможно, так как d может поменять свое значение до того, как будет использована в операторе F; тогда как при последовательной реализации оператор преобразуется в F(d, e: b); G(d, e: a) и конфликта не возникнет.

Регионы для параллельного оператора строятся по регионам его подоператоров. Цель алгоритма построения регионов – стремится найти наибольшее количество склеиваний за приемлемое время. Если регион содержит и уникальные и общие аргументы, то все общие удаляются. Далее для регионов только с общими аргументами выбирается аргумент, встречающийся в наименьшем количестве регионов. Выбираются регионы, содержащие его, и удаляются все, кроме одного. В оставшемся регионе из левой части удаляются все аргументы, кроме данного общего. Таким способом обрабатываются все регионы подоператоров; получившиеся регионы становятся регионами параллельного оператора. Отметим, что, следуя такому алгоритму, мы можем потерять возможные варианты склеивания.

Регионы **условного оператора** строятся из регионов его ветвей. Сначала регионы ветвей корректируются с учетом запретов на склеивание. Если есть регион <…, a, …: b> и запрет на склеивание {a: b}, то аргумент a удаляется из региона.

Пусть есть регионы <…, a, …: b> и <…, a, …: d>, где a — аргумент оператора, а b и d — результаты разных ветвей. Тогда a удаляется из этих регионов, так как нельзя склеить одну переменную с двумя живыми, и создаются запреты на склеивание {a: b} и {a: d}. Они являются живыми, так как являются результатами оператора, следовательно, либо являются результатами предиката, либо используются далее для вычисления других переменных.

Данное правило не относится к регионам с локалами ветвей в правых частях. Если b локал ветви, то оба этих склеивания можно провести, так как локал, в отличие от результата ветвей, не будет использоваться после условного оператора.

Пусть есть регионы <x: z> и <y: z>, где z — результат условного оператора, а x и y — наборы аргументов и локалов. Такие регионы объединяются в один <x, y: z>, который означает, что z можно склеить только с одной из переменных наборов x и y.

Обработанные регионы ветвей становятся регионами условного оператора.

Для **оператора суперпозиции** регионы строятся как совокупность регионов подоператоров.

Запреты на склеивание условного, параллельного оператора и оператора суперпозиции строятся как совокупность запретов подоператоров.

Построение команд склеивания. Команды склеивания для предиката строятся уточнением регионов склеивания тела этого предиката. Общая цель – по региону склеивания <x: y> с несколькими аргументами построить команду склеивания <z: y>, определяющую замену y на z. Сначала регионы уточняются набором *априорных склеиваний* и *запретов*, заданных в исходной программе. Программист может задать в заголовке предиката склеивание результата с аргументом. В этом случае имя результата есть имя аргумента с добавлением штриха в конце (<a’: a>). Команда склеивания (или запрета склеивания) может быть также задана прагмой. Проверяется, что априорные склеивания содержатся в регионах, т. е. для каждого априорного склеивания есть такой регион склеивания, в правой части которого содержится результат априорного склеивания, а одна из переменных левой части — аргумент априорного склеивания. В противном случае выдаются сообщения о некорректном задании априорных склеиваний. Запрет на склеивание — пара переменных, означающая, что вторую переменную нельзя склеить с первой. Проверяется, что данная пара переменных не принадлежит ни одному региону. Если есть такой регион склеивания, в правой части которого содержится вторая переменная запрета склеивания, а одна из переменных левой части региона — первая переменная запрета склеивания, то эта первая переменная удаляется из региона.

Все получившиеся регионы с одним результатом и аргументом считаются командами склеивания. Для регионов с несколькими аргументами произвольным образом выбираются команды склеивания с одним результатом и одним аргументом. Все возможные варианты выбора равноценны, так как склеивание результата с любым из аргументов уберет одно копирование в предикате.

Далее обрабатываются команды, содержащие локалы предиката. Если команда имеет вид <аргумент: локал>, то во все команды, кроме текущей, аргумент подставляется на место локала.

После построения полного набора команд склеивания программа обходится сверху вниз, склеивая переменные исходя из команд склеивания. Там же реализуется замена параллельных операторов, в которых происходит склеивание общих аргументов, операторами последовательного исполнения.

Алгоритм преобразования параллельного оператора. Для параллельного оператора из построенных команд склеивания выбираются те, что содержат его результаты в правых частях. Рассматривается каждая выбранная команда склеивания <x: y>. В подоператорах параллельного оператора аргументы делятся на уникальные — участвующие только в одном подоператоре, и общие — участвующие более чем в одном подоператоре. Если x — уникальный аргумент параллельного оператора, то склеивание можно произвести без конфликтов с другими подоператорами. Если x — общий аргумент параллельного оператора, а y — результат параллельного оператора, то необходимо трансформировать параллельный оператор. Подоператор **B**, в результатах которого содержится y, необходимо удалить из параллельного оператора, заменив последний на **A; B** , где **A** — исходный параллельный оператор без подоператора **B**. В операторе **B** этот аргумент станет уникальным, и с ним можно будет склеить результат.

Пример 1.

Работу алгоритма склеивания переменных покажем на примере предикатной программы нахождения целочисленного квадратного корня:

```
sq1(nat x, k, n : nat m) {
	nat p = n + 2* k + 1;
	if (x < p) m = k else  sq1(x, k + 1, p: m)
}
```

Построение регионов реализуется, начиная с нижних, простых операторов. Для оператора присваивания `m = k` строится регион <k: m>, так как k – неживая переменная, т.е. ее значение не используется после этого оператора. Для хвостового рекурсивного оператора вызова `sq1(x, k + 1, p: m)` регионов при первом просмотре программы не строится. Для условного оператора регионы его ветвей объединяются: <k: m>. Для оператора присваивания `nat p = n + 2* k + 1` регионов не строится, т.к. это не оператор вида a = b, где b - переменная. Тело программы — суперпозиция оператора присваивания с условным оператором. Регионы оператора суперпозиции строятся объединением регионов его

подоператоров. В результате для тела программы и для программы sq1 в целом получим регионы <k: m>. Этот регион является командой склеивания. Далее по этой команде склеивания строятся регионы на хвостовых рекурсивных вызовах. Но для оператора вызова sq1(x, k + 1, p**:** m) регионов не строится, так как соответствующий аргумент k + 1 — не переменная.

Процесс склеивания реализуется обходом дерева программы. Вхождения в программе переменных из правых частей команд склеивания заменяются соответствующими переменными из левых частей, т.е. переменная переменная m заменяется на k.

Итоговая программа, в которой произведено склеивание <k: m>:

```
sq1(nat x, k, n) {
        nat p = n + 2* k + 1;
        if (x < n) k = k else  sq1(x, k + 1, p: k)
}
```

При упрощениях программы оператор k = k будет удален.

Пример 2.

Предикат sort1 реализует сортировку массива простыми вставками.

```
type T;
nat n;
type natn = 0 .. n;
type Arn = array (natn, T);

pop_into(Arn a, natn k, m, T e: Arn a’) {...}

sort1(Arn a, natn m: Arn a’) {
      if (m = n) a’ = a
      else { T e = a[m+1];
            if (a[m] <= e) sort1(a, m+1: a’)
            else { pop_into(a, m+1, m, e: Arn c);
                  sort1(c, m+1: a’)
            }
      }
}
```

Построение регионов реализуется, начиная с простых операторов. Для оператора присваивания a’ = a строится регион <a: a'>, так как a – неживая переменная. Для оператора присваивания T e = a[m+1] регионов не строится, т.к. это не оператор вида a = b, где b - переменная. Для хвостовых рекурсивных операторов вызова sort1(a, m+1**:** a’) и sort1(c, m+1**:** a’), регионов на первом просмотре программы не строится. В вызываемом предикате pop_into первый аргумент склеивается с результатом. Поэтому для оператора

вызова pop_into(a, m+1, m, e: Arn c) строится регион <a: c>. В операторах суперпозиции и условных операторах регионы подоператоров объединяются. В результате для тела программы и для программы sort1 в целом получим регионы <a: c> и <a: a'>. Эти регионы являются командами склеивания. Команда <a: a'> склеивает параметры предиката и поэтому по ней строятся регионы на хвостовых рекурсивных вызовах. Для оператора вызова sort1(a, m+1: a′) строится регион <a: a'>, для оператора вызова sort1(c, m+1: a′) строится регион <c: a'>. Оба региона уже присутствуют в командах склеивания.

Итоговая программа, в которой произведены склеивания <a: c> и <a: a'>:

```
sort1(Arn a, natn m: a) {
        if (m = n) a = a
        else { T e = a[m+1];
                if (a[m] <= e) sort1(a, m+1: a)
                else { pop_into(a, m+1, m, e: Arn a);
                        sort1(a, m+1: a)
                }
        }
}
```

8. Упрощения и оформления

В результате склеивания переменных a с b в операторах вида a = b возникает тождественный оператор a = a, который удаляется из программы. Также производятся другие изменения в программе, относящиеся к упрощениям:

- Упрощаются параллельный оператор и оператор суперпозиции, содержащие пустой оператор.
- Упрощаются пустые ветви условного оператора и сам оператор, если обе ветви пусты.
- Упрощается оператор switch, в котором все ветви пусты.
- Замена оператора модификации вида a = a with [k: e] оператором присваивания a[k] = e.
- Удаление оператора перехода на следующий оператор. Реализуется после открытой подстановки вызова гиперфункции.

9. Примеры трансформации программ

Приведем на примерах этапы трансформации программ.

9.1. Программа нахождения целочисленного квадратного корня:

```
sq1(nat x, k, n : nat m) {
        nat p = n + 2* k + 1;
        if (x < p) m = k else  sq1(x, k + 1, p: m)
}
```

Склеивания в sq1: k ← m;

```
sq1(nat x, k, n : nat k) {
        nat p = n + 2* k + 1;
                if (x < p) k = k else  sq1(x, k + 1, p: k)
}
```

Замена хвостовой рекурсии циклом:

```
sq1(nat x, k, n : nat k) {
        M: nat p = n + 2* k + 1;
        if (x < n) k = k else { |x, k, p| = |x, k + 1, n|; goto M }
}
```

Упрощения и оформления:

```
sq1(nat x, k, n : nat k) {
        M: nat p = n + 2* k + 1;
        if (x >= n) { |k, p| = |k + 1, n|; goto M }
}
```

9.2. Программа умножения через сложение:

```
Умн(nat a, b:  nat c) {
         Умн1(a, b, 0: c)
}
Умн1(nat a, b, d:  nat c) {
         if (a = 0)  c = d
         else Умн1(a – 1, b, d + b: c)
}
```

Склеивание в Умн1: c ← d.

```
Умн1(nat a, b, c:  nat c) {
         if (a = 0)  c = c
         else  Умн1(a – 1, b, c + b: c)
}
```

Замена хвостовой рекурсии циклом:

```
Умн1(nat a, b, c:  nat c) {
         M: if (a = 0) c = c
         else {
               |a, b, c| = |a – 1, b, c + b|;
                goto M
         }
}
```

Подстановка определения Умн1 на место вызова в Умн:

```
Умн(nat a, b: nat c) {
        c = 0;
        M: if (a = 0) c = c
        else {
                |a, b, c| = |a – 1, b, c + b|;
                goto M
        }
}
```

Упрощения и оформления:

```
Умн(nat a, b: nat c) {
        c = 0;
        M: if (a != 0) {
                |a, c| = |a – 1, c + b|;
                goto M
        }
}
```

9.3. Программа сортировки простыми вставками:

```
type T; // произвольный тип с линейным порядком
nat n;  // n – 1 – число элементов сортируемого массива
type natn = 0 .. n;
type Arn = array (natn, T);
sort(Arn a: Arn a’) { sort1(a, 0: a') }
```

В предикате sort1 предполагается, что первые m элементов массива a – отсортированы. Итоговый массив a’ полностью отсортирован.

```
sort1(Arn a, natn m: Arn a’) {
      if (m = n) a’ = a
      else { T e = a[m+1];
               if (a[m] <= e) sort1(a, m+1: a’)
               else { pop_into(a, m+1, m, e: Arn c);
               sort1(c, m+1: a’)
               }
      }
}
```

Предикат pop_into вставляет элемент e, который изначально был элементом a[m]. При этом элементы от a[k] до a[m-1] сдвинуты на одну позицию вправо и все они больше e. При этом в позиции k – «дыра». Итоговый массив a’ – отсортирован.

```
pop_into(Arn a, natn k, m, T e: Arn a') {
        Arn b = a with [k: a[k-1]];
        if (k = 1) a' = b with [0: e]
        else if (b[k-2] <= e) a' = b with [k-1: e]
                else pop_into(b, k-1, m, e: a')
}
```

Склеивания в sort: a ← a’

```
sort(Arn a: a) { sort1(a, 0: a) };
```

Склеивания в sort1: a ← a’, c

```
sort1(Arn a, natn m: a) {
        if (m = n) a = a
        else { T e = a[m+1];
                if (a[m] <= e) sort1(a, m+1: a)
                else { pop_into(a, m+1, m, e: Arn a);
                sort1(a, m+1: a)
                }
        }
}
```

Склеивания в pop_into: a ← a’, b

```
pop_into(Arn a, natn k, m, T e: a) {
        a = a with [k: a[k-1]];
        if (k = 1) a = a with [0: e]
                else if (a[k-2] <= e) a = a with [k-1: e]
                else pop_into(a, k-1, m, e: a)
};
```

Замена хвостовой рекурсии циклом в sort1 и pop_into:

```
sort1(Arn a, natn m: a) {
         M: if (m = n) a = a
         else { T e = a[m+1];
                if (a[m] <= e) { |a, m| = |a, m+1|; goto M; }
                else { pop_into(a, m+1, m, e: Arn a);
                        |a, m| = |a, m+1|; goto M;
                }
         }
}
pop_into(Arn a, natn k, m, T e: a) {
         M: a = a with [k: a[k-1]];
         if (k = 1) a = a with [0: e]
                else if (a[k-2] <= e) a = a with [k-1: e]
                        else { |a, k, m| = |a, k - 1, m|; goto M; }
};
```

Подстановка определения pop_into на место вызова в sort1:

```
sort1(Arn a, natn m: a) {
        M: if (m = n) a = a
        else { T e = a[m+1];
                if (a[m] <= e) { |a, m| = |a, m+1|; goto M; }
                else {
                        natn k =  m+1;
                        M1: a = a with [k: a[k-1]];
                        if (k = 1) a = a with [0: e]
                        else if (a[k-2] <= e) a = a with [k-1: e]
                                else { |a, k, m| = |a, k - 1, m|; goto M1; }
                        |a, m| = |a, m+1|; goto M;
                }
        }
}
```

Подстановка определения sort1 на место вызова в sort:

```
sort(Arn a: a) {
        natn m = 0;
        M: if (m = n) a = a
        else { T e = a[m+1];
                if (a[m] <= e) { |a, m| = |a, m+1|; goto M; }
                else {
                        natn k =  m+1;
                        M1: a = a with [k: a[k-1]];
                        if (k = 1) a = a with [0: e]
                            else if (a[k-2] <= e) a = a with [k-1: e]
                                  else { |a, k, m| = |a, k - 1, m|; goto M1; }
                        |a, m| = |a, m+1|; goto M;
                }
        }
 };
```

Упрощения и оформления:

```
sort(Arn a: a) {
        natn m = 0;
        M: if (m != n) {
                T e = a[m+1];
                if (a[m] <= e) { m = m+1; goto M; }
                else {
                        natn k = m+1;
                        M1: a[k] = a[k-1];
                        if (k = 1) a[0] = e
                        else if (a[k-2] <= e) a[k-1] = e
                                else { k = k - 1; goto M1; }
                        m = m+1; goto M;
                }
        }
```

```
};
```

При завершении оптимизации и генерации программы на C++ реализуются косметические преобразования вставки циклов **while** и **for** вместо операторов перехода.

10. Обзор работ

Наиболее известным проектом в области трансформационного программирования является проект CIP [15-17], реализованный в Техническом университете Мюнхена. В проекте CIP определяется язык CIP-L, называемый также The wide spectrum language 84, содержащий следующие подъязыки:

- язык спецификаций, аналогичный языку исчисления предикатов;
- аппликативный язык – язык функционального программирования;
- диалекты императивных языков Pascal и Algol**.**

Различные уровни языка CIP-L интегрируются формальным описанием трансформационной семантики. Трансформационное правило для каждой языковой конструкции определяет построение математической функции из функций для подконструкций данной конструкции. Построение программы на языке CIP-L реализуется применением набора трансформаций для начальной более простой версии программы. Трансформации реализуются в автоматическом режиме с проверкой условий их корректности. Аппарат трансформаций является универсальным в CIP. Трансформации могут применяться для доказательства утверждений, для преобразования программы в рамках аппликативного языка, для преобразования с аппликативного языка на императивный, а также для оптимизации программы на императивном языке. Имеются трансформации линейной рекурсии в хвостовую, а хвостовой – в цикл типа **while**. Отметим, что наши трансформации склеивания переменных и кодирования алгебраических типов значительно сложнее определяемых в проекте CIP. Они определяются в рамках формальной операционной семантики, а не денотационной.

В функциональном программировании оптимизация программы полностью возлагается на транслятор. Определенный прогресс в эффективности достигается разработкой сверхмощных интерпретаторов и генераторов кода для языка SequenceL [18], определяющего предельно декларативный стиль программирования. Функциональное программирование существенно уступает в эффективности, поскольку невозможно автоматически воспроизвести серию оптимизаций, совершаемых вручную в предикатном программировании.

Замена хвостовой рекурсии циклом и подстановка тела программы на место вызова – типичные, хорошо известные оптимизирующие преобразования. Замена хвостовой рекурсии циклом проводится лишь для функциональных языков. Трансформация склеивания переменных реализуется только для предикатного программирования. Склеиваются аргумент и результат некоторого фрагмента программы, включая также используемые промежуточные переменные между аргументом и результатом. Стиль программирования, при котором такое склеивание подразумевается, характерен лишь для предикатного программирования. В императивном программировании склеивание проводит программист. В функциональном программировании склеивание возможно лишь для конструкций типа **let** и **where**, однако в публикациях по функциональным языкам подобной трансформации не обнаружено.

Алгоритм склеивания переменных разработан Э. Петровым [8] в 2003г. в рамках первой версии системы предикатного программирования. Склеивание переменных проводилось на базе кандидатных множеств, по структуре отличных от регионов склеивания. Алгоритм Э. Петрова существенно сложнее нашего. Для параллельного оператора алгоритм имеет временную сложность $O(n!)$, тогда как наш алгоритм – $O(n^2)$, однако последний может пропустить некоторые варианты склеивания.

Термин «склеивание переменных» впервые появился в рамках задачи экономии памяти в классических работах А. П. Ершова, С. С. Лаврова, В. В. Мартынюка. Склеивание переменных определялось как выбор переобозначения аргументов и результатов из заданного множества корректных переобозначений, которое позволяет в наибольшей степени уменьшить объем необходимой памяти [4]. В нашем подходе, в отличие от задачи экономии памяти, склеиванию подлежат только те переменные одного типа, между которыми имеется информационная связь.

Оптимизация памяти для современных компьютеров актуальна при распределении регистров. В этой задаче большое число используемых переменных необходимо разместить на небольшом количестве регистров. Используется метод раскраски графа несовместимостей [19], по цветам распределяя регистры для хранения переменных.

11. Заключение

В настоящей работе описывается оптимизация предикатных программ с трансляцией на язык C++, названная оптимизацией среднего уровня, существенно отличающаяся от классической оптимизации императивных программ. Оптимизация реализует следующий

набор трансформаций: склеивание переменных, замену хвостовой рекурсии циклом, подстановку определения предиката на место вызова, упрощения и оформления.

В целях оптимизации проводится потоковый анализ предикатной программы. Строится граф вызовов программы с использованием достаточно точной аппроксимации значений переменных предикатного типа. Определяются аргументы и результаты операторов программы и области жизни переменных.

Описываемые оптимизирующие трансформации иллюстрируются на различных предикатных программах в работах [3, 6, 9, 10, 12, 20]. Трансформации частично реализованы в рамках экспериментальной системы предикатного программирования. Трансформация кодирования алгебраических типов (списков, строк и деревьев) через массивы и указатели описана в работах [2, 11, 14].

В дальнейшем планируется разработка трансформаций для автоматического распараллеливания программ. Планируется также расширить возможности склеивания переменных реализацией склеивания переменных с отдельными компонентами структур.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Список литературы

1. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман Компиляторы: принципы, технологии и инструментарий = Compilers: Principles, Techniques, and Tools. — 2 изд. — М.:Вильямс, 2008. — ISBN 978-5-8459-1349-4
2. Булгаков К.В., Каблуков И.В., Тумуров Э.Г., Шелехов В.И. Оптимизирующие трансформации списков и деревьев в системе предикатного программирования // Системная информатика, № 9. — Новосибирск, 2017. — С. 63-92. [Электронный ресурс]. URL: http://www.system-informatics.ru/files/article/105.pdf
3. В.А. Вшивков, Т.В. Маркелова, В.И. Шелехов. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ, Т.4(33), с. 79-94, 2008.
4. Ершов А.П. Введение в теоретическое программирование. М.: Наука, 1977. 288с.
5. Каблуков И. В., Шелехов В.И. Реализация склеивания переменных в предикатной программе. – Новосибирск, 2012. – 6с. – (Препр. / ИСИ СО РАН; N 167).
6. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12 — Новосибирск, 2013. — 52с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf
7. Касьянов В. Н., Евстигнеев В. А. - Графы в программировании: обработка, визуализация и применение – БХВ – Петербург, 2003 – ISBN 5-94157-184-4

8. Петров Э.Ю. Склеивание переменных в предикатной программе // Методы предикатного программирования. Новосибирск, 2003. С. 48-61.
9. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.
10. Шелехов В.И. Доказательное построение, верификация и синтез предикатных программ // Знания-Онтологии-Теории (ЗОНТ-2017), Том 2. — Институт Математики СО РАН, Новосибирск, 2017. — С. 156-165. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/lbase.pdf
11. Шелехов В.И. Предикатная программа вставки в АВЛ-дерево // Системная информатика, № 9. — Новосибирск, 2017. — С. 23-42. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/avl_insert.pdf
12. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. – Новосибирск, 2012. – 30с. – (Препр. / ИСИ СО РАН. N 164).
13. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. Новосибирск, 2015. 13с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf
14. Шелехов В.И. Списки и строки в предикатном программировании // Системная информатика, №3, 2014. ИСИ СО РАН, Новосибирск. С. 25-43. [Электронный ресурс] URL: http://persons.iis.nsk.su/files/persons/pages/String.pdf
15. Bauer F.L., Broy M., et.al. The Wide Spectrum Language 84 / Inst. For Informatik. Technische Universitat Munchen, 1983. – 157p.
16. Bauer F.L., Broy M., et.al. Wide Spectrum Language for Program Specification and Development (Tentative Version). – Munchen, 1981. – 236p. – (Prepr: Inst. For Informatik / Technische Universitat Munchen / TUM-18104)
17. Brass B., Erhard F., Horsch A., Riethmayer H.-O., Steinbruggen R. CIP-S: An instrument for program transformation and rule generation. – Munchen, 1982. – P. 44-62. – (Prepr: Inst. For Informatik / Technische Universitat Munchen / TUM-18211)
18. Cooke D. E., Rushton J. N. Taking Parnas's Principles to the Next Level: Declarative Language Design // Computer, Vol. 42, no. 9. 2009. P. 56-63.
19. George, Lal; Appel, Andrew W. (May 1996). "Iterated Register Coalescing". ACM Trans. Program. Lang. Syst. 18 (3): 300–324 doi:10.1145/229542.229546. ISSN 0164-0925.
20. Shelekhov V. I. 2011. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements. Automatic Control and Computer Sciences. Vol. 45, No. 7, 421–427.

УДК 004, 9

SoRuCom-2017:

Международная конференция по истории информатики

Крайнева И.А. (Институт систем информатики СО РАН)

В статье представлены итоги работы 4-й Международной конференции «Развитие вычислительной техники в России и странах бывшего СССР: история и перспективы» (SoRuCom-2017) Зеленоград, Москва, 3-5 октября 2017 г., в организации которой ИСИ СО РАН принимал самое активное участие. Поддержку конференции оказал Российский фонд фундаментальных исследований, Грант 17-07-20538, Computer History Museum (USA, CA), российский гражданин Владимир Курляндчик.

Ключевые слова: *SoRuCom, история информатики, советские ЭВМ, научно-техническая политика в области ВТ, модернизация, СССР*

1. Введение

Зарождение практики изучения вычислительной техники и программирования как исторических феноменов относится ко второй половине 1950-х годов. Причем если по программированию сразу появляются работы обобщающего характера, то ЭВМ представляют штучно, поскольку информация о них была, как правило, закрытой [1, 9]. Некоторое исключение составили материалы конференции «Пути развития советского математического машиностроения и приборостроения» 1956 г., где были названы и охарактеризованы первые советские ЭВМ, с энтузиазмом восприняты новые возможности, которые предоставляла эта техника [7].

М.Р. Шура-Бура отметил особенности раннего этапа становления программирования до появления трансляторных проектов, для которого были характерны поиск и изобретение надежных методов организации вычислений [12]. В начале 1960 г. Н.А. Криницкий выделил две проблемы общего характера, которые возникли с появлением ЭВМ: проблема подготовки задач для решения на этих машинах и выявление областей их применимости. Он, как и Шура-Бура, констатировал, что методика программирования в ранний период состояла в использовании символических адресов, что при ручном программировании заключалось в применении буквенно-цифровых обозначений [6]. А.А. Ляпунов, характеризуя теоретические исследования в области программирования, отметил различие в подходах

московских (А.П. Ершов, А.А. Ляпунов, Р.И. Подловченко, Ю.И. Янов – операторный метод), ленинградских (Л.В. Канторович, Л.Т. Петрова, В.А. Булавский – крупноблочное программирование), и киевских математиков (Л.А. Калужнин – метод граф-схем) [8]. Несколько позднее появилось исследование, где были названы первые отечественные проекты по созданию ВТ, проанализирован начальный период становления программирования в СССР [3].

Следующий этап в изучении истории отечественной ВТ наступил в середине 1990-х гг. Э.М. Пройдаковым (Москва) был организован сайт «Виртуальный компьютерный музей» [2], возникла критическая масса информации об отечественных проектах. Пришло время для создания форума, не только аккумулирующего фактографическую информацию об отечественных разработках в области вычислительной техники и программирования, но и способного ее анализировать в историческом контексте. В это время многие советские закрытые проекты были обнародованы, а специалисты, участвовавшие в этих проектах, смогли открыто говорить о своих работах, сравнивать их с западными. В открытой печати появились технические характеристики практически всех советских ЭВМ, стали публиковаться воспоминания участников событий, биографические эссе ключевых фигур отечественного и зарубежного компьютеростроения и программирования. Через 10 лет была созвана первая конференция «Развитие вычислительной техники в России и странах бывшего СССР: история и перспективы» (SoRuCom, Soviet-Russian Computing). Она прошла в 2006 г. в Петрозаводске, следующая – в 2011 г. в Великом Новгороде [4], третья – в 2014 г. в Казани [10]. Ее ведущими участниками являлись и являются профильные специалисты, люди, которые принимали непосредственное участие в советских проектах по созданию ЭВМ, представители школ программирования, инженеры и ученые.

2. Научная программа SoRuCom-2017

Конференция прошла в Зеленограде 3–5 октября 2017 г. на базе Национального исследовательского университета МИЭТ. Она является уникальным в России мероприятием, где участники представляют и обсуждают результаты исследований по проблематике советской вычислительной техники и программирования в исторической ретроспективе [11]. На SoRuCom-2017 был принят 61 доклад (81 докладчик, из них 1 академик, 24 доктора наук, 30 кандидатов наук, 26 научных сотрудников и инженеров). В составе докладчиков 40 представителей вузов, 20 из институтов РАН , 17 из проектно-конструкторских учреждений и компаний, 4 – от учреждений культуры. 61 доклад опубликован в трудах. География конференции: Россия (Зеленоград, Москва, Мытищи, Санкт-Петербург, Новосибирск,

Самара, Казань, Екатеринбург, Нижний Новгород, Ростов-на-Дону), Беларусь, Польша, Прибалтика, Финляндия, США.

Как в начале, так и теперь историки науки на конференции представлены слабо. Это говорит о том, что в нашем отечестве недостаточно привлечено внимание к данной проблематике в среде профессиональных историков науки, как в силу ее сложности, так и в силу отсутствия достаточного числа кадров (к примеру, в Институте истории естествознания и техники РАН историей информатики занимается 1 сотрудник, математик по образованию). Однако можно говорить о нескольких сложившихся центрах исследования истории вычислительной техники и программирования в СССР, локализованных по месту нахождения ведущих школ информатики: киевском, московском, санкт-петербургском и новосибирском. Представители этих научных школ являлись и являются активными акторами исследовательского процесса. В этот раз специалисты из Украины были представлены только д.ф.-м.н. Е.М. Лаврищевой, которая долгое время работала в Институте кибернетики в Киеве, а ныне является сотрудником Института системного программирования РАН в Москве. Ее доклад был посвящен развитию теории программ и систем в СССР.

Традиционным для конференции стал ее формат: пять секций (история вычислительной техник, история программирования, сети и программные системы, информатика и образование, сохранение историко-научного наследия), открывающая и завершающая пленарные сессии. В работе секции «Информатика и образование» участвовали представители двух поколений, принимавших непосредственной участие в работе по информатизации и компьютеризации школы (А.Г. Гейн и Н.А. Юнерман из Екатеринбурга, Т.И. Тихонова из Новосибирска). В докладах была отмечена роль академика А.П. Ершова в разработке национальной программы информатизации школы, а так же тех практических шагов и навыков, которые он и его ученики, в частности Г.А. Звенигородский, применяли в учебном процессе. Благодаря своевременному появлению предмета «Основы информатики и вычислительной техники», СССР, а затем и Россия оказались в числе стран, обладающих высоким кадровым потенциалом программистов, навыки которых высоко ценят в софтверных компаниях всего мира.

Секция «Сохранение историко-научного наследия» была сформирована уже в 2011 г., поскольку к работе конференции подключились историки науки, музеологи и журналисты. Эта секция по количеству представленных докладов (13 докладов) успешно конкурировала с секцией по истории вычислительной техники, традиционно многочисленной (16 докладов). Кроме того, исторический подход проник в среду профильных специалистов, которые

увязывали свои исследования не только с конкретными проектами, но и с историческим контекстом (Холодная война, техническая модернизация, идеологические дискуссии в области науки и техники). На данной конференции вклад в историческое исследование реалий информатики внесли ученые Санкт-Петербурга, Москвы, Новосибирска, Нижнего Новгорода, Казани, Риги, Самары, Ростова-на-Дону и Катовице. На секции были заслушаны доклады специалиста музейной работы из Нижнего Новгорода Н.Г. Панкрашкиной, сотрудника Политехнического музея М.Э. Смолевицкой, которые рассказали о формировании, хранении и презентации коллекций ВТ. Ю.В. Ревич и В.В. Шилов познакомили аудиторию с непубличными оценками вычислительной техники в высказываниях современников в период 1950-х–1960-х гг.

Фото 1. Наталия Никифорова, Санкт-Петербург (здесь и далее фото Ю.В. Ревича)

На секции «Сети и программные системы» были рассмотрены теоретические и практические вопросы применения ЭВМ в различных народнохозяйственных проектах. Так В.Н. Парамонов (Самара) изучил роль ВТ в промышленной автоматизации в контексте вызовов научно-технической революции позднесоветского периода. С.Л. Мушер и С.В. Бредихин были непосредственными участниками проекта по созданию сети Интернет в Новосибирском Академгородке, и изложили историю его развития. Ю.Е. Поляк представил историю возникновения поисковой системы Yandex.

На конференции в Зеленограде укрепилась тенденция привлечения архивных материалов, углубления и расширения тематики. Впервые были представлены результаты исследований в области становления отечественной научно-технической политики, которое сделало ВТ объектом государственного планирования. Цифровая техника появилась вне больших проектов, но решающий импульс получила в недрах Советского атомного проекта в начале 1950-х годов (доклад В.В. Шилова, И.А. Крайневой, Н.Ю. Пивоварова). Следующий этап в развитии отечественного компьютерного машиностроения – проект ЕС ЭВМ (конец 1960-х – начало 1970-х) – нашел отражение в докладе Ю.С. Ломова «ЕС ЭВМ сквозь призму

отечественной и мировой вычислительной техники». Докладчик отметил, что «в Советском Союзе при всём богатстве реализуемых идей и опыте практического конструирования, признанного во всём мире, не нашлось ведущей организации, которая могла бы создать ряд совместимых ЭВМ различной производительности, как это было сделано фирмой IBM. Многочисленные мощные отечественные компьютерные коллективы отраслевой науки не смогли этого сделать, поскольку каждый из них решал свою задачу. А после того, как проект IBM был не только опубликован, но и подтверждён практическими разработками ЭВМ третьего поколения, Советский Союз, как и все страны, занимающиеся ВТ, вынужден был сверять свои часы на предмет готовности и возможности разрабатывать и производить высокотехнологичную продукцию нового поколения» [7]. Социально-политические и идеологические аспекты информатизации советского общества затронуты были в докладе С.Б. Ульяновой, Н.В. Никифоровой и И.В. Сидорчука (Санкт-Петербург).

Доклады о многочисленных разработках отечественных специалистов, выполненных ими для оборонного и разведывательного ведомств, продемонстрировали те стороны деятельности отечественных специалистов, которые были скрыты в свое время (В.С. Криворученко «Эволюция систем автоматизации научных исследований аэромеханики летательных аппаратов», Б.М. Басок «Первые отечественные программно-аппаратные комплексы моделирования цифровых схем», В.И. Штейнберг «Комплекс БЦВМ АРГОН и элементная база», Ю.В. Романец «История создания первых отечественных шифропроцессоров...» и другие).

Как уже было отмечено, история ЭВМ широко представлена на всех конференциях SoRuCom: доклад, посвященный ЭВМ «Наири» Ереванского НИИ машиностроения подготовили С.Б. Оганджанаян и Т.Г. Гаспарян (Москва), ЭВМ ЕС-1033 – А.У. Ярмухаметов из Казани. Основам и истории создания отечественной модулярной техники был посвящен доклад Б.М. Малашевича, изучению модулярных процессов – доклад С.А. Инютина. Н.Е. Балакирев, Л.Е. Карпов, В.М. Фельдман и А.Е. Ширай в совокупности изложили историю развития архитектурных решений и программного обеспечения МВК «Эльбрус». В докладах, представленных специалистами Казанского завода ЭВМ и КНИТУ-КАИ им. А.Н. Туполева (И.М. Якимов, М.Ш. Бадрутдинова, Л.М. Забирова, В.Ф. Гусев, В.В. Дьячков) постулированы особенности деятельности казанской инженерно-конструкторской школы ЭВМ, охарактеризованы разработки различных средств автоматизации проектирования, подсистем управления потоками данных для ЕС ЭВМ, а также создание промышленной основы разработок и поставок пакетов прикладных программ научно-исследовательского профиля, история создания аппаратно-программных систем криптографической защиты

информации (И.М. Якимов, В.В. Девятков, В.М. Трегубов, М.В. Тумбинская, В.В. Песошин, В.М. Захаров, В.М. Кузнецов). Вопросы применения вычислительной техники в советском народном хозяйстве осветил Р.В. Сусов (МФ МВТУ им. Н.Э. Баумана, Мытищи).

На секции «Программирование» было представлено 7 докладов. Некоторые из них охарактеризованы в тематических блоках данной статьи, объединяющих локальные группы исследователей и затронутые ими проблемы (Казань, Прибалтика). Остается представить другую проблематику секции. А.Е. Недоря констатировал слабое развитие технологии программирования, которая не менялась, по его мнению, на протяжении последних 20 лет. Он выделил уровень «бытового программирования», чем «является программирование человеком своих устройств для решения бытовых задач. А.Н. Терехов изложил историю развития графических технологий программирования в компании Ланит-Терком (Санкт-Петербург). Разработке терминальных устройств – телемониторов – уделено внимание в работе В.А. Китова и А.Н. Чеснокова (Москва). Новосибирцы С.В. Кратов и О.Д. Соколова исследовали историю Фондов алгоритмов и программ.

Пополнилась новыми исследованиями историко-биографическая тематика в области ВТ и программирования. С.С. Михалкович и Ю.С. Налбандян (Ростов-на-Дону) охарактеризовали творческий путь математика и программиста Адольфа Львовича Фуксмана (1937–1979), безвременная кончина которого прервала весьма многообещающее развитие системного программирования в Ростове-на-Дону. С.Б. Оганджанян (Москва) рассказал о выдающемся математике Сергее Никитовиче Мергеляне (1928–2008), который в свое время возглавлял ЕрНИИМ; П.Д. Сафонов и А.К. Поляков (Москва) представили биографию создателя ЭВМ для противоракетного комплекса С300 Евгения Александровича Кривошеева (1932–2006); Р.Н. Парамонова (Самара) показала особенности инженерно-изобретательской деятельности в СССР на примере творческих биографий самарских кибернетиков Юрия Михайловича Горского (1926–2004) и Михаила Александровича Ханина (1927 г.р.). В.М. Соболь (Москва) посвятил свой доклад Владимиру Николаевичу Березину (1930–2007) – начальнику Научно-технического центра систем передачи данных (АО НИИАА), д.т.н., профессору, который занимался разработкой системных программ проекта «Алмаз» в интересах развития средств управления ПВО в1970-е гг. сравнительный анализ биографических хроник выдающихся создателей ЭВМ С.А. Лебедева и И.С. Брука дал В.Н. Захаров. Ю.С. Владимирова проанализировала исследования Н.П. Брусенцова в области трехзначной логики.

Наши зарубежные коллеги проанализировали состояние телекоммуникаций и вычислений в Латвии (Р. Балодис, И. Опмане), формирование информационных потоков в области микроэлектроники в Польской республике с помощью научно-технической разведки (М.

Сикора). Белорусские коллеги (Г.К. Столяров, М.Е. Неменман, М.С. Марголин) поделились воспоминаниями о работе минских математиков и программистов, которые разрабатывали программное обеспечение ЭВМ «Минск», о чем сообщалось в докладе И.А. Крайневой и Л.В. Городней (Новосибирск), ими же совместно с А.Г. Марчуком исследовано становление центров программирования в странах Балтии.

Некоторое количество докладов было посвящено истории организаций, которые внесли вклад в подготовку специалистов в различных областях отрасли ВТ, а также являлись свидетельствами институализации основных направлений развития средств ВТ, микроэлектроники, инженерии, программирования, использования техники и ПО (И.И. Дзегеленок, И.И. Ладыгин, А.К. Поляков на примере Московского энергетического института; В.Н. Зенин, Ю.В. Рогачев на примере НИИВК им. М.А. Карцева; М.Б. Игнатьев – Международного института кибернетики и артоники; В.А. Китов, Н.И. Кротов – Вычислительного центра Госплана, Проблемной лаборатории ЭВМ ГИФТИ в Нижнем Новгороде – М.Я. Эйнгорин). Историю Новосибирского филиала ИТМиВТ, созданного по программе «пояса внедрения» в 1970-е годы в новосибирском Академгородке, впервые представили Н.А. Черемных и Г.В. Курляндчик (Москва, Санта-Клара).

Фото 2. Мирослав Сикора, Польша *Фото 3. Станислав Михалкович, Ростов-на-Дону*

На завершающем конференцию круглом столе участники конференции заслушали доклад Т.М. Александриди (Москва), посвященный 90-летию со дня рождения известного ученого и инженера в области вычислительной техники и автоматизированных систем управления Н.Я. Матюхина. Кроме того прозвучал ряд докладов, посвященных 50-летию советского суперкомпьютера БЭСМ-6 (А.Н. Томилин, Москва; М.В. Тумбинская, Казань). 55-летию Зеленоградского центра микроэлектроники был посвящен доклад Б.М. Малашевича (Зеленоград).

Необходимо отметить возросший уровень научного содержания докладов, представленных некоторыми инженерно-техническими специалистами, расширение тематики конференции в рамках ее устоявшейся организационной структуры. С момента проведения в июле 2006 года первой международной конференции «История вычислительной техники и ее программного обеспечения в России и странах бывшего СССР: история и перспективы» сформировалось сообщество SoRuCom, состоящее из активных исследователей, обеспокоенных проблемой сохранения наследия в области вычислительной техники и ее программного обеспечения, курирующих вопросы сбора и обработки уникальных научных материалов, публикации аналитических материалов, повышения качества обучения по истории информатики, развития информационных технологий в России.

Помимо организации и проведения конференций сообщество SoRuCom активно развивает сайт Виртуального компьютерного музея, публикует книги и статьи в ведущих рецензируемых журналах, в том числе на английском языке при поддержке Международной федерации по обработке информации IFIP и Международного Института инженеров электротехники и электроники IEEE, проводит локальные семинары и конференции (ИСИ СО РАН, ИНИОН РАН, Политехнический музей, Музей истории вычислительной техники Казанского завода ЭВМ (Казань), ИВМиМГ СО РАН, СПИИРАН, ИПИ РАН и др.), поддерживает контакты с зарубежными историками информатики Л. Грэхэмом, С. Геровичем, П. Джозефсоном, Д. Петерсом, К. Татарченко и другими. Происходит обмен идеями, литературой, Музей истории компьютеров из Маунтин Вью (Computer History Museum, CA, USA) и Российский фонд фундаментальных исследований (РФФИ) неизменно оказывают финансовую поддержку конференции. Членами сообщества создаются и поддерживаются электронные ресурсы по истории информатики, высоко оцененные международным сообществом: электронный архив академика А.П. Ершова

(http://ershov.iis.nsk.su), сайт Новосибирского филиала Института точной механики и вычислительной техники АН СССР (http://nfitmivt.ru) и др.

Вместе с тем приходится констатировать, что конференция не охватывает всей проблематики народнохозяйственных применений ЭВМ (экономика, сфера обслуживания). Участие зарубежных специалистов на конференции не слишком велико (порядка 5-6 человек), хотя в состав программного комитета они охотно входят. Их непосредственное участие в данной конференции зависит от личного настроя, интереса к проблематике и стремления к живому общению. В Зеленограде, отмечавшем 55-летие своего основания как центра микроэлектроники, конференция не вызвала того резонанса, на который могла бы рассчитывать. Принимающий вуз не проявил к ней должного интереса, не рекомендовал своим студентам включиться в ее работу. Префектура Зеленограда также осталась к ней равнодушна. Экскурсия в музей «Ангстрема» – одного из отечественных флагманов микроэлектроники вызвала скорее сожаление, чем гордость, поскольку продемонстрировала упадок интереса к отечественному научно-техническому наследию со стороны администрации предприятия. Тем не менее, участники конференции искреннее благодарны принимающей стороне – Национальному исследовательскому университету МИЭТ в Зеленограде за предоставленную возможность работать в стенах института.

3. Решение конференции

1. Отметить высокий уровень представленных докладов, тематика которых была актуальной для истории отечественной науки.
2. Выразить благодарность Московскому институту электронной техники за поддержку конференции.
3. Выразить благодарность Российскому фонду фундаментальных исследований, Музею истории вычислительной техники в Маунтин Вью (США) за поддержку конференции.
4. Опубликовать избранные доклады участников конференции на английском языке в электронных изданиях IEEE по представлению экспертного совета.
5. Отметить, что, несмотря на выполнение конференцией своей миссии по аккумулированию воспоминаний создателей ВТ и систем программирования, наступает время перехода от фактографии к аналитическим исследованиям и, тем самым, к повышению качества аргументации за счет обращения к архивам, привлечению профессиональных историков к проблематике конференции, преподаванию элементов истории ИТ и ВТ в профильных вузах.

6. Считать необходимым в дальнейшем при изучении истории информатики и представлении результатов обращать особое внимание на исследования по социальной истории науки, биографические исследования, расширить тематику конференции в области развития системного программирования, прикладного программирования, производства ВТ отдельными предприятиями СССР, информационной безопасности.
7. Рекомендовать Программному комитету как можно шире ознакомить общественность с результатами конференции через публикацию ее материалов на сайте конференции, сайте Виртуального компьютерного музея, сайте IEEE Russian chapter и др.
8. Сформировать рабочую группу из участников конференции по изучению вопроса преподавания истории информатики в вузах России (отв. И.А. Крайнева, М.В. Тумбинская).
9. В связи с реформой РАН участники конференции выражают обеспокоенность судьбой Научного архива СО РАН, который в настоящее время практически не имеет ведомственной принадлежности, закрыт для исследователей. Это уникальное собрание документов должно быть сохранено и доступно пользователям.
10. Провести очередную конференцию в 2020 году.

Список литературы

1. Быстродействующая вычислительная машина М-2/ под. ред. И.С. Брука. М.: ГИТТЛ, 1957. 228 с.
2. Виртуальный компьютерный музей http://www.computer-museum.ru/
3. Ершов А.П., Шура-Бура М.Р. Становление программирования в СССР/ изд. 2-е, доп. Новосибирск, 2016. 78 с. (первое издание вышло в 1976 г.).
4. Китов В. А., Трояновский В. М. 2-я Международная конференция по истории отечественной вычислительной техники и информатики SoRuCom-2011 // ВИЕТ, 2012. № 3. С. 166–168.
5. Конференция «Пути развития советского математического машиностроения и приборостроения». Пленарные заседания. Москва, 12–17 марта 1956 г. М.: ВИНИТИ, 1956. 64 с.
6. Криницкий Н.А. Основные этапы развития вычислительной техники и методов программирования// История информатики в России. Ученые и их школы. М.: «Наука», 2003. С. 183–192.
7. Ломов Ю.С. ЭВМ сквозь призму отечественной и мировой вычислительной техники// Сборник Трудов SoRuCom-2017. М., 2017. С. 185.

8. Ляпунов А.А. Математические исследования, связанные с эксплуатацией электронных вычислительных машин. Математика в СССР за 40 лет. М.: Государственное изд-во физико-математической литературы. 1959. С. 861–862.
9. Михайлов Г.А., Шитиков Б.Н., Явлинский Н.А. Цифровая электронная машина ЦЭМ-1// Проблемы кибернетики. Вып.1. М.: ГИМЛ, 1958. С.190–202.
10. Томилин А.Н., Крайнева И.А., Трегубов В.М., Тумбинская М.В. Третья международная конференция «История вычислительной техники и ее программного обеспечения в России и странах бывшего СССР: история и перспективы» (SoRuCom-2014)// ВИЕТ, 2015. Т. 36. № 1. С. 173–180.
11. Труды конференции SoRuCom-2017 в электронном виде доступны по адресу http://www.iis.nsk.su/files/news/sorucom_2017-6.pdf
12. Шура-Бура М.Р. Программирование// Математика в СССР за сорок лет, том I. М.: Государственное изд-во физико-математической литературы, 1959. С.779–886.

УДК 004.85

Логико-вероятностный метод управления модульными роботами

Демин А.В. (Институт систем информатики СО РАН)

В данной работе представлен логико-вероятностный метод адаптивного управления модульными системами, основанный на использовании свойств функциональной схожести модулей и логико-вероятностного алгоритма направленного поиска правил. Предложенный метод основан на совместном обучении управляющих модулей, начиная с поиска общих для всех модулей управляющих правил и закачивая их последующей спецификацией в соответствии с идеями вероятностного логического вывода. С помощью интерактивного 3D-симулятора были проведены успешные эксперименты с четырьмя виртуальными моделями роботов. Экспериментальные исследования показали, что предложенный подход достаточно эффективный и может быть использован для управления модульными системами с большим количеством степеней свободы.

Ключевые слова: *система управления, обнаружение закономерностей, извлечение знаний.*

1. Введение

Гиперизбыточные робототехнические системы, характеризующиеся большим числом степеней свободы, обладают рядом существенных преимуществ перед традиционными системами. Типичные представители гиперизбыточных систем – это змеевидные и многоногие роботы, многозвенные манипуляторы, модульные роботы и др. Гиперизбыточность наделяет подобные системы более универсальными свойствами, позволяя им решать больший круг задач, а также повышает их отказоустойчивость. Одними из наиболее интересных и перспективных представителей гиперизбыточных систем являются так называемые «модульные роботы». Это класс робототехнических систем, конструкции которых состоят из множества простых однотипных модулей [16,19]. Данное направление робототехники активно развивается в последнее время и сулит целый ряд новых возможностей, начиная с создания роботов-трансформеров, меняющих свою конструкцию для решения конкретных задач, и заканчивая удешевлением производства за счет использования однотипных модулей.

Однако развитие и использование гиперизбыточных робототехнических систем сталкивается с серьезными трудностями, связанными со значительной сложностью управления подобными системами. Наличие большого количества степеней свободы делает невозможным применение традиционных подходов к созданию систем управления путем прямого задания сенсорно-моторных функций человеком-разработчиком. Поэтому становиться актуальным разработка способов автоматического порождения системы управления на основе различных моделей обучения.

В мировой практике в области адаптивного управления гиперизбыточными и модульными системами чаще всего предлагаются решения, основанные на использование популяционных методов (эволюционные методы, методы роя частиц и т.д.) в интеграции с другими известными методами машинного обучения (Reinforcement Learning, нейронные сети и др.) [7,8,11-13,17,18]. Однако эволюционные методы имеют серьезные ограничения, связанные с необходимостью наличия популяции роботов, что не позволяет проводить обучение и адаптацию в режиме реальной работы [14]. Общим же недостатком подобных решений является невозможность обучения в режиме реальной работы и слабая масштабируемость относительно увеличения сложности системы (количества степеней свободы). В целом, следует отметить, что в настоящее время пока еще не предложено достаточно универсального решения задачи адаптивного управления гиперизбыточными системами.

В наших работах предлагается альтернативный подход к созданию обучающихся систем управления для модульных роботов, основанный на использовании логико-вероятностных методов извлечения знаний из данных и эксплуатации свойств функциональной симметрии элементов конструкции роботов [2-5,9]. В соответствии с данным подходом управляющие правила системы описываются при помощи языка логики первого порядка, что позволяет использовать логико-вероятностные методы извлечения знаний из данных для обнаружения эффективных правил управления в статистических данных о взаимодействии системы с окружающим миром. В качестве основного пути для преодоления проблемы большого числа степеней свободы предлагается идея использования функциональной симметрии элементов системы, что позволяет существенно сократить пространство поиска управляющих правил за счет использования одних и тех же правил для схожих по своим функциям модулей. Интеграция логико-вероятностного подхода и свойств функциональной симметрии позволили разработать специальный метод поиска управляющих правил, который в первую очередь пытается найти правила, общие для всех модулей, а уже затем специфицировать их для каждого конкретного модуля в отдельности. Эффективность подхода предлагается оценить на примере обучения типичных представителей простейших гиперизбыточных

модульных роботов: змеевидного робота, многоногого робота, хоботовидного манипулятора и многоногого робота с двумя типами конечностей.

2. Система управления

Для создания системы управления модульными роботами предлагается использовать модель с сетевой структурой, в которой базовым элементом управления является обучаемый логический нейрон. Задачей каждого логического нейрона является управление отдельным модулем робота.

Логические нейроны функционируют в дискретном времени $t = 0,1,2,...$. Каждый нейрон содержит некоторый набор входов $input_1,...,input_k$, принимающих действительные значения, и один выход $output$, принимающий значения из заранее заданного набора $\{y_1,...,y_m\}$. В каждый момент времени t на входы нейрона подается входящая информация путем присвоения входам некоторых действительных значений $input_1 = x_1,...,input_k = x_k$, $x_1,...,x_k \in \mathbf{R}$. Результатом работы нейрона является выходной сигнал $output = y$, $y \in \{y_1,...,y_m\}$, принимающий одно из возможных значений $\{y_1,...,y_m\}$.

После того, как отработают все нейроны сети, от внешней среды поступает награда. Функция награды задается в зависимости от конечной цели и служит оценкой качества управления. Задачей системы управления является обнаружение таких закономерностей функционирования нейронов, которые бы обеспечивали получение максимальной награды.

Множество закономерностей, определяющих работу нейронов, предлагается искать в виде логических закономерностей с оценками, имеющих следующий вид:

$$\forall i(P(i), X_1(i),..., X_m(i), Y(i) \to r), \qquad (1)$$

где $i = 1,..,n$ – переменная по объектам - индексам нейронов.

$X_j(i) \in \mathbf{X}$ – предикаты из заданного множества входных предикатов $\mathbf{X}$, описывающих входы j нейронов N_i $(i = 1,..,n)$. К примеру, в простейшем случае данные предикаты могут быть заданы как $X_j(i) = (input_k(i) = x_r)$, где x_r – некоторые константы из области значении входящих сигналов, которые могут быть заданы, к примеру, путем квантования диапазона возможных значений соответствующих входов нейронов.

$Y_j(i) \in \mathbf{Y}$ – предикаты из заданного множества выходных предикатов $\mathbf{Y}$, описывающих выходы нейронов N_i $(i = 1,..,n)$ и имеющих вид $Y_j(i) = (output(i) = y_r)$, где y_r – некоторые константы из набора значений выходных сигналов.

$P(i) \in \mathrm{P}$ – предикаты из множества предикатов P, имеющих вид $(i = j)$, где $j = 1,..,n$, смысл которых – сужать область применения правил вида (1) до конкретных нейронов.

r – награда, максимизация которой является постоянной задачей нейрона.

Данные закономерности предсказывают, что если на вход нейрона N_i, $i = 1,..,n$ будут поданы сигналы, удовлетворяющие входным предикатам $X_1(i),...,X_m(i)$ из посылки правила, и нейрон подаст на свой выход сигнал, указанный в выходном предикате $Y(i)$, то математическое ожидание награды будет равно некоторой величине r.

Отдельно отметим, что если какой-либо нейрон N_j имеет вход, специфичный только для этого нейрона, то предполагаем, что предикат $X(i)$, описывающий этот вход, будет принимать значение «0» для всех $i \neq j$, т.е. для всех других нейронов. Аналогично, если выход какого-либо нейрона N_j может принимать некоторое значение y, характерное только для этого нейрона, то соответствующий выходной предикат $(output(i) = y)$ также будет принимать значение «0» для всех $i \neq j$.

Поясним необходимость введения множества предикатов P. В том случае, если правила (1) не содержит предикатов из P, то они будут иметь вид $\forall i(X_1(i),...,X_m(i),Y(i) \rightarrow r)$ и будут описывать закономерности, общие для всех нейронов N_i, $i = 1,..,n$. Добавление в посылку правила предиката из P автоматически суживает область применения правила до конкретного нейрона. Таким образом, правила, содержащие предикаты из P, описывают закономерности, специфичные для конкретных нейронов. Также следует отметить, что сужение области применимости правил (1) может происходить не только за счет предикатов из P, но также за счет входных либо выходных предикатов из $\mathbf{X}$ и $\mathbf{Y}$, описывающих специфичные входы либо выходы конкретных нейронов.

Для нахождения закономерностей вида (1) предлагается использовать алгоритм, основанный на идеях семантического вероятностного вывода, описанного в работах [1,5]. При помощи данного алгоритма анализируется множества данных, хранящих статистику работы нейронной сети (вход-выход нейронов и полученная награда) и извлекаются все статистически значимые закономерности вида (1).

Рассмотрим алгоритм поиска закономерностей подробнее.

Для простоты в дальнейшем будем записывать правила (1) в упрощенном виде:

$$P, X_1,..., X_m, Y \rightarrow r. \tag{2}$$

Введем ряд формальных определений.

Перепишем правило (2) в виде $\mathrm{A}, Y \to r$, где A обозначает множество предикатов из множеств P или $\mathbf{X}$, входящих в посылку правила, т.е. $\mathrm{A} = \{P, X_1, ..., X_m\}$.

Определение 1. Подправилом правила $R_1 = \mathrm{A}_1, Y \to r$, $\mathrm{A}_1 \neq \varnothing$ будем называть любое правило $R_2 = \mathrm{A}_2, Y \to r$, для которого выполнено условие $\mathrm{A}_2 \subset \mathrm{A}_1$.

Определение 2. Закономерностью будем называть правило вида (2), удовлетворяющее следующим условиям:

1. Математическое ожидание награды r для правила определено.
2. Математическое ожидание награды r правила строго больше математических ожиданий награды для каждого из его подправил.

Определение 3. Правило $R_2 = \mathrm{A}_2, Y \to r$ будем называть уточнением правила $R_1 = \mathrm{A}_1, Y \to r$, если для него выполняется одно из условий

1. $\mathrm{A}_2 = \mathrm{A}_1 \cup X$, где $X \in \mathbf{X}$ и $X \notin \mathrm{A}_1$, либо
2. $\mathrm{A}_2 = \mathrm{A}_1 \cup P$, где $P \in \mathrm{P}$ и A_1 содержит только предикаты из $\mathbf{X}$.

Т.е. R_2 будет являться уточнением R_1, если оно получено либо добавлением в посылку R_1 нового входящего предиката из $\mathbf{X}$, либо добавлением любого предиката из P, если в R_1 нет предикатов данного вида. Таким образом, суть операции уточнения состоит в конкретизации области применения правила либо путем добавления новых входных признаков либо путем сужения применимости правила до конкретного нейрона.

Суть алгоритма обнаружения закономерностей заключается в последовательном уточнении правил, начиная с правил единичной длины, путем добавления в посылку правил новых предикатов с последующей проверкой уточненных правил на принадлежность к вероятностным закономерностям. По существу реализуется направленный перебор правил, позволяющий существенно сократить пространство поиска. Сокращение перебора достигается за счет использования эвристики, которая заключается в том, что, начиная с момента, когда длина посылки правил достигает некоторой заданной величины, называемой глубиной базового перебора, начинается последовательное уточнение только тех правил, которые являются вероятностными закономерностями.

Перейдем к описанию алгоритма, реализующего поиск множества закономерностей, определяющих работу нейронов. Обозначим через $Spec(RUL)$ – множество всех возможных уточнений всех правил из RUL, где RUL – произвольное множество правил вида (2). Входным параметром алгоритма также является глубина базового перебора d, где $d \geq 1$ – натуральное число.

1. На первом шаге генерируем множество RUL_1 всех правил единичной длины, имеющих следующий вид $Y \to r$, $Y \in \{Y_1,...,Y_k\}$. Все правила RUL_1 проходят проверку на выполнение условий принадлежности к закономерностям. Правила, прошедшие проверку, будут являться закономерностями. Обозначим через REG_1 множество всех закономерностей, обнаруженных на первом шаге.

2. На шаге $k \le d$ генерируется множество $RUL_k = Spec(RUL_{k-1})$ всех уточнений правил, сгенерированных на предыдущем шаге. Все правила из RUL_k проходят проверку на выполнение условий принадлежности к закономерностям. Обозначим через REG_k полученное множество закономерностей.

3. На шаге $l > d$ генерируется множество $RUL_l = Spec(REG_{l-1})$ уточнений всех закономерностей, обнаруженных на предыдущем шаге. Все правила из RUL_l проходят проверку на выполнение условий принадлежности к закономерностям. Обозначим REG_l – множество всех закономерностей, обнаруженных на данном шаге.

4. Алгоритм останавливается на шаге $m > d$, когда не обнаружено новых закономерностей $REG_m = \varnothing$.

5. Результирующее множеством закономерностей является объединение всех множеств обнаруженных закономерностей REG_i.

Шаги алгоритма $k \le d$ соответствуют базовому перебору, а шаги $k > d$ – дополнительному перебору.

Оценка математического ожидания награды для правил осуществляется по множеству данных, хранящих статистику работы системы (вход-выход нейронов и полученная награда), следующим образом: $r = \sum_{i \in I} r_i \Big/ |I|$, где I – множество событий, когда правило может быть применено, r_i – награда нейрона для i-го события, $i \in I$.

Преимущество использования семантического вероятностного вывода и правил вида (1) состоит в организации поиска правил таким образом, что сначала будут обнаруживаться правила, общие для всех нейронов, а только затем – более сложные, включающие специфичные для конкретных нейронов правила. В результате, в задачах управления модульными роботами, если хотя бы часть модулей имеет схожие функции, которые можно описать общими правилами, предложенный подход позволяет существенно сократить время поиска решения.

Функционирование нейронной сети в составе системы управления происходит следующим образом. На каждом такте работы сети на входы нейронов поступают входящие сигналы. После чего последовательно для каждого нейрона запускается процедура принятия решения, в процессе которой из множества правил, описывающих работу нейронов, выбираются те, которые применимы к текущему нейрону на текущих входных сигналах. Затем среди отобранных правил выбирается одно правило, прогнозирующее максимальное значение математического ожидания награды r. Далее на выход нейрона подается выходной сигнал $output = y$, указанный в правиле. В начальной стадии функционирования сети, когда множество правил, описывающих работу нейронов, еще пусто, либо когда нет правил, применимых к текущему набору входящих сигналов, выход нейрона определяется случайным образом. После того, как все нейроны сгенерируют свои выходные сигналы, система управления запускает на выполнения все действия, которые были активированы этими сигналами. После выполнения действий от внешней среды поступает награда и осуществляется обучение, в процессе которого ищутся новые и корректируются текущие правила работы в соответствии с предложенным алгоритмом поиска закономерностей.

3. Симулятор

Для проведения экспериментов с предложенной моделью управления был разработан интерактивный 3D-симулятор с графическим интерфейсом. Основное назначение программы – проведение экспериментов по управлению роботами в среде, приближенной к реальному миру. Программа обладает возможностями визуализации виртуальной среды и записью экспериментов в видео-файл. В качестве физического движка в симуляторе используется библиотека Open Dynamic Library (ODE) [15], которая позволяет моделировать динамику твердых тел с различными видами сочленений. Преимуществом данной библиотеке является скорость, высокая стабильность интегрирования, а также встроенное обнаружение столкновений. При помощи данного симулятора было построено четыре модели роботов: змеевидный робот, многоногий робот, хоботовидный манипулятор и многоногий робот с двумя типами конечностей.

Модель змеевидного робота была реализована в симуляторе в виде совокупности шести одинаковых прямоугольных блоков («позвонков»), соединенных вместе при помощи универсальных сочленений (рис. 1а). Все сочленения идентичны и обладают двумя угловыми двигателями («мускулами»), обеспечивающими вращение суставов в вертикальной и горизонтальной плоскостях. Предложенная конструкция, несмотря на

простоту, обеспечивает достаточную гибкость модели и позволяет принимать характерные для биологических змей положения тела.

Вторая модель – многоногий робот представлена в виде конструкции из шести одинаковых модулей, связанных друг с другом жесткими сочленениями (рис. 1b). Каждый модуль имеет пару Г-образных ног с правой и левой стороны соответственно. Таким образом, суммарно робот имеет двенадцать ног-конечностей. Каждая нога соединена с модулем при помощи универсального сочленения, имеющего два угловых двигателя, которые позволяют поворачивать ногу в суставе в горизонтальной и вертикальной плоскостях. В целом, конструкция робота напоминает своим видом биологических многоножек и позволяет реализовать характерные для данного вида способы передвижения.

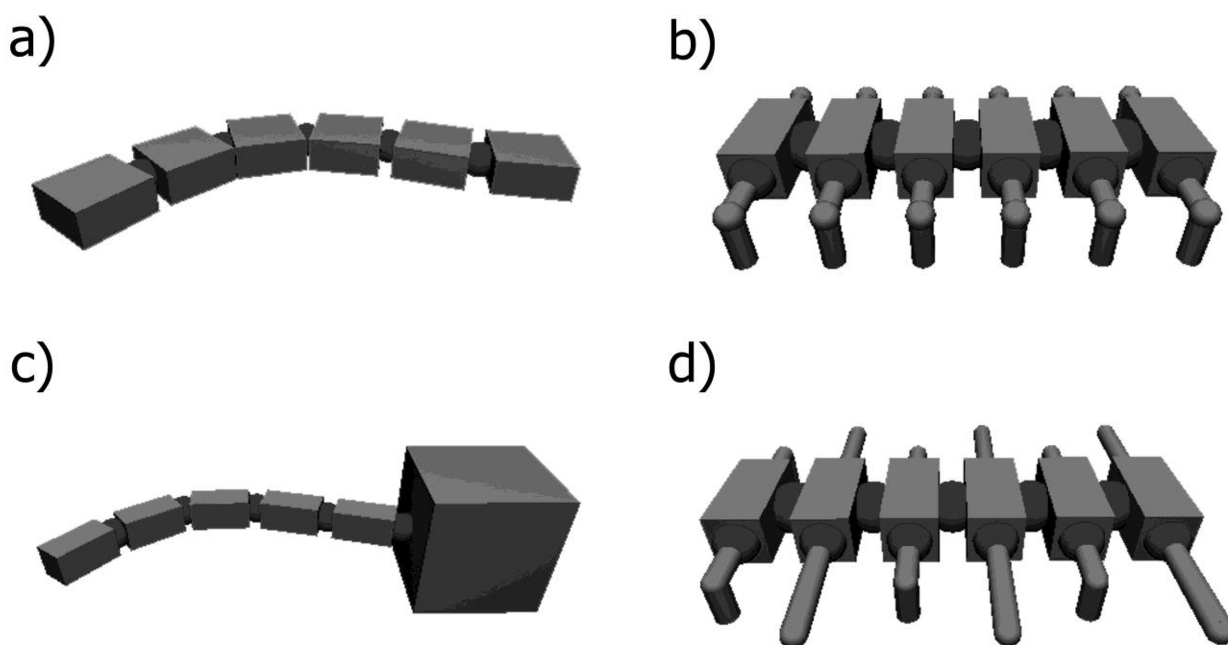

Рис.1. Модели роботов

Модель хоботовидного манипулятора представлена в виде многосекционного «хобота», соединенного универсальным сочленением с массивной неподвижной платформой, реализованной в виде куба (рис. 1c). Сам хобот реализован как последовательность пяти одинаковых прямоугольных блоков, связанных универсальными сочленениями с угловыми двигателями. Размеры блоков и позиции сочленений были подобраны таким образом, чтобы обеспечить достаточную для проведения экспериментов гибкость системы и область достижимости.

Последняя модель представляет собой многоного робота, состоящая из двух повторяющихся типов модулей (рис. 1d). Четные модули имеют пару Г-образных конечностей с правой и левой стороны, способные двигаться только в горизонтальной плоскости. Нечетные модули имеют пару прямых конечностей, способных двигаться только в вертикальной плоскости. Модули поочередно соединены друг с другом посредством жестких сочленений. Всего робот имеет шесть модулей: три модуля с Г-образными конечностями и три – с прямыми. Данная модель робота была разработана специально для того, чтобы проверить возможности предложенной модели успешно обнаруживать эффективные управляющие правила для различных типов модулей.

4. Система управления движением змеевидного робота

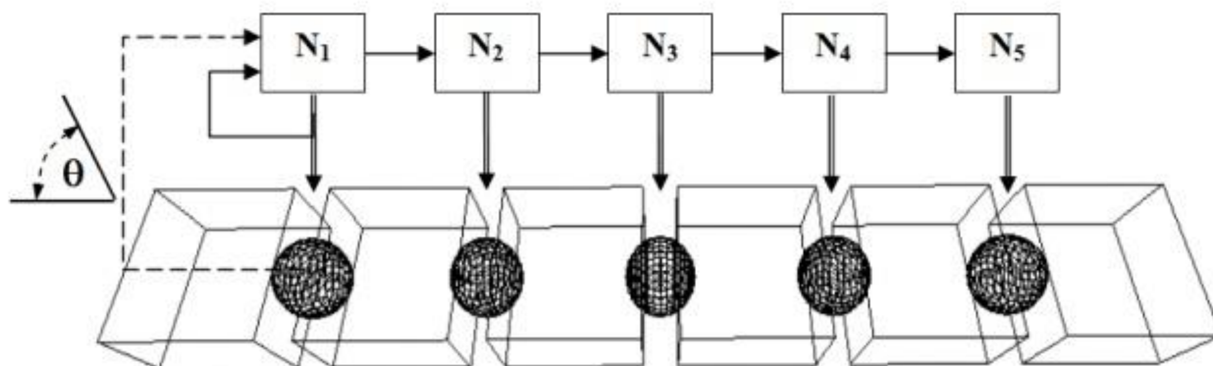

Рис.2. Схема нейронного контура управления движением змеевидного робота

Целью данного эксперимента являлось обучение способам передвижения вперед простейшей модели змеевидного робота (рис. 1a). В предыдущих работах [6,10] нами была предложена модель нейронного контура управления локомоцией нематоды C.Elegans, которая показала высокую эффективность в экспериментах по обучению волнообразному способу передвижения. Схема данного контура предполагала, что головная часть нематоды выступает в качестве источника колебаний, основываясь только на обратной связи от рецептора растяжения. Далее сигнал распространяется по телу нематоды с некоторой временной задержкой, обеспечивая тем самым характерное волнообразное движение. Поскольку конструкция змеевидного робота имеет много общих черт с моделью нематоды, то в данной работе было решено использовать похожую схему нейронного контура для управления движением робота.

В результате был выбран нейронный контур, состоящий из 5 нейронов (рис. 2). Каждый нейрон N_i, $i = 1,...,5$ контролирует один сустав тела робота, подавая активирующие сигналы на угловые двигатели, расположенные в этом суставе. Головной нейрон N_1 получает на вход информацию об углах сгиба между головным и последующим сегментом. Помимо этого на вход нейрона по обратным связям поступает сигнал от его собственного выхода с временной задержкой Δt. Остальные нейроны N_i, $i = 2,...,5$ получают на свой вход только сигнал от выхода предыдущего нейрона N_{i-1} с временной задержкой Δt.

Множество входных и выходных предикатов для нейронов задается путем квантования диапазона возможных значений соответствующих входов и выходов нейрона. Награда для всего нейронного контура управления движением определяется в зависимости от величины скорости, которую разовьет робот на отрезке времени Δt: чем выше скорость – тем больше награда.

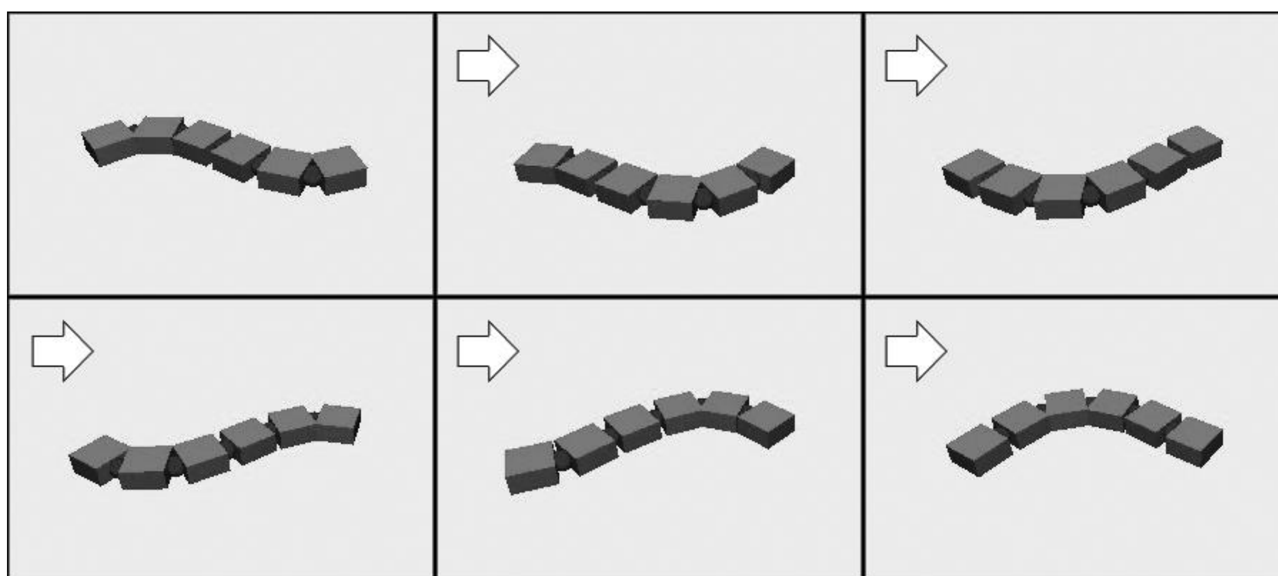

Рис.3. Последовательность движений змеевидного робота при перемещении вперед

Используя 3D-симулятор, был проведен ряд успешных экспериментов по обучению предложенной модели способам передвижения. Как показали результаты экспериментов, системе управления удается стабильно обучаться эффективному способу передвижения вперед, основанному на волнообразном движении туловища в горизонтальной плоскости. Данный способ передвижения является самым распространенным среди биологических змей, а также характерен и для некоторых других животных, к примеру, нематод. На рисунке 3

приведены найденные системой в ходе обучения оптимальные последовательности движений при перемещении вперед.

5. Система управления движением многоногого робота

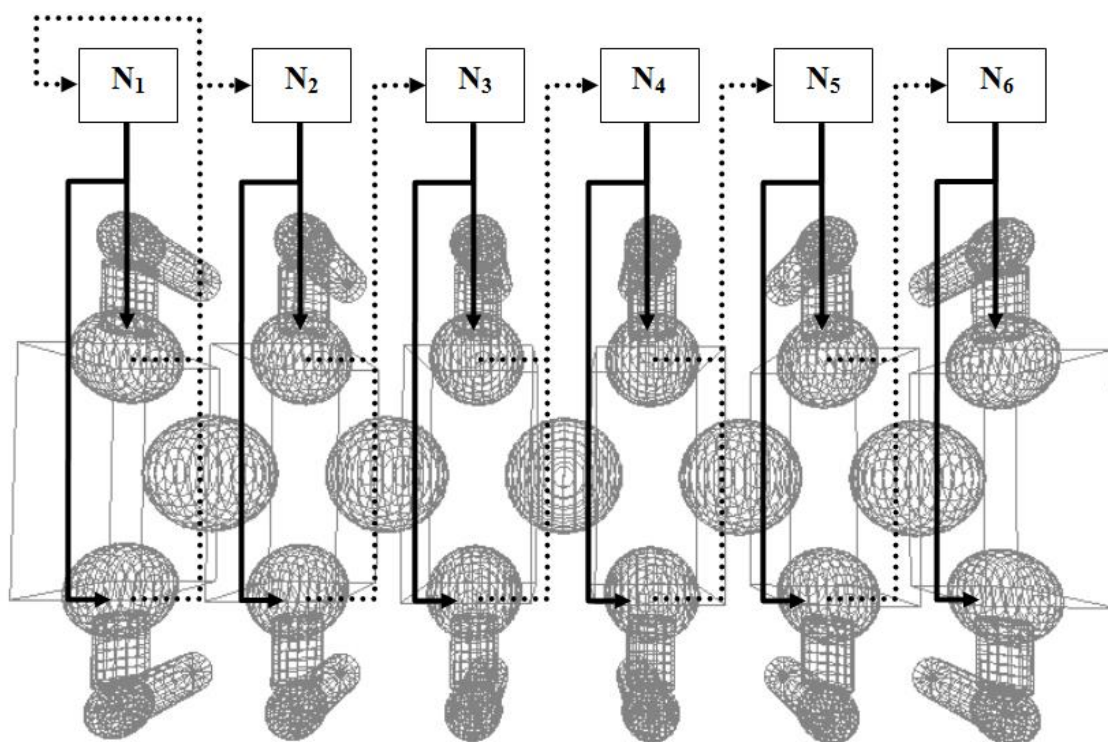

Рис.4. Схема нейронного контура управления движением многоногого робота

В данном эксперименте была поставлена задача обучить способу движения вперед модель многоногого робота (рис. 1b). Для системы управления роботом был выбран нейронный контур, состоящий из шести нейронов – по одному нейрону на каждый модуль робота (рис. 4). Каждый нейрон N_i, $i = 1,...,6$ контролирует движения левой и правой ноги своего модуля, подавая активирующие сигналы на соответствующие угловые двигатели, вращающие конечности в суставе. Чтобы немного упростить задачу, движения правой и левой ног робота были синхронизированы таким образом, что движение одной ноги всегда происходит в противофазе с другой. Т.е., к примеру, движение левой ноги вперед всегда сопровождается движением правой ноги назад. Таким образом, нейрону, по сути, достаточно контролировать движения только одной ноги, поскольку вторая нога будет повторять эти же движения, только в противофазе.

Нейрон первого модуля N_1 получает на вход информацию о положении ног первого модуля. Остальные нейроны N_i, $i = 2,...,6$ получают на свои вход данные о положении ног предыдущего модуля. Информация о положение ноги задается парой углов сгиба конечности в суставе в вертикальной и горизонтальной плоскостях. Как и в предыдущей задаче множество входных и выходных предикатов для нейронов задается путем квантования диапазона возможных значений их входов и выходов. Аналогично, награда для всей системы управления движением определяется в зависимости от величины скорости, которую разовьет робот на отрезке времени Δt : чем выше скорость – тем больше награда.

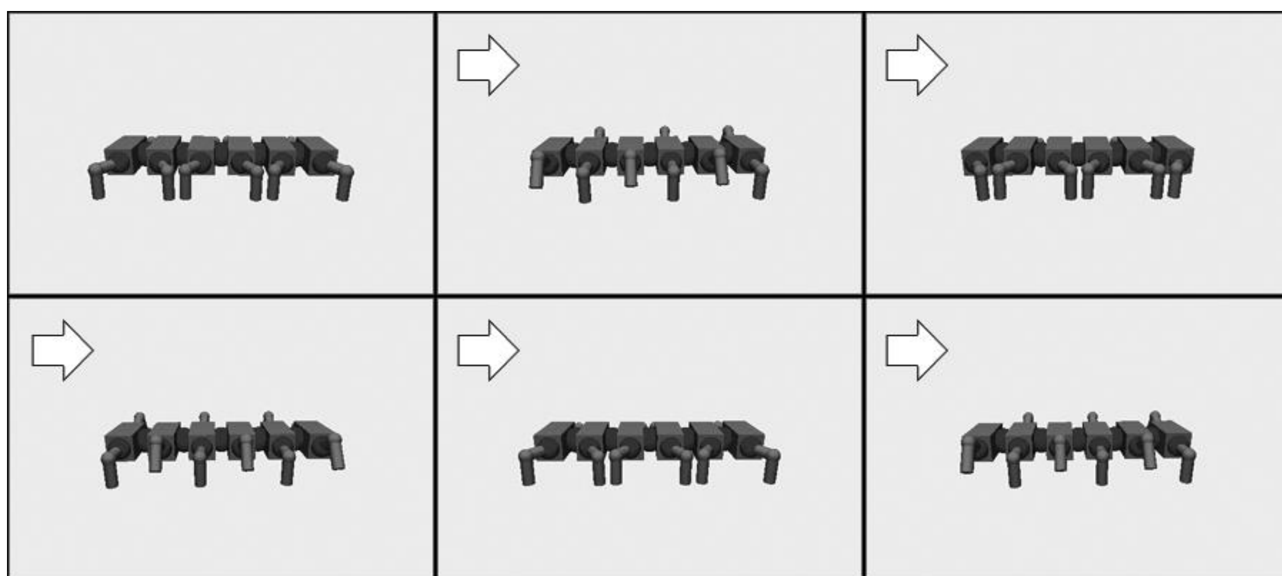

Рис.5. Последовательность движений многоногого робота при перемещении вперед

При помощи 3D-симулятор была проведена серия экспериментов по обучению движению модели многоногого робота. Результаты экспериментов показали, что система управления успешно обнаруживает согласованные движения конечностей, обеспечивающие эффективное перемещение вперед. На рисунке 5 приведен пример оптимальной последовательности движений, найденной в процессе обучения.

6. Система управления хоботовидным манипулятором

В данном эксперименте перед манипулятором (рис. 1c) была поставлена задача захватить цель, которая появляется в случайной позиции в пределах области, достижимой для хобота. В качестве цели выступает сфера радиусом равным длине одного сегмента хобота. Цель считается захваченной, если конец хобота (последний сегмент) окажется внутри сферы. После захвата сфера-цель исчезает и появляется в новой случайной позиции. Таким образом,

эксперимент может продолжаться непрерывно неограниченное время. Задачей системы управления является обнаружение таких правил управления движением хобота, которые бы обеспечивали захват цели в любой доступной для манипулятора позиции.

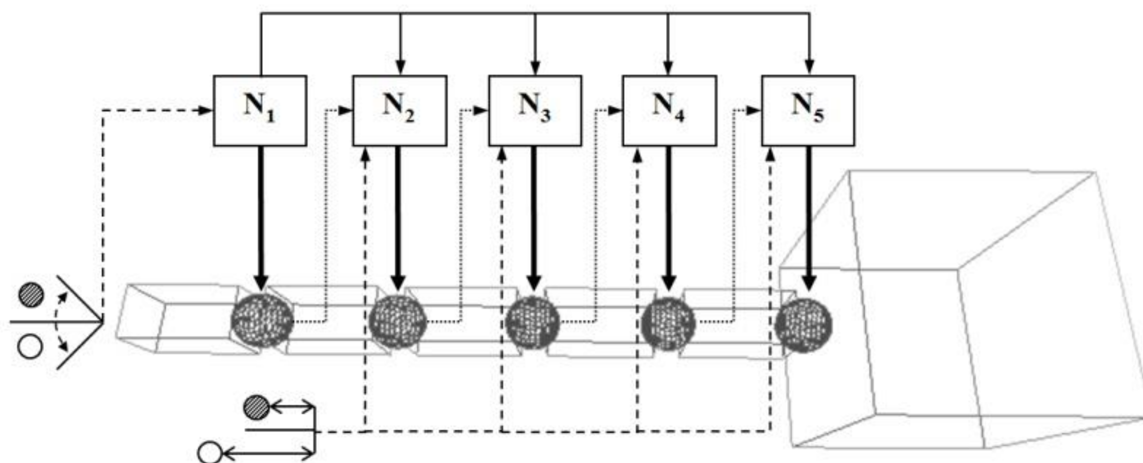

Рис.6. Схема нейронного контура управления хоботовидным манипулятором

Для решения данной задачи была выбрана схема управления из пяти нейронов N_i, $i=1,...,5$, каждый из которых контролирует один сегмент хобота (рис. 6). Нейроны подают активирующие сигналы на угловые двигатели, вызывая тем самым изгиб хобота в соответствующих сочленениях. Поскольку в данной задаче цель может располагаться только на одном уровне, то все движения ограничены только горизонтальной плоскостью, что упрощает задачу.

Первый нейрон N_1 получает на свой вход бинарную информацию о том, по какую сторону относительно хобота располагается цель: справа или слева. Остальные нейроны N_i, $i=2,...,5$ получают на свой вход следующие сигналы: (1) информацию об угле сгиба между контролируемым и предыдущим сегментом; (2) сигнал от выхода предыдущего нейрона N_{i-1} в предшествующий момент времени; (3) бинарную информацию о положении цели по отношение к концу хобота и точки его крепления – результат сравнения расстояний от крепления до цели и от крепления до конца манипулятора. Таким образом, в данном эксперименте манипулятор фактически «слеп», т.е. не видит точное положение цели, а только «ощущает» ее: справа-слева и ближе-дальше.

Награда для системы управления рассчитывается по факту захвата цели манипулятором следующим образом. Пусть цель появилась в новой позиции в момент времени t_0, а манипулятор захватил ее в момент времени t_1. Тогда все действия с момента времени t_0 по момент времени t_1 получать награду в размере $r = 1/(1+(t_1 - t))$, где t – момент времени, для которого рассчитывается награда. Таким образом, наибольшую награду получают самые последние действия, приведшие к достижению цели, и чем глубже в прошлом от момента захвата произошло действие, тем меньшую награду оно получает.

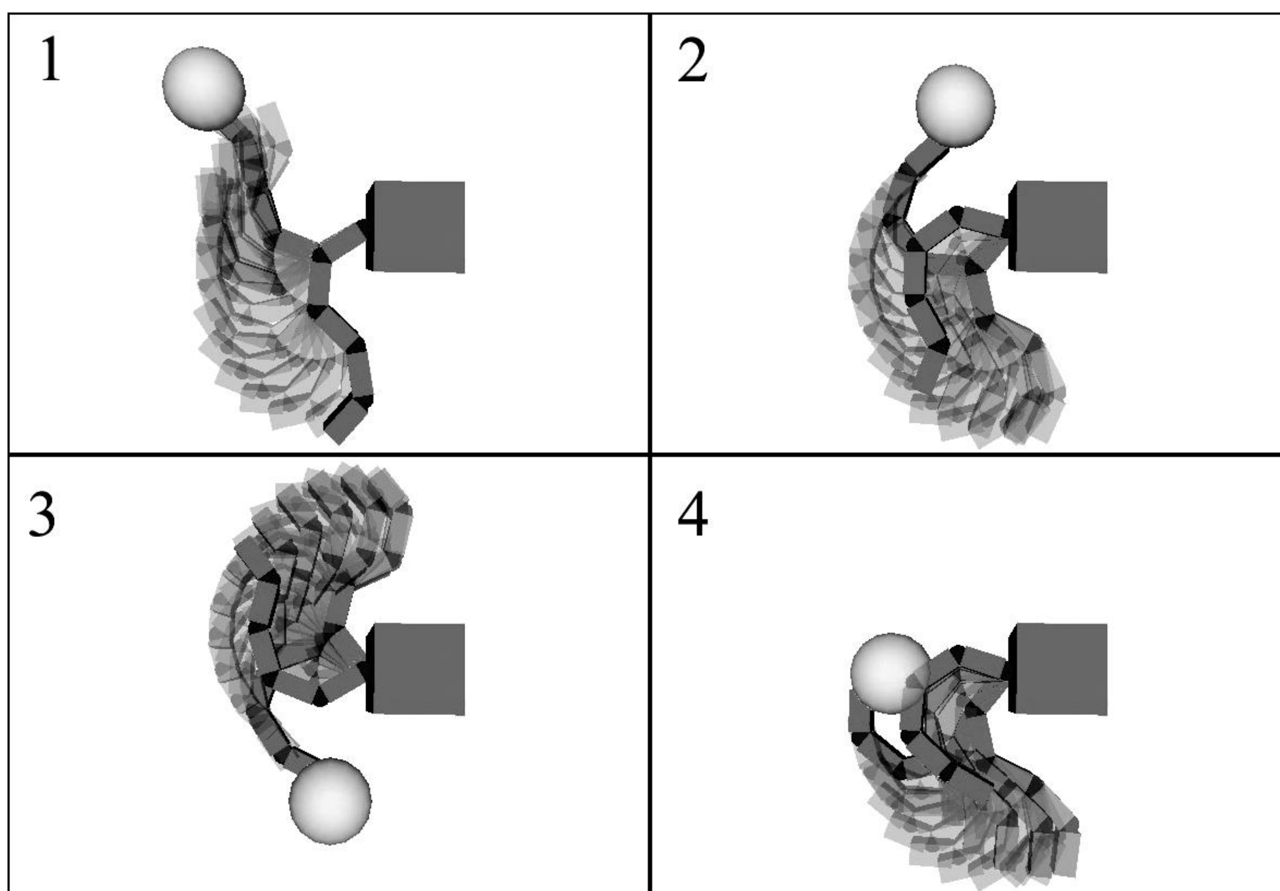

Рис.7. Примеры траекторий движения манипулятора при захвате цели

При помощи 3D-симулятора были проведены эксперименты по обучению системы управления манипулятором захвату целей в любой позиции в пределах зоны достижимости. Как показали результаты экспериментов, в результате обучения системе управления удается обнаружить эффективные правила, обеспечивающие стопроцентный захват целей в указанной зоне. На рисунке 7 приведены четыре примеры траекторий движения уже обученного хобота при захвате цели. Примеры со второго по четвертый показывают, что система управления научилась делать специальные подготовительные движения, чтобы

выйти на удобное положения для совершения захвата. В указанных примерах стартовый изгиб хобота направлен в сторону, противоположную цели, и для того, чтобы достичь цель, манипулятор делает волнообразное движение в другом направлении, в результате которого меняет изгиб тела на противоположное, а затем успешно захватывает цель. Приведенные примеры являются демонстрацией того, что обнаруженные системой правила управления порождают достаточно сложное поведение, направленное на достижение цели.

7. Система управления многоногим роботом с двумя типами конечностей

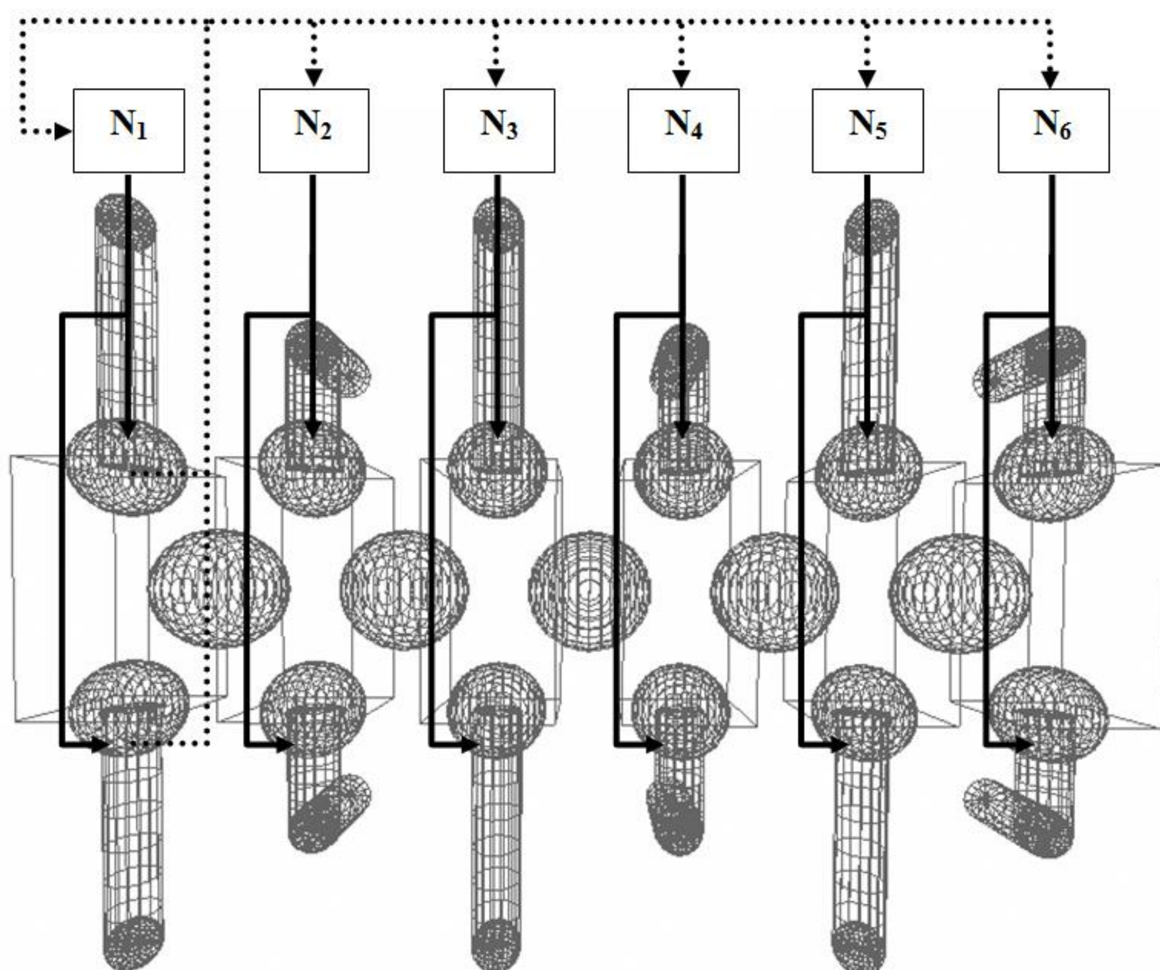

Рис.8. Схема нейронного контура управления многоногим роботом с двумя типами конечностей

Рассмотренные выше эксперименты показали высокую скорость обучения системы управления и хорошую масштабируемость относительно увеличения числа модулей, однако они были ограничены тем, что конструкции роботов состояли из одинаковых модулей.

Основной задачей последнего эксперимента являлась демонстрация применимости предложенного подхода для управления модульными роботами, состоящими из разных типов модулей. С этой целью был поставлен эксперимент по обучению способам передвижения вперед простого многоногого робота, имеющего конечности двух разных типов (рис. 1d). При этом Г-образные конечности могут двигаться только в горизонтальной плоскости, а прямые конечности – только в вертикальной. Очевидно, что данный робот может эффективно двинуться вперед только за счет продвижения Г-образных конечностей назад. Однако поскольку Г-образная конечность может двигаться только в горизонтальной плоскости, то для того, чтобы забросить ее вперед для следующего шага и при этом не сдвинуть робота в обратном направлении, необходимо задействовать прямые конечности, чтобы приподнять робота над землей. В результате, эффективное движение робота возможно только при согласованной работе модулей разных типов. Таким образом, выбранная конструкция робота, несмотря на простоту, является хорошей тестовой моделью для проверки возможностей системы обнаруживать согласованные управляющие правила для различных типов модулей.

Схема нейронного контура, выбранного для управления роботом, состояла из шести нейронов – по одному нейрону на каждый модуль робота (рис. 8). Каждый нейрон N_i, $i = 1,...,6$ контролировал движения левой и правой конечности своего модуля, подавая активирующие сигналы на соответствующие угловые двигатели, вращающие конечности в суставе. Для упрощения задачи движения левой и правой конечностей каждого модуля были синхронизированы таким образом, что одна конечность зеркально повторяла движения другой. Таким образом, каждому нейрону достаточно было выдавать только один активирующий сигнал, чтобы привести в движение сразу обе конечности.

Первый нейрон N_1 получает на свой вход информацию о положении конечностей первого модуля. Эта же информация поступает на входы всех остальных нейронов N_i, $i = 2,...,6$. Таким образом, в данной схеме состояние первого модуля, по сути, можно рассматривать как своеобразный счетчик тактов для всех остальных модулей.

Награда для системы управления рассчитывалась по факту завершения цикла выполнения шага и возврата конечностей первого модуля в исходную точку. Под шагом подразумевается вся последовательность действий, которая была выполнена в промежуток времени между текущим и предыдущим фактами нахождения конечностей в исходном состоянии. В качестве исходной точки было выбрано максимальное вертикальное положение конечностей первого модуля.

Вычисление награды осуществлялось следующим образом. Пусть в текущий момент времени t_1 положение конечностей первого модуля соответствуют исходной точке начала шага. Пусть t_0 – предыдущий момент времени, когда эти конечности находились в исходной точке. Тогда все действия в промежутке времени от (t_0+1) до t_1 будут входить в цикл выполнения шага, а награда для этих моментов времени t, где $(t_0+1) \le t \le t_1$ и $(t_0+1) < t_1$, будет равна $r = S/(t_1-(t_0+1))$. Где S – расстояние, которое преодолеет робот по направлению вперед за этот же промежуток времени (от (t_0+1) до t_1). В случае «пустого» шага, т.е. когда $t_1 = (t_0+1)$ и конечности первого модуля просто остаются в исходной точке два такта подряд, награда для момента времени t_1 устанавливается равной 0. Данная функция награды стимулирует систему управления находить такие последовательности действий, которые бы позволяли преодолевать как можно большее расстояние при совершении как можно меньшего числа действий.

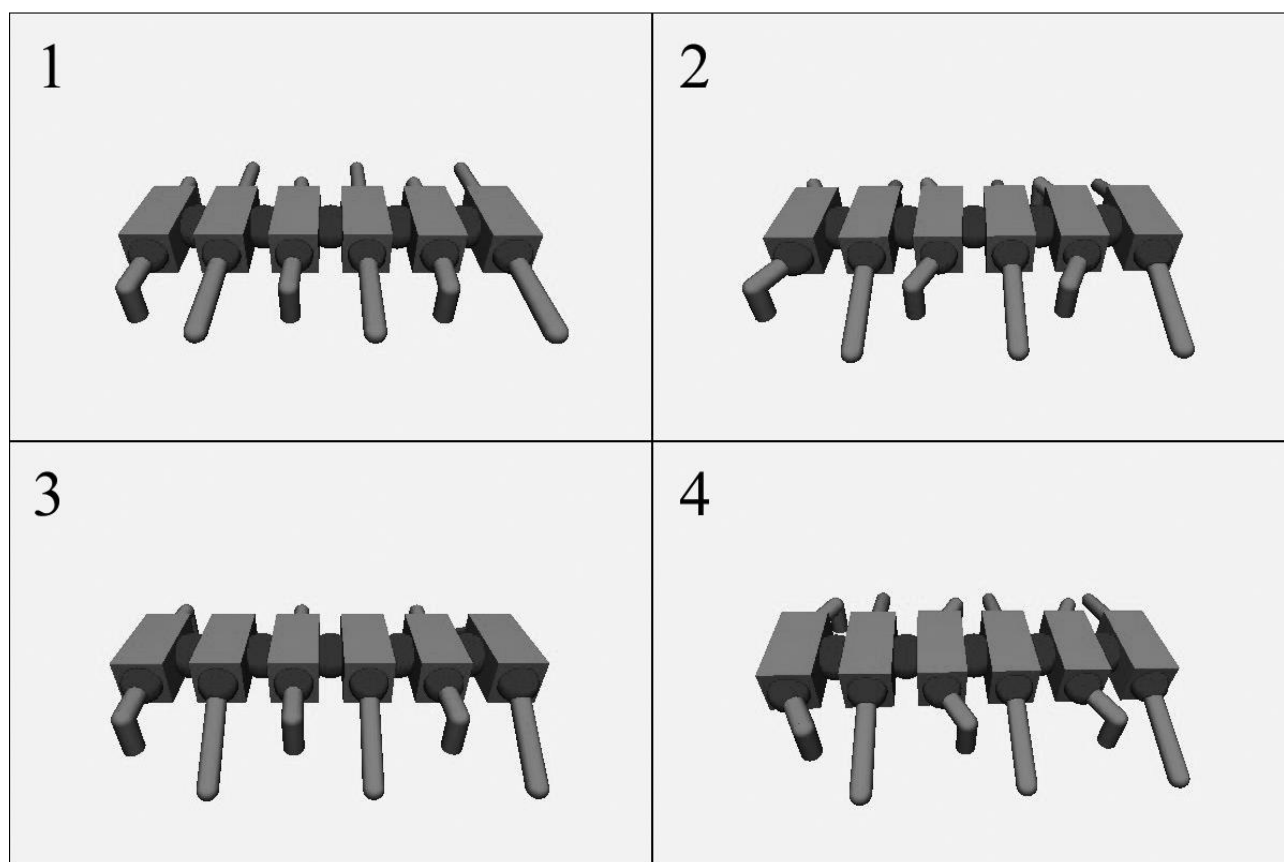

Рис.9. Последовательность движений многоногого робота с двумя типами конечностей при перемещении вперед

Используя симулятор 3D-симулятор, был проведен ряд успешных экспериментов по обучению рассмотренной системы управления способам передвижения. В серии экспериментов системе управления удавалось стабильно обнаруживать правила управления, обеспечивающие согласованные движения конечностей модулей разных типов, приводящие к эффективному перемещению робота вперед. На рисунке 9 приведен пример оптимальной последовательности движений, найденной в процессе обучения.

8. Заключение

В данной работе рассмотрен логико-вероятностный подход к задаче адаптивного управления модульными системами с большим числом степеней свободы. Основными преимуществами предложенного подхода являются способность обучения в режиме реальной работы, основываясь только на опыте взаимодействия с окружающей средой, и высокая скорость обучения, которая достигается за счет эффективного использования свойств функциональной схожести модулей и алгоритма направленного поиска правил. Кроме того, предложенный подход достаточно хорошо масштабируется относительно увеличения числа модулей. В частности, добавление новых сегментов к конструкциям роботов в проведенных экспериментах слабо влияет на эффективность обучения, поскольку не изменяет количество общих для модулей правил. Однако следует отметить, что эффективность предложенного подхода во многом зависит от количества схожих по функциям модулей в конструкции робота. С уменьшением доли схожих модулей преимущества от использования общих правил теряются.

Список литературы

1. Витяев Е.Е. Извлечение знаний из данных. Компьютерное познание. Модели когнитивных процессов. – Новосибирск: НГУ, 2006. – 293 с.
2. Демин А.В. Модель адаптивной системы управления и ее применение для управления движением виртуального робота // Молодой ученый. – 2012. – № 11 (46) – С. 114-119.
3. Демин А. В. Обучающаяся система управления движением для 3D модели многоногого робота // Молодой ученый. — 2015. — №19 (99). — С. 74-78.
4. Демин А.В. Обучение способам передвижения виртуальной модели змеевидного робота // Молодой ученый. – 2014. – № 19 (78) – С. 147-150.
5. Демин А.В., Витяев Е.Е. Логическая модель адаптивной системы управления // Нейроинформатика. – 2008. – Т. 3. – № 1. – С. 79-107.
6. Демин А.В. Обучающаяся модель управления хемотаксисом нематоды C.Elegans // Нейроинформатика. – 2013. – Т. 7. – № 1. – С. 29-41.

7. Bongard J.C. Evolutionary Robotics // Communications of the ACM. – 2013. – Vol. 56. – No. 8. – pp. 74-83.
8. Daoxiong Gong, Jie Yan, Guoyu Zuo. A Review of Gait Optimization Based on Evolutionary Computation // Applied Computational Intelligence and Soft Computing. – 2010. – vol. 2010. – Article ID 413179. – 12 p.
9. Demin A.V. Logical Model of the Adaptive Control System Based on Functional Systems Theory // Young Scientist USA. Applied science. – Auburn, Washington, 2014. – pp. 113-118.
10. Demin A.V., Vityaev E.E. Learning in a virtual model of the C. elegans nematode for locomotion and chemotaxis // Biologically Inspired Cognitive Architectures (2014). – Elsevier, 2014. – V. 7. – pp. 9-14.
11. Ito K., Matsuno F. Control of hyper-redundant robot using QDSEGA // Proceedings of the 41st SICE Annual Conference (2002). – 2002. – V. 3. – pp. 1499-1504.
12. Kamimura A., Kurokawa H., Yoshida E., Tomita K., Murata S., Kokaji S. Automatic locomotion pattern generation for modular robots // Proceedings of 2003 IEEE International Conference on Robotics and Automation. – 2003. – pp. 714-720.
13. Marbach D., Ijspeert A.J. Co-evolution of configuration and control for homogenous modular robots // Proceedings of the eighth conference on Intelligent Autonomous Systems (IAS8). – IOS Press, 2004. – pp. 712-719.
14. Mataric M., Cliff D. Challenges in evolving controllers for physical robots // Robotics and Autonomous Systems. – October 1996. – 19(1). – pp. 67–83.
15. Smith R. Open Dynamics Engine. – URL: http://ode.org/.
16. Stoy K., Brandt D., Christensen D.J. Self-Reconfigurable robots: an introduction // Intellegent robotics and autonomous agents series. – MIT Press, 2010. – 216 p.
17. Tanev I., Ray T., Buller A. Automated Evolutionary Design, Robustness and Adaptation of Sidewinding Locomotion of Simulated Snake-like Robot // IEEE Transactions on Robotics. – V.21. – N. 4. – August 2005. – pp.632-645.
18. Valsalam V. K. Miikkulainen R. Modular neuroevolution for multilegged locomotion // In Proceedings of GECCO. – 2008. – pp. 265–272.
19. Yim M.H., Duff D.G., Roufas K.D. Modular reconfigurable robots, an approach to urban search and rescue // 1st International Workshop on Human Welfare Robotics Systems (HWRS2000). – 2000. – pp. 19-20.

УДК 002.53:004.89

Метод поиска информации на основе онтологии

Ахмадеева И.Р. (Институт систем информатики СО РАН)

В статье предлагается метод поиска информации на основе онтологии научной деятельности. Для поиска используются глобальные поисковые системы, которым отправляются автоматически сгенерированные поисковые запросы, включающие названия сущностей онтологии и термины тезауруса. Поисковые запросы формируются таким образом, чтобы найти как можно больше научных ресурсов, релевантных определенной области знаний. При этом результаты поиска, не содержащие информации о научной деятельности, отфильтровываются с использованием онтологии.

Ключевые слова: *онтология, тезаурус, информационный поиск, поисковый запрос*

1. Введение

Несмотря на то, что в настоящее время накоплены огромные объемы информации по различным областям научных знаний, причем значительная ее часть представлена в сети Интернет, ученые пока не имеют удобного содержательного доступа ко всем интересующим их знаниям и данным, в той области, в которой они проводят исследования.

Для решения этой проблемы была предложена концепция и архитектура тематического интеллектуального научного интернет-ресурса (ИНИР) [1], обеспечивающего доступ к систематизированным научным знаниям и информационным ресурсам определенной области знаний и к средствам их интеллектуальной обработки и анализа. Информация в ИНИР доступна ученому в виде сети знаний и данных, как наиболее естественной и удобной форме подачи информации для человека.

Основу системы знаний ИНИР составляет онтология [6], которая вводит формальные описания понятий некоторой области знаний, типов информационных ресурсов и методов их интеллектуальной обработки в виде классов и отношений между ними.

Важным этапом построения ИНИР является наполнение его контента актуальной информацией о реальных объектах моделируемой области, интегрируемых информационных ресурсах и методах и средствах их обработки и анализа. Эта задача довольно трудоемкая и решить ее можно только за счет автоматизации сбора и накопления релевантной информации из сети Интернет.

В данной работе анализируются различные модели информационного поиска, делается обзор исследований, изучающих поведение пользователей в процессе поиска информации, а также предлагается подход к автоматизации поиска ресурсов в Интернете для сбора информации о научной деятельности в определенной области знаний.

Благодарности. Работа поддержана Российским Фондом Фундаментальных Исследований (грант №16-07-00569).

2. Особенности поведения пользователей в процессе поиска информации

Большинство информационно-поисковых систем (ИПС) работают в соответствии с традиционной моделью информационного поиска: Google, Yandex, Bing, Yahoo!. Такие системы лучше всего подходят для задач поиска фактической информации, когда известна цель поиска. Традиционная модель информационного поиска состоит из четырех компонентов:

- коллекция документов;
- индекс документов (обычно инвертированный для быстрого поиска);
- информационная потребность пользователя;
- поисковый запрос, сформулированный пользователем.

В такой модели предполагается, что пользователь может выразить свою информационную потребность в виде списка ключевых слов. Обычно это можно сделать при решении задач поиска фактов, навигации, ответов на вопросы. В работе [9] такой поиск называется простым. Ему противопоставляется разведывающий поиск (exploratory search), который используется в задачах получения, интерпретации, интеграции, анализа, синтеза знаний и т.д. Для разведывающего поиска характерны следующие особенности [14]:

- пользователь не знаком с областью, которая его интересует (т.е. он хочет получить некоторую информацию об этой области, чтобы понять, как достичь своей цели),
- пользователь не уверен в способах достижения своей цели,
- пользователь не уверен в своей цели, не знает, что ищет.

В работе [3] была предложена метафора «сбора ягод», которая ближе к реальному поведению людей ищущих информацию, чем традиционная модель информационного поиска. В данной модели поиск рассматривается как итеративный процесс, в котором пользователь в самом начале знает совсем немного об интересующей его области и постепенно, получив новую крупицу информации, переформулирует запрос с учетом нового

знания. И таким образом шаг за шагом в процессе эволюционирующего поиска пользователь «собирает» информацию. Причем в процессе такого поиска может измениться как направление поиска, так и сама цель.

Многие исследователи изучали поведение пользователей в процессе поиска информации. В работе [15] выделяются две стратегии поведения пользователей: исследователи и навигаторы. Для навигаторов характерна последовательность в поведении, а для исследователей – изменчивость. Авторы предполагают, что навигаторы решают простую задачу нахождения фактов, а исследователи – более сложную задачу определения смысла.

В работе [2] анализируется поведение пользователей, когда перед ними стоит сложная задача поиска, и выделяются следующие особенности:

- пользователи формулируют больше поисковых запросов,
- они чаще используют специальные операторы поисковых запросов,
- они тратят больше времени на странице с результатами запроса,
- они формулируют самый длинный запрос в середине поисковой сессии.

Часто в исследованиях опытных пользователей сравнивают с новичками и выделяют следующие отличия: опытные пользователе склонны тратить меньше времени на поисковые задачи [11], реже переформулируют запросы [7], используют более длинные запросы [7], используют более систематическую стратегию уточнения запроса [4].

Таким образом, решение сложной задачи поиска является итеративным процессом, в процессе которого пользователь уточняет свой поисковый запрос в зависимости от полученных результатов. Эта идея использовалась при разработке автоматизированной системы поиска информации о научной деятельности в определенной области знаний.

3. Подход к автоматизации поиска информации

Чтобы анализировать каждый сайт в Интернете, учитывая, что Интернет постоянно растет и изменяется, нужно иметь огромные вычислительные мощности. Это могут себе позволить немногие компании, поэтому необходимо разрабатывать методы поиска, которые позволяют выбирать из всех ресурсов небольшое подмножество для последующего анализа.

В данной статье предлагается метод поиска информационных ресурсов, который использует глобальные поисковые системы (метапоиск [10]), онтологию и тезаурус для генерации поискового запроса и оценки релевантности найденных информационных ресурсов тематике ИНИР. Онтологии и тезаурусы часто используются в задачах информационного поиска [12,13,16]. Обычно они помогают расширить поисковый запрос (например, синонимами), который запросил пользователь [16]. В данной же работе

онтология и тезаурус используются для автоматической генерации поисковых запросов. Понятиям онтологии сопоставляются термины тезауруса, с помощью которых их можно выразить на естественном языке. Соответствующие термины тезауруса используются при построении поисковых запросов и оценке релевантности.

Преимущество использования глобальных ИПС заключается в том, что они индексируют весь Интернет. С другой стороны, базируясь на традиционной модели информационного поиска, они требуют формулирования информационной потребности в виде запроса в текстовом виде. В данном случае информационная потребность подсистемы сбора информации состоит в необходимости получить информацию о научной деятельности в определенной области знаний, еще не представленную в контенте ИНИР.

Процесс поиска в сети Интернет научных ресурсов включает следующие этапы:

- Генерацию поисковых запросов;
- Отправку поисковых запросов ИПС;
- Выбор из списка результатов релевантных ресурсов, и сохранение их в базе данных;
- Анализ ссылок в релевантных ресурсах;
- Уточнение поисковых запросов на основе полученных результатов;

Можно выделить несколько видов информационных потребностей, возникающих в процессе поиска научных интернет-ресурсов.

Во-первых, необходимо искать уже известные факты. Такая необходимость возникает в случаях, когда необходимо проверить на корректность только что найденную информацию либо информацию, уже содержащуюся в контенте ИНИР. Кроме этого, такие ресурсы могут ссылаться на другие ресурсы, потенциально содержащие релевантную информацию. Корректность найденной информации предлагается проверять аналогично работе [5] по количеству найденных результатов на соответствующий запрос.

Во-вторых, информация о некоторых экземплярах онтологии может быть неполной. Например, значения атрибутов экземпляра понятия или его связи с экземплярами других понятий могут отсутствовать в контенте ИНИР.

В-третьих, не все экземпляры классов, описанных в онтологии, представлены в контенте ИНИР. В данном случае требуется найти информацию о таких экземплярах.

Для каждого из этих случаев строятся свои наборы шаблонов генерации запросов. Элементами шаблона могут быть классы, экземпляры, отношения и атрибуты онтологии. Пример шаблона показан на рисунке 1.

При построении поискового запроса по шаблону каждый его элемент связывается с конкретным понятием в онтологии (с учетом ограничений, заданных в шаблоне), после чего формируется список ключевых слов по правилам, указанным в шаблоне.

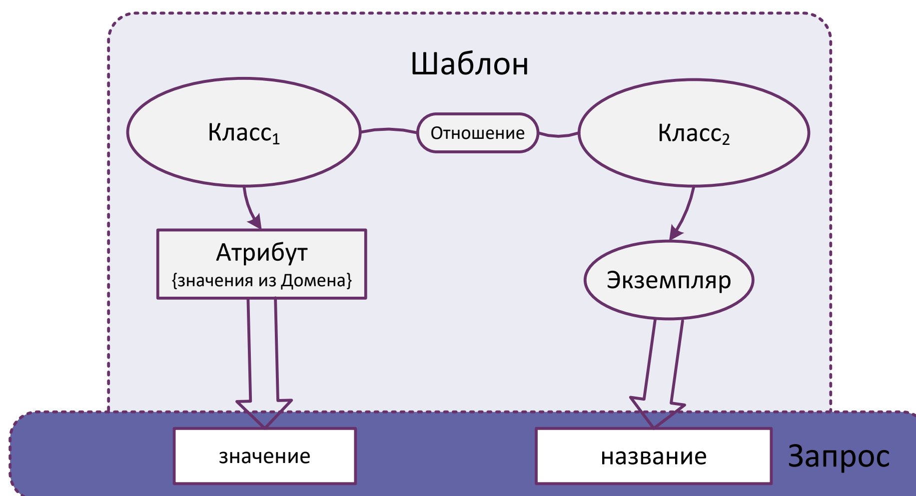

Рис. 1. Пример шаблона поискового запроса.

Например, шаблону, представленному на рисунке 1, соответствует фрагмент онтологии, изображенный на рисунке 2, поскольку он удовлетворяет его ограничениям: два класса, связанные отношением, у одного из которых атрибут должен иметь значения из *Домена*, т.е. у такого атрибута ограничена область допустимых значений. Причем, информация о всех возможных значениях атрибута хранится в онтологии и может быть использована для поиска экземпляров этого класса.

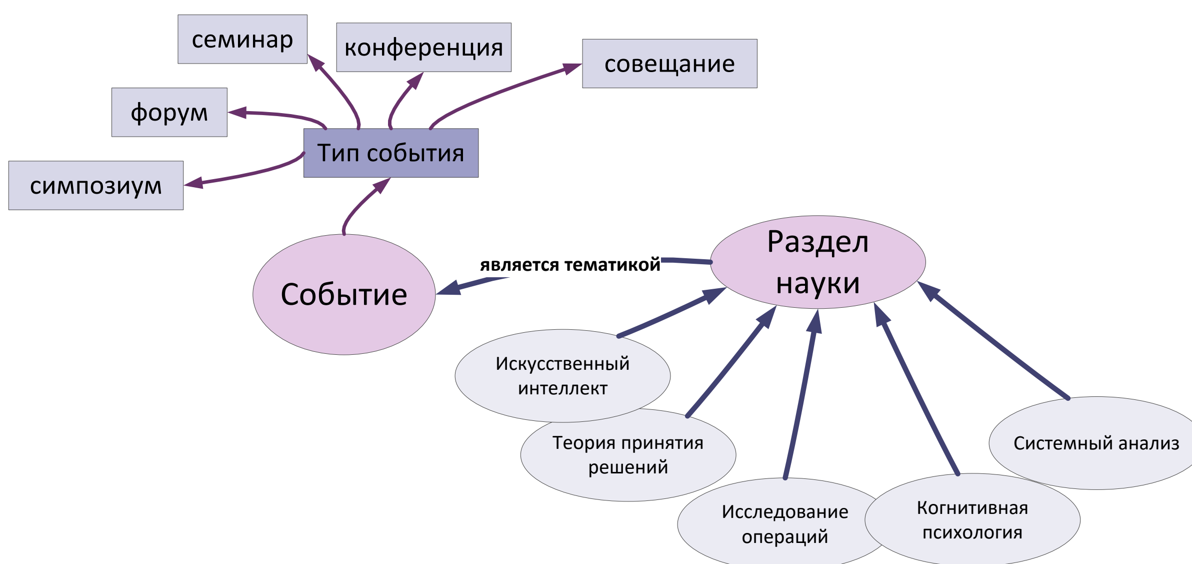

Рис. 2. Фрагмент онтологии, соответствующий шаблону на рисунке 1.

В примере таким атрибутом является *Тип события*, который может принимать одно из пяти возможных значений. Таким образом, $Класс_1$ в шаблоне соответствует классу *Событие* онтологии, а $Класс_2$ классу *Раздел науки*.

Согласно шаблону поискового запроса, приведенному на рисунке 1, в поисковый запрос попадает значение доменного атрибута первого класса и название экземпляра второго класса. Тогда для фрагмента онтологии, представленного на рисунке 2, можно построить поисковые запросы, представленные в таблице 1.

Таблица 1. Пример сгенерированных поисковых запросов

конференция Системный анализ
конференция Когнитивная психология
конференция Искусственный интеллект
конференция Теория принятия решений
конференция Исследование операций
семинар Системный анализ
семинар Когнитивная психология
…

Чтобы получить множество ссылок на веб-страницы, релевантные построенному поисковому запросу, используется свободная метапоисковая система с открытым исходным кодом Searx[1], которая соответствует спецификациям OpenSearch[2]. Система Searx позволяет выполнять поиск на различных языках и с помощью различных поисковых систем, сгруппированных по категориям.

Далее для каждого найденного документа (веб-страницы) оценивается его релевантность, на основании которой принимается решение о сохранении ее в базу и дальнейшем анализе. Релевантность оценивается в два этапа:

- В первую очередь нужно понять, посвящен ли найденный документ научной деятельности в определенной области знаний. Для этого сначала вычисляется релевантность данного документа онтологии, лежащей в основе ИНИР.
- Затем нужно определить конкретный класс (классы) онтологии, которому посвящен документ.

Для оценки релевантности документа классу онтологии используется векторная модель [8], в которой вектор документа включает абсолютные частоты встречаемости терминов

[1] http://asciimoo.github.io/searx/

[2] http://www.opensearch.org/Specifications/OpenSearch/1.1

(слов) за исключением стоп-слов, т.е. слов, не несущих смысловой нагрузки (предлогов, общеупотребимых слов и т.п.). Для учета встречаемости терминов в разных морфологических формах используются их основы (стемминг).

Векторное представление класса онтологии строится по формуле (1) на основе описания этого класса в онтологии: его атрибутов, связей с другими классами и соответствующих терминов тезауруса. Вспомогательный вектор $\vec{c}_{attr}$ включает :

- абсолютные частоты встречаемости терминов, входящих в название этого класса, и его атрибутов;
- абсолютные частоты терминов, входящих в допустимые значения доменных атрибутов.
- абсолютные частоты терминов тезауруса, связанных с этим классом.

$$\vec{c} = \vec{c}_{attr} + \gamma \sum \vec{c}'_{attr} \quad (1)$$

Где $\vec{c}$ – вектор класса онтологии, $\vec{c}_{attr}$ – вспомогательный вектор, учитывающий описание класса без его связей, γ – коэффициент, показывающий вклад связанных (отношением в онтологии) классов в векторное представление данного класса, $\vec{c}'_{attr}$ – вспомогательные вектора классов онтологии, связанных с данным классом.

Значение релевантности вычисляется с помощью косинусной меры между векторами класса онтологии и документа по формуле (2).

$$similarity_c = \frac{\vec{c} \cdot \vec{d}}{\|\vec{c}\| \cdot \|\vec{d}\|} = \frac{\sum_{i=1}^{n} c_i \times d_i}{\sqrt{\sum_{i=1}^{n} c_i^2} \times \sqrt{\sum_{i=1}^{n} d_i^2}} \quad (2)$$

Где $\vec{c}$ – вектор класса онтологии, $\vec{d}$ – вектор документа, n – длина векторов (число учитываемых терминов), а i – позиция термина в векторе.

Релевантность документа онтологии вычисляется аналогичным образом: вектор документа строится также, как и в предыдущем случае, а вот вектор онтологии строится аналогично вектору класса, только теперь учитываются все сущности онтологии. Классом онтологии, которому посвящен документ, считается класс с максимальным значением $similarity_c$.

Для уточнения запроса предполагается выделять ключевые слова из найденных документов с помощью статистических методов и в зависимости от значения релевантности добавлять эти слова в поисковый запрос вместе с различными операторами на языке поисковых запросов, которые поддерживает метапоисковая система *Searx*. Так, если

документ оказался не релевантным онтологии, в поисковый запрос добавляются ключевые слова из этого документа с оператором «-», исключающим результаты, содержащие эти слова.

Общий алгоритм поиска научных ресурсов, релевантных определенной области знаний, представлен на рисунке 3. Поиск запускается в соответствии с установленным расписанием. В начале каждого сеанса на основе *шаблонов поисковых запросов* генерируются запросы и добавляются в *очередь поисковых запросов*. Каждый поисковый запрос из очереди отправляется в метапоисковую систему *Searx*, после чего найденные с ее помощью ссылки на страницы добавляются в *очередь страниц на обработку*.

Затем для каждой страницы вычисляется релевантность и если она выше определенного порога, то ссылка на страницу сохраняется для дальнейшей обработки. Далее анализируются ссылки на этой странице и добавляются в *очередь страниц на обработку*. Кроме этого для каждой страницы уточняется поисковый запрос (в зависимости от значения релевантности) и добавляется в *очередь поисковых запросов*.

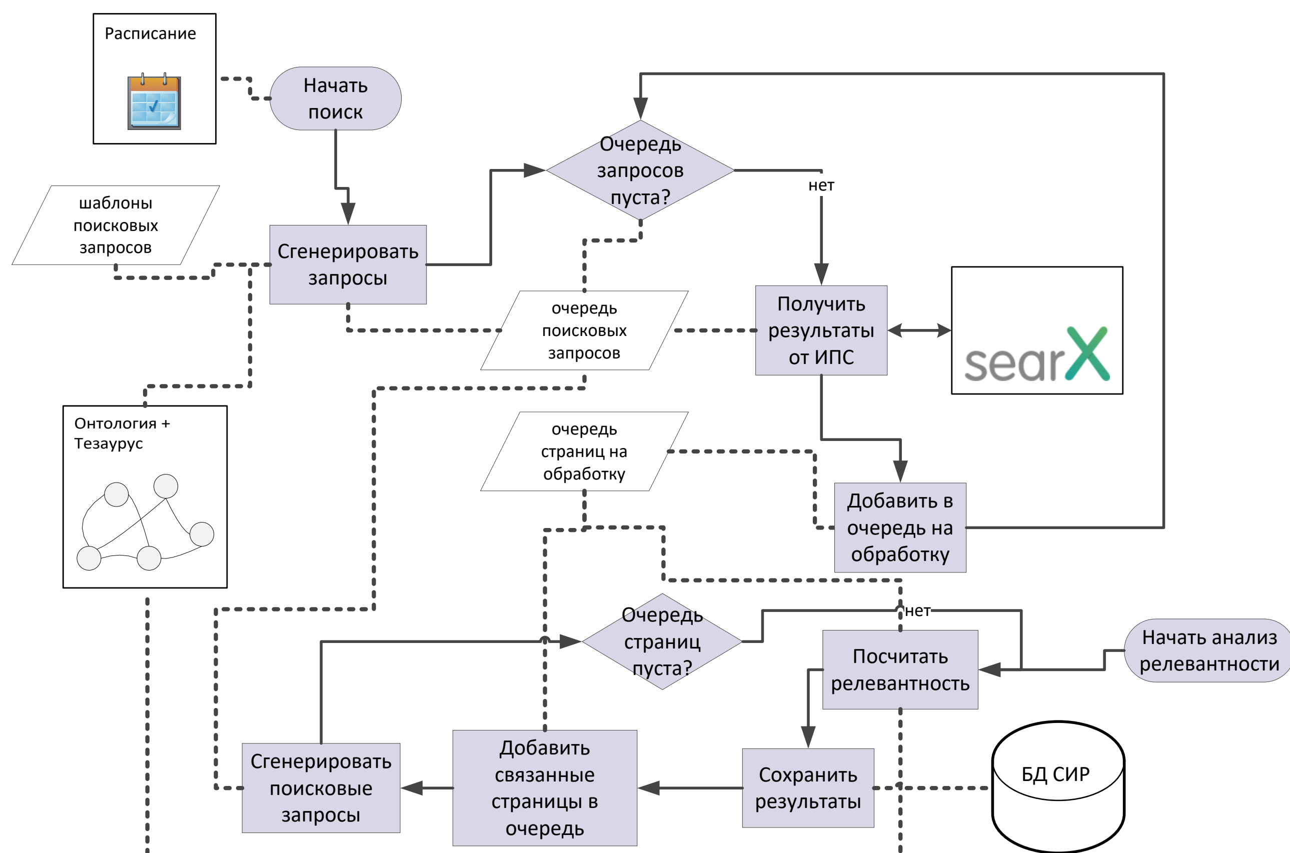

Рис. 3. Алгоритма поиска релевантных ресурсов.

4. Заключение

В статье предлагается метод поиска научных интернет-ресурсов в определенной области знаний, использующий онтологию для построения поисковых запросов и оценки релевантности найденных ресурсов. Для поиска используются глобальные поисковые системы, которым отправляются сгенерированные поисковые запросы, включающие названия сущностей онтологии и термины тезауруса.

В ходе дальнейшей работы предполагается доработать метод уточнения поисковых запросов и применить предложенный подход для пополнения онтологии ресурса по поддержке принятия решений в слабоформализованных областях.

Список литературы

1. Загорулько Ю. А., Загорулько Г. Б., Боровикова О. И. Технология создания тематических интеллектуальных научных Интернет-ресурсов, базирующаяся на онтологии // Программная инженерия. 2016. Т. 7, № 2. С. 51-60.
2. Aula A., Khan R. M., Guan, Z. How does search behavior change as search becomes more difficult? // Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. New York: ACM, 2010. P. 35-44
3. Bates M. J. The design of browsing and berrypicking techniques for the online search interface // Online review. 1989. Vo.13(5). P. 407-424.
4. Fields B., Keith S., Blandford A. Designing for expert information finding strategies //People and computers XVIII—Design for life. London: Springer, 2005. P. 89-102.
5. Geleijnse G., Korst J. H. M. Automatic Ontology Population by Googling // Proceedings of the Seventeenth Belgium-Netherlands Conference on Artificial Intelligence. Koninklijke Vlaamse Academie van Belie voor Wetenschappen en Kunsten, 2005. P. 120-126.
6. Guarino N. Formal Ontology in Information Systems // Formal Ontology in Information Systems. Proceedings of FOIS'98, Trento, Italy, June 6–8, 1998 / Ed. N. Guarino. Amsterdam: IOS Press, 1998. P. 3-15.
7. Hölscher C., Strube G. Web search behavior of Internet experts and newbies //Computer networks. 2000. Vol. 33. P. 337-346.
8. Manning C. D., Raghavan P., Schütze H. An Introduction to Information Retrieval. Online edition. Cambridge University Press. 2009. 544 pp.
9. Marchionini G. Exploratory search: from finding to understanding //Communications of the ACM. 2006. Vol. 49(4). P. 41-46.
10. Meng W., Yu C., Liu K. L. Building Efficient and Effective Metasearch Engines // ACM Computing Surveys (CSUR). 2002. Vol. 34. No. 1. P. 48–89.

11. Saito H., Miwa K. A cognitive study of information seeking processes in the WWW: the effects of searcher's knowledge and experience // Proceedings of the Second International Conference on Web Information Systems Engineering. IEEE, 2001. Vol. 1. P. 321-327.
12. Vallet D., Fernández M., Castells P. An ontology-based information retrieval model //European Semantic Web Conference. – Springer, Berlin, Heidelberg, 2005. – C. 455-470.
13. Waitelonis J., Exeler C., Sack H. Linked data enabled generalized vector space model to improve document retrieval //Proceedings of NLP & DBpedia 2015 workshop in conjunction with 14th International Semantic Web Conference (ISWC). CEUR-WS. – 2015. – T. 1486.
14. White R. W. et al. Supporting exploratory search, introduction, special issue, communications of the ACM //Communications of the ACM. 2006. Vol. 49(4). P. 36-39.
15. White R. W., Drucker S. M. Investigating behavioral variability in web search //Proceedings of the 16th international conference on World Wide Web. ACM, 2007. P. 21-30.
16. Xiong C., Callan J. Query expansion with Freebase //Proceedings of the 2015 International Conference on The Theory of Information Retrieval. – ACM, 2015. – C. 111-120.

УДК 004.4

Разработка систем автоматизированной оценки заданий по программированию

Иртегов Д.В. (Новосибирский государственный университет),

Нестеренко Т.В. (Институт систем информатики СО РАН, Новосибирский государственный университет),

Чурина Т.Г. (Институт систем информатики СО РАН, Новосибирский государственный университет)

В статье рассмотрена двадцатилетняя история и опыт создания автоматизированных систем проверки знаний и навыков по программированию в Новосибирском государственном университете. Приведен анализ эффективности или неэффективности методов и средств в использующихся системах проверки знаний. Описано современное состояние архитектуры системы NSUts.

Ключевые слова: *тестирование знаний, обучение программированию, олимпиады по программированию, система тестирования, NSUts, защита от мошенничества.*

1. Введение

Для оценки качества образования активно используется метод автоматизированного тестирования знаний и навыков. В различных формах этот метод применяется в школьном, высшем и послевузовском обучении. Сертификаты многих компаний, например, Microsoft Certified System Engineer, Cisco Engineer, Certified Lotus Professional выдаются на основе автоматизированного тестирования.

Эффективным комплексным средством проверки знаний и навыков программиста является написание программы, соответствующей заданным требованиям, с последующим её тестированием. Такие системы отличаются от традиционных автоматизированных систем тестирования тем, что они предполагают написание испытуемыми программ, последующий запуск которых осуществляется на заранее подготовленном наборе тестов.

В настоящее время опубликовано немного статей по автоматизированной проверке знаний, тем не менее, в разных странах такие системы существуют, и они достаточно популярны и востребованы, особенно при проведении различных соревнований по

программированию, так как проверять масштабные соревнования методом ручного тестирования затратно. Но, как правило, при работе таких систем необходимо присутствие автора.

Одной из первых, созданных в России, систем автоматизированной проверки была система ***T/Run*** для MS-DOS. Программа ***run*** обеспечивала запуск решений с ограничением по времени работы, а программа ***t*** использовала ***run*** для проверки решения на заданных тестах.

Система ***olympiads.ru*** [10] в своей основе содержит программу запуска решений с ограничением времени работы и использования памяти. Она дополнена базой данных задач и простым графическим интерфейсом пользователя, обеспечивающим редактирование базы данных и запуск решений на проверку.

На Всероссийских олимпиадах использовалась система ***Cyber Judge,*** созданная М.А. Бабенко. Но использование этой системы без автора затруднительно, так как настройка предполагает редактирование файлов конфигурации, кроме того, часто настройка системы предполагает дописывание к ней небольших модулей для конкретной олимпиады.

В качестве своеобразной системы самотестирования можно отметить систему ***tchoose***, используемую на Всероссийской олимпиаде школьников по информатике во время составления тестов для визуальной оценки их прохождения заранее заготовленными правильными и неправильными решениями.

Первой попыткой создать интегрированную систему стала связка Automated Programming Problem Evaluation System (APPES) на Java и Programming Contest Management System (PCMS) на Delphi, созданные в Санкт-Петербургском государственном институте точной механики и оптики. Их развитие превратилось в проект ***PCMS2*** [11]. Благодаря низкой связности модулей и качественному выделению высокоуровневых концепций достигается огромная гибкость. Однако при настраивании системы ***PCMS2*** большинство действий в нем осуществляется написанием файлов конфигурации, которые составляют значительную часть системы, как по важности, так и по объему, связывая ее модули вместе. В настоящий момент сервер ***PCMS2*** поддерживает только ОС Windows.

Олимпиады московского и некоторых других регионов проходили под управлением системы ***ejudge***, созданной в Московском государственном университете. Она является в некотором смысле противоположностью ***PCMS2***. ***Ejudge*** поддерживает только операционную систему Linux, написана на языке Си и поддерживает заранее определенный набор функций. Недостатком системы является возможная конкуренция за ресурсы между

тестируемым решением и самой системой. Тестирование проходит на одной машине, и подключить дополнительные физически невозможно.

Система автоматической проверки с архивом задач в испанском городе Вальядолиде [2] достаточна популярна среди студенческого сообщества. При работе используется операционная система Linux и языки программирования: GNU C, GNU C++, Free Pascal. Архив содержит сотни задач с тестами и постоянно пополняется. Для многих неудобным является использование системы Linux. Главным же недостатком данной системы и систем ***timus*** [1] Уральского государственного университета и ***Yandex Contest*** компании Яндекс [12] является то, что они используют модель Software as Service (SaS).

Для понимания функциональности и архитектуры системы NSUts, созданной в Новосибирском государственном университете (НГУ), в данной статье рассмотрена история разработки этой системы, описаны ее предшественники: системы на языке REXX, Chjudge, Uhjudge, аргументированы выбранные решения и проведен анализ их эффективности.

Опыт разработки и эксплуатации ряда разных программных решений сыграл важную роль как при определении требований к системе NSUts, так и при формировании архитектуры.

2. «Система» на REXX

Работы по автоматизации соревнований по программированию велись в Новосибирском государственном университете с 1998 года. Первое программное решение представляло собой скрипт на языке REXX длиной около тысячи строк. Оно имело многочисленные неустранимые недостатки и никогда не рассматривалось, как что-то пригодное для постоянной эксплуатации.

Однако, этот скрипт использовался более трех лет для проведения тренировок и соревнований, в том числе, для четвертьфинальных соревнований ACM-ICPC (Association for Computing Machinery – International Collegiate Programming Contest [7]), проводившихся в Новосибирском государственном техническом университете в 1999 году, а также Открытой Всесибирской Олимпиады по программированию в 2000 и 2001 годах [15, 16].

Историю этого решения следует начать с ноября 1997 года, когда был проведен первый региональный полуфинал чемпионата ACM-ICPC по Северо-Восточному Европейскому региону. НГУ получил официальное приглашение участвовать в этом мероприятии. Результаты команд НГУ трудно было назвать блестящими [13].

Руководством НГУ и заинтересованными преподавателями было осознано, что без систематической работы достичь приличных результатов в соревнованиях такого рода

невозможно. Первоначально обсуждалась идея наладить автоматическое тестирование при помощи скриптов командной оболочки MS/DR DOS *command.com* (*.bat*-файлов), но быстро было осознано, что возможностей языка *command.com* для решения этой задачи недостаточно.

Осенью 1998 года от организаторов Всероссийской олимпиады школьников была получена система, которая называлась PCMS, и была непосредственным предшественником ***PCMS2***. Эту систему планировалось использовать для проведения университетской олимпиады, по результатам которой должны были быть отобраны команды для участия в полуфинале ACM-ICPC.

В архиве файлов этой системы было обнаружено несколько бинарных исполняемых файлов для MS DOS, несколько конфигурационных файлов и никаких сопроводительных текстов. Было ясно, что формат конфигурационных файлов очень сложен, и по нему невозможно понять, в какой среде должна была исполняться система. В архиве был найден исполняемый модуль, функции которого удалось понять и которым можно было управлять при помощи параметров командной строки. Это была утилита ***run.exe***, которая позволяла запускать программы для DOS с контролем времени исполнения. Максимальное время задавалось параметром командной строки, при истечении этого времени запущенная программа принудительно завершалась. Также, утилита ***run*** отлавливала некоторые варианты нештатного завершения запущенной программы, например, ошибку среды исполнения Turbo Pascal. По результатам исполнения, утилита выдавала один из трех возможных вердиктов:

- «Успешное исполнение» (программа штатно завершилась с кодом возврата 0),
- «Ошибка времени исполнения» (программа штатно завершилась с кодом возврата, отличным от 0, или завершилась из-за ошибки среды исполнения)
- «Превышен лимит времени» (программа была принудительно завершена утилитой).

Д.В. Иртегов написал на языке REXX скрипт, использующий эту утилиту, который совмещал в себе реализацию очереди решений, тестовой обвязки и подсчета рейтинга.

Архитектура «системы» в целом выглядела следующим образом. Скрипт запускался на выделенном компьютере под Windows 95. Выбор ОС был продиктован следующими причинами:

1. Тестировать решения планировалось под DOS на языках программирования Borland C и Turbo Pascal. DOS-эмуляторы OS/2 и Windows NT имели многочисленные проблемы и ограничения, так что в них сложно было добиться корректной работы компиляторов этих языков. Возможно также, утилита ***run.exe*** не смогла бы корректно работать в этих эмуляторах. Это оставляло только две возможности: использовать настоящую DOS или DOS-окно Windows 95.

2. Интерпретатора REXX для DOS не существовало, был доступен только REXX/Win32, работающий под Windows NT и Windows 95. Выбор языка сложно было назвать осознанным решением. Фактически, использовался такой продукт, для которого имелся интерпретатор, и который был пригоден для решения задачи.

3. Операционная система Windows 95 обеспечивала несколько более качественную изоляцию программ для DOS, чем чистая DOS. Некоторые ошибки времени исполнения, которые под чистой DOS приводили к зависанию компьютера и *холодной* перезагрузке, под Windows 95 приводили к снятию задачи без перезагрузки всей «системы». Впрочем, полноценной изоляции Windows 95 не предоставляла, так что нередко все-таки приходилось делать холодную перезагрузку.

В качестве очереди решений использовалось дерево каталогов файловой системы. Это дерево должно было размещаться на сетевом файловом сервере. В нашем случае использовался сервер Novell Netware. В принципе, можно было использовать любой файловый сервер, который обеспечивал доступ со стороны клиентов DOS/Windows 95 и управление доступом на уровне пользователей. Последнему требованию на 1998 год удовлетворяли системы OS/2 LAN Manager и Windows NT, а позднее появился сервер Samba для ОС семейства Unix. Для проведения интернет-тура Открытой Всесибирской Олимпиады участники получали доступ к своим каталогам по протоколу FTP.

Компьютер с «системой» должен быть подключен к серверу от имени пользователя, имеющего доступ ко всему дереву каталогов. Для каждого участника соревнований создавалась отдельная учетная запись, которая имела доступ только к одному каталогу в дереве. Таким образом, участники не могли видеть решения друг друга.

Чтобы отправить решение на тестирование, участник должен был положить файл с исходным текстом решения в свой каталог дерева. Имя файла должно было соответствовать номеру задачи, а расширение — языку, на котором написана программа.

«Система» сканировала все каталоги дерева по очереди. Обнаружив файл с решением, она выполняла следующие действия:

1. Перемещала его в рабочий каталог и копировала в архивный каталог, добавляя к имени файла префикс, соответствующий имени каталога участника.

2. Определяла язык программирования и номер задачи.

3. Запускала соответствующий компилятор.

4. Проверяла результат компиляции и, в случае неудачи, выдавала вердикт «ошибка компиляции».

5. При успешной компиляции, последовательно запускала полученную программу под контролем утилиты **`run.exe`** на наборах заранее подготовленных тестов. Наборы тестов хранились в отдельных каталогах с именами, соответствующими номерам задач. Каждый набор представлял собой наборы файлов с именами 1.in, 1.out, 2.in, 2.out и т. д. Легко понять, что имя файла соответствовало номеру теста, а расширение — типу файла: входные данные или эталонные (правильные) выходные данные.

6. Если ***`run.exe`*** выдавала вердикт «Успешное исполнение», «система» сравнивала выходной файл программы с эталонным файлом правильного ответа при помощи утилиты ***`diff`***. Использовалась версия ***`diff`*** из поставки Borland C. Позднее была реализована также более сложная схема проверки при помощи программы, предоставленной жюри.

7. Если программа в процессе работы «подвешивала» компьютер и приходилось его перезагружать, то это обнаруживалось следующим образом: при нормальном запуске «системы» рабочий каталог должен был быть пуст. Если там лежали какие-то файлы, предполагалось, что они остались от неудачной попытки запуска. Это интерпретировалось как ошибка времени исполнения.

По результатам выполнения всех предыдущих шагов, «системы» могла получить один из следующих вердиктов:

- «Ошибка компиляции» (программа не скомпилировалась).
- «Ошибка исполнения на тесте N» или «Лимит времени на тесте N». Это определялось по соответствующим вердиктам утилиты ***`run`***, а также по наличию файлов в рабочем каталоге после перезагрузки.
- «Неверный ответ на тесте N». Этот ответ определялся по результатам сравнения выдачи программы с эталоном.

- «Принято». В этом случае программа участника выдала ответы, соответствующие эталонным, на всех тестах.

Определив вердикт, «система» выкладывала его в каталог участника в виде файла с именем, соответствующим номеру задачи, и с расширением .RES. Кроме того, «система» вносила вердикт в общий файл рейтинга.

Файл рейтинга представлял собой текстовый файл, в котором каждая строка соответствовала результатам одного участника. Строка была разбита на позиции при помощи символа `'|'`. В позициях, соответствующих номерам задач, стояло текущее состояние, закодированное следующим образом:

1. пробел соответствовал задаче, которую участник сдавать не пытался.
2. Знак '−' и десятичное число соответствовали одной или нескольким неудачным попыткам сдачи задачи.
3. Знак '+', возможно с последующим десятичным числом, соответствовал успешно сданной задаче. Число при этом соответствовало количеству неудачных попыток.
4. В последней позиции хранилось штрафное время, вычисляемое по правилам соревнований ACM. При этом, временем сдачи задачи считалось время модификации файла с исходным текстом.

Таким образом, файл был похож по формату на таблицу рейтинга соревнований ACM-ICPC. Для хранения рейтинга нужно было использовать какую-то базу данных, но в тот момент этого еще не было реализовано. Позднее, в 2001 году, Д.В. Иртегов этот недостаток устранил, переписав скрипт на Perl, вместо текстового файла была использована хранимая хэш-таблица Berkeley DB.

Среди наиболее важных недостатков и проблем этой «системы» следует упомянуть следующие.

1. Нетранзакционную процедуру отправки решения на проверку. Участник обнаруживает, что задача передана на проверку, по исчезновению файла из каталога. Это неудобно для пользователя и порождает целый класс возможных ошибок соревнования (*race condition*). Особенно чувствительным это неудобство было при работе через протокол FTP.

2. Невозможность отслеживать очередь тестирования для участников.

3. Нетранзакционную работу с базой данных рейтинга. Файл с рейтингом считывался и разбирался заново после каждого тестирования каждого решения, он перезаписывался полностью после каждого изменения рейтинга. Сбой сети, сервера или рабочей станции в

этот момент мог привести к потере данных. Кроме того, при пересчете рейтинга «система» делала много лишней работы. С переходом на Berkeley DB этот недостаток был, в основном, устранен.

4. Хранение только последнего отправленного участником решения. Это делало невозможной корректную обработку некоторых апелляций. Так, если в предоставленных жюри тестах будет обнаружена ошибка, то, следует проверить все решения всех участников на откорректированном наборе тестов, и зачесть как успешную ту попытку сдачи, которая первая пройдет все тесты. Но, поскольку «система» хранила только последнее решение, это оказывалось невозможным. Значимость этого недостатка была осознана после того, как такая апелляция была принята и корректно обработана системой на финале ACM-ICPC 2000.

5. Невозможность горизонтального масштабирования. Поскольку выбор решений из каталогов нетранзакционен, то запустить несколько экземпляров «системы» на разных машинах для параллельного тестирования невозможно. Работу с файлом рейтинга можно было бы синхронизовать, используя механизм блокировок файлов, но из-за неразрешимой проблемы с выборкой решений не делалось даже попыток реализовать это.

6. «Система» выбирает решения не в соответствии с временем их отправки, а в соответствии с лексикографическим порядком имен каталогов, так что решения участников с лексикографически неудачными именами тестируются позднее.

7. «Система» не защищена от простого мошенничества, когда участник задает произвольное время модификации файла с решением. Технически это несложно, в DOS есть соответствующий системный вызов, и это приведет к существенным нарушениям в алгоритме подсчета рейтинга по правилам ACM. К счастью, за все время эксплуатации системы никому из участников это не пришло в голову.

8. «Система» не предоставляла удобных средств коммуникации между участниками и жюри. Правила олимпиад ACM, на которые мы ориентировались, предусматривали такую коммуникацию в форме вопросов участников и ответов жюри. В нашей «системе» такая функциональность была реализована следующим образом. Если участник загружал в свой каталог файл с расширением .QUE (Question), то «система» вместо его компиляции выдавала на консоль тестового компьютера сообщение «Вопрос от участника». Член жюри, сидящий за другим компьютером, мог получить доступ к файлу с вопросом и положить в каталог участника другой текстовый файл с расширением .ANS (Answer). Это работало, но было очень неудобно и для участников и для жюри.

Как уже говорилось, все эти недостатки были ясны уже при разработке решения. Одной из главных причин, по которой решение со всеми этими проблемами все-таки эксплуатировалось, было мнение, что разрабатывать «нормальную» систему для тестирования задач под DOS не имеет смысла. Штатная среда исполнения — чистый DOS или Windows 95 — не обеспечивает полноценной изоляции, поэтому плохо себя ведущая программа может «подвешивать» ОС. Следовательно, компьютер, на котором проводится тестирование, должен находиться под постоянным присмотром. Таким образом, следует говорить не об автоматизированном, а о механизированном тестировании, и неполная адекватность средств такой механизации не является главной проблемой.

3. Система CHjudge

В 2000 году произошло несколько событий, которые заставили отказаться от ранее описанной «системы» и заняться разработкой полноценного решения.

Во-первых, команда НГУ вышла в финал ACM-ICPC, и на финале произошел эпизод с перетестированием, который произвел на нас большое впечатление. Стало ясно, что если даже на таких высокоранговых соревнованиях случаются ошибки в тестах, то и наши университетские мероприятия от этого не застрахованы. Следовательно, процедура тестирования должна допускать справедливую по отношению к участникам обработку таких ситуаций.

Во-вторых, соревнования Северо-Восточного Европейского полуфинала ACM-ICPC, наконец-то, отказались от DOS и перешли на тестирование заданий под Windows. NT обеспечивает гораздо более качественную изоляцию пользовательских процессов, так что «плохо себя ведущая» программа практически не может нарушить функционирование системы в целом. Таким образом, главный аргумент, который удерживал нас от разработки полноценной системы, потерял силу.

В-третьих, организация интернет-тура Открытой Всесибирской Олимпиады по программированию вызвала нарекания у многих участников, и значительная часть этих нареканий была связана с системой тестирования.

В обсуждении архитектуры новой системы принимали участие наши финалисты ACM-ICPC: Евгений Четвертаков, Александр Шапеев и Алексей Бабурин, которые на тот момент были студентами НГУ, Д.В. Иртегов, Т.Г. Чурина, сотрудник ИСИ СО РАН С.К. Черноножкин. Довольно быстро был достигнут консенсус по следующим требованиям к системе:

1. Должно поддерживаться проведение соревнований по правилам ACM-ICPC с тестированием программ на языках C, C++, Pascal для Win32.

2. Язык программирования должен указываться участником явно при отправке задания, а не определяться по косвенным признакам, таким, как расширение файла. Это позволило бы проводить в рамках одного тура тестирование программ на нескольких диалектах одного языка, например, Visual C и Borland C.

3. Система должна предоставлять участникам веб-интерфейс для отправки задач, просмотра состояния их тестирования и рейтинга, а также для коммуникации с жюри.

4. Сами задачи, очередь тестирования и рейтинг должны храниться в транзакционном хранилище, скорее всего в реляционной СУБД.

5. Должна сохраняться полная история всех попыток отправки решения, все исходные тексты и точные времена их отправки. Это необходимо как для обработки апелляций, требующих перетестирование, так и для расследования попыток мошенничества.

6. Тестирование решений должно производиться на выделенных компьютерах. Главной мотивацией этого требования было то, что время исполнения задачи в многозадачной среде определяется сложными сценариями конкуренции за разные ресурсы: оперативную память, кэш процессора, время переключения контекста процессора и т. д. Исполнение решения на выделенном компьютере позволило бы свести влияние такой конкуренции до минимума.

Кроме того, исполнение задач на выделенных компьютерах позволило использовать разные операционные системы. На тестирующих компьютерах («тестовых клиентах») могла использоваться Windows NT, а для веб-сервера системы и сервера базы данных — Linux. Это оказалось особенно важно потому, что в 2000 году не было приемлемых по производительности и надежности веб-серверов для Windows NT.

Пункт 6 привел к разделению системы на два компонента:

- центральный сервер, предоставляющий веб-интерфейс, а также хранилища тестов, очереди решений, рейтинга и другой информации;
- тестирующие клиенты.

За разработку системы взялся Евгений Четвертаков, к тому моменту уже имевший некоторый опыт разработки веб-приложений. В качестве языка разработки был выбран язык Perl, в то время довольно популярный для реализации серверной части веб-приложений. В качестве сервера СУБД был использован MySQL.

Серверная часть системы представляла собой типичное веб-приложение, общающееся с пользователями – участниками и членами жюри, при помощи HTML-форм и хранящее персистентные данные в СУБД.

Логика работы тестирующего клиента была похожа на логику работы скрипта «системы», рассмотренной в предыдущем разделе. Разница состояла в том, что клиент содержал логику работы с очередью решений и тестовую обвязку, а логика вычисления рейтинга была перенесена на сервер. Несмотря на сходство логики, код скрипта «системы» в коде клиента не переиспользовался. Большая часть кода клиента была реализована на Perl, часть – в виде .BAT файлов для CMD.EXE (командного процессора Win32).

Отдельным вопросом при разработке архитектуры системы был механизм взаимодействия центрального сервера с тестирующими клиентами. Логически возможны два варианта:

1. Режим проталкивания (*push*), когда клиент ожидает запроса от сервера на тестирование задачи. При появлении задачи в очереди, сервер выбирает одного из клиентов и передает ему задачу.

2. Режим вытягивания (*pull*), когда клиенты в холостом цикле опрашивают сервер, не появилось ли у него новой задачи для тестирования. Защита от передачи задачи двум клиентам одновременно осуществляется за счет использования транзакционной очереди, реализованной на основе СУБД.

На первый взгляд кажется, что вариант проталкивания более привлекателен. Действительно, при вытягивании, клиенты должны постоянно опрашивать сервер, что создает паразитную нагрузку на него. Однако при реализации системы был принят режим вытягивания, не столько из-за простоты его реализации, сколько из-за соображения, которое можно кратко сформулировать следующим образом: при малых нагрузках режим проталкивания бесполезен, а при больших он не работает.

Дело в том, что тестирование каждого отдельного решения – это длительный процесс, состоящий из многих шагов, каждый из которых может завершиться неудачей: ошибка компиляции, ошибка на первом тесте, ошибка на втором тесте и т.д. Предсказать время тестирования каждой отдельной программы невозможно. В худшем случае, с точки зрения нагрузки, программа может пройти все тесты, затратив на каждый максимально допустимое время, а в лучшем случае она может быть снята по ошибке компиляции.

Поэтому, передав решение клиенту, сервер не может определить, когда этот клиент освободится, и даже не может оценить интервал, с которым этот клиент будет обновлять информацию о своем состоянии.

Большую нагрузку на систему можно определить, как состояние, когда значительную часть времени почти все клиенты заняты тестированием. В этих условиях, сервер не имеет свободных клиентов и не может определить, когда они освободятся. Поэтому режим проталкивания оказывается неработоспособен.

Паразитная нагрузка на сервер, создаваемая опросом со стороны вытягивающих клиентов, возникает только в условиях, когда клиенты ничем не заняты, то есть в условиях низкой нагрузки. В этой ситуации, паразитная нагрузка не представляет самостоятельной проблемы.

Разработанное Е.А. Четвертаковым программное решение оказалось весьма удачным. Оно удовлетворяло всем осознаваемым на тот момент требованиям, обеспечивало высокую надежность и приемлемую производительность даже под довольно высокими нагрузками.

В 2003 году система была использована для проведения интернет-тура Открытой Всесибирской Олимпиады. В интернет-туре олимпиады участвовало 102 команды.

До 2007-2008 года, система Четвертакова активно использовалась в НГУ для проведения Всесибирской олимпиады по программированию, тренировок команд НГУ и ряда других мероприятий. Были попытки ее использования в учебном процессе в курсе «программирование на языке высокого уровня» на ФИТ и ММФ НГУ.

Первоначально система поддерживала проведение соревнований только по правилам ACM-ICPC, и соответствующая бизнес-логика организации тестирования и правила подсчета рейтинга были жестко закодированы. Позднее был добавлен модуль, поддерживающий проведение соревнований по правилам российской школьной олимпиады по информатике. В этом модуле и правила, и бизнес-логика были также жестко закодированы.

Наиболее важным отличием с точки зрения бизнес-логики является следующее:

- При тестировании задачи по правилам ACM-ICPC, принятая задача обязана успешно пройти все тесты, т.е. получить вердикт «принято». Таким образом, если на каком-то тесте получается вердикт, отличный от «принято», последующие тесты можно не запускать.
- По правилам российской школьной олимпиады, задача получает определенное количество баллов за каждый принятый тест, и не обязана проходить все тесты, чтобы

эти баллы были включены в рейтинг. Поэтому, если программа выдала «неверный ответ» или «ошибку времени исполнения», тестирование необходимо продолжать.

Оказалось, что система вполне работоспособна без участия разработчика, благодаря достаточно хорошо написанным пошаговым инструкциям, как развернуть систему и подготовить ее к проведению мероприятия.

В ходе эксплуатации был выявлен ряд недостатков, как функциональных, так и нефункциональных.

Среди наиболее важных функциональных недостатков, требовавших пересмотра архитектуры системы, следует назвать следующие.

1. Система была рассчитана на однократное проведение одного мероприятия. Для проведения другого мероприятия с другими задачами и тестами необходимо было развернуть систему заново, создав новую базу данных. Конфигурация системы была рассчитана на сосуществование нескольких экземпляров системы на одном сервере, поэтому данная операция не требовала уничтожения данных предыдущих мероприятий. Для проведения соревнований этот недостаток не казался важным, но для тренировок и использования в учебном процессе это превратилось в существенную проблему.

2. Система обеспечивала веб-интерфейс для большинства сценариев штатного функционирования, но многие нештатные ситуации, например, дисквалификация участника соревнований или удаление ошибочно созданной учетной записи, требовали прямой модификации базы данных.

3. Система имела очень простое управление привилегиями с разделением всех учетных записей на две категории: участников и жюри/администраторов. При проведении крупных мероприятий с многочисленным жюри и техническим комитетом это превращалось в существенную проблему. Так, доброволец-студент, работа которого состояла в том, чтобы распечатывать исходные тексты программ по запросам участников и разносить их по терминальным классам, должен был иметь административный доступ к системе, что давало ему технические возможности останавливать и продолжать тур, отвечать на вопросы от имени жюри и т. д. Случаев злоупотребления этими правами зафиксировано не было, но само существование такой возможности вызывало у оргкомитета серьезные опасения.

Кроме названных функциональных недостатков, система имела и ряд существенных архитектурных и реализационных проблем, а именно:

1. Система была реализована с грубыми нарушениями принципа разделения содержания, представления и поведения. Каждая страница веб-интерфейса генерировалась отдельным скриптом, который выглядел, скорее, как HTML-код со вставками кусочков кода на Perl, чем как программа. В начале 2000-х годов такой стиль веб-программирования рекомендовался многими учебниками и размещенными в интернете руководствами, но уже к середине десятилетия он был признан плохой практикой. Это затрудняло как изменения дизайна веб-интерфейса, к 2007-2008 году интерфейс уже выглядел устаревшим, так и доработку функциональности и бизнес-логики системы.

2. Код системы не имел выделенного слоя модели данных. SQL-запросы к БД были включены непосредственно в код скриптов, генерирующих веб-страницы. Изменение модели данных могло потребовать внесения согласованных изменений во все файлы исходного кода приложения. Это также признано в отрасли плохой практикой и сильно затрудняло доработку и поддержку системы.

3. Тестирующие клиенты опрашивали очередь решений при помощи прямых запросов к СУБД. Это снижало безопасность, например, приходилось хранить в коде системы имя и пароль для доступа к СУБД, и затрудняло решение некоторых вспомогательных задач. Практически невозможно было отслеживать состояние клиента через веб-интерфейс системы. Если по какой-то причине клиент зависал или вовсе отключался, система не могла это обнаружить.

4. К 2007 году версия Debian Linux, под которую была разработана CHjudge, была официально снята с поддержки. В более новых версиях Debian, некоторые библиотеки, использовавшиеся при реализации системы, были объявлены устаревшими и исключены из репозиториев Debian, а некоторые из них — и из репозиториев CPAN (Compherensive Perl Archive Network).

Устаревшие архитектура и стиль кодирования системы затрудняли привлечение студентов к ее доработке, а потребности в доработке нарастали. Кроме перечисленных выше крупных функциональных недостатков, накапливалось большое количество мелких функциональных и эргономических замечаний. Поэтому в 2007 году был поставлен вопрос о разработке новой системы с нуля.

4. Ujudge

Относительно успешный опыт разработки и эксплуатации Chjudge породил у оргкомитета Всесибирской Олимпиады и лично у авторов иллюзию, что задача автоматизации

соревнований по программированию – это задача студенческого уровня. Это утверждение может быть более подробно изложено следующим образом.

1. Функциональность серверной части системы тестирования достаточно проста и почти не содержит вычислительно сложных алгоритмов, поэтому серверный компонент не может оказаться узким местом в производительности системы в целом.

2. Вследствие предыдущего пункта, не нужно уделять большого внимания производительности серверной части. Можно сосредоточиться на сборе и формулировке точных функциональных требований и пожеланий к системе, и на точном воплощении этих требований в программном коде.

3. Выбор языка и платформы для разработки не играет большой роли. Действительно, Perl, на котором была реализована система CHJudge, представляет собой интерпретируемый язык со слабой динамической типизацией и поздним связыванием. Производительность таких языков обычно довольно низка, и по большинству синтетических тестов Perl находился примерно на одном уровне с другими популярными языками этого типа, такими, как PHP, Python или популярный в 2007-2008 годах Ruby.

4. Поскольку на пути от функциональных требований к законченному продукту нет существенных подводных камней, с разработкой по заданным требованиям может справиться любой достаточно мотивированный успевающий студент, специализирующийся в области информационных технологий, или небольшая группа таких студентов. Эти студенты, как и при разработке CHjudge, могут работать без плотного контроля со стороны опытных наставников. Также, поскольку мотивация студентов-разработчиков является одним из приоритетных параметров, студентам можно предоставить выбор языка и платформы для разработки.

Практика показала ошибочность всех этих пунктов, но именно ими мы и руководствовались при организации работ над новой версией системы.

Руководство разработкой новой версии системы взяла на себя преподаватель ФИТ и ММФ НГУ, участник жюри Всесибирской Олимпиады Т.Г. Чурина. Она же координировала сбор и формализацию требований к системе. Д.В. Иртегов выступал в роли консультанта и внес достаточно большой вклад, чтобы нести ответственность за результаты.

Главным источником требований к новой системе были те функции, которые успешно исполняла система CHJudge, а также списки замечаний к этой системе. Замечания можно

рассматривать как функции, которые были нужны жюри или участникам, и которые система не исполняла. Эти требования впоследствии легли в основу требований к системе NSUts.

Непосредственно разработкой занялся студент бакалавриата ФИТ А.В. Таранцов. По его предложению, разработка велась на языке Ruby с использованием фреймворка Rails. Несколько позже к группе разработчиков присоединились студенты ФФ А. В. Киров и С.Б. Факторович.

Весной 2008 года разработка новой версии была сочтена завершенной. А.В. Таранцов успешно защитил квалификационную работу бакалавра.

Архитектура новой системы, получившей название Ujudge, сохранила определенную преемственность с CHJudge. Система состояла из сервера, реализующего веб-интерфейс для работы жюри и участников соревнований, и тестирующих клиентов, на которых выполнялось, собственно, тестирование. Код и логика работы тестирующего клиента не подверглись сколько-нибудь значительным изменениям.

Серверная часть была полностью переписана с учетом новых требований на новом языке и новых технологиях. Модель данных также была спроектирована с нуля. Единственное, что у серверной части Ujudge было общего с CHJudge — это использование MySQL в качестве СУБД.

Наиболее заметным для пользователей усовершенствованием был пользовательский интерфейс системы. Он был полностью перепроектирован, введены элементы технологии AJAX. В частности, для получения оповещений об ответах на вопросы и новостей не надо было перезагружать веб-страницу.

Еще одной новой особенностью Ujudge был механизм подсчета рейтинга олимпиады. Заложенную в этом механизме идею можно описать как компромисс между

- «жестким кодированием», когда весь алгоритм подсчета рейтинга реализован как модуль на том же языке программирования, что и сама система. Этот модуль может без ограничений использовать все внутренние интерфейсы системы, поэтому для разработки новых модулей такого же типа необходимо глубокое знание внутренней организации системы, а реорганизация системы, в свою очередь, может потребовать переделки модуля.
- «мягким кодированием», когда алгоритм подсчета рейтинга описан в конфигурационном файле. Это требует введения в язык конфигурации конструкций, аналогичных языкам программирования (переменных, присваиваний,

условных операторов, циклов), а в пределе и вовсе разработки нестандартного полного по Тьюрингу императивного или декларативного языка. Это приводит к тому, что конфигурация системы становится очень сложной. Действительно, для стандартных языков программирования существует хорошая документация и разного рода инструментальные средства, облегчающие разработку: редакторы с синтаксическим подчеркиванием, отладчики, интегрированные среды и т.д. Нестандартный язык программирования ничего этого не имеет. Поэтому «мягкое кодирование» может сделать процесс настройки системы существенно более сложным, чем модификация исходного кода.

Идея компромисса состоит в том, чтобы предоставить пользователям системы API для разработки плагинов – модулей, взаимодействующих с остальной системой через выделенные хорошо документированные интерфейсы. Таким образом, пользователь может разработать свой модуль подсчета рейтинга по своим правилам, используя стандартный язык программирования и стабильный API.

А.С. Таранцов разработал сложную компонентную архитектуру, фактически целый фреймворк, для подсчета рейтингов по разным алгоритмам. Этот фреймворк включал в себя алгоритм построения таблицы путем вызова предоставленных пользователем, в данном случае – разработчиком рейтинга, функций. Для доступа к результатам проверки решений предоставлялись специальные классы языка Ruby, которые следовало использовать вместо прямых обращений к СУБД.

Система была опробована на тренировках сборной НГУ и получила положительные отзывы, хотя некоторые из пользователей жаловались на низкую наблюдаемую производительность. Но попытка ее использования для проведения очного-тура Всесибирской Олимпиады по программированию в сентябре 2008 года закончилась неудачей.

В очном туре 2008 года участвовало 46 команд. Во время пробного тура участники ознакомились с новым интерфейсом системы, и многие высказали положительные отзывы. В первые полчаса тура система вела себя вполне удовлетворительно, но потом стало заметно, что загрузка процессоров сервера растет. Приблизительно ко второму часу соревнований, загрузка всех процессорных ядер дошла до 100%, а через некоторое время сервер перестал реагировать на HTTP запросы. Сложилась сложная ситуация, но удалось быстро написать скрипт миграции данных и учетных записей из Ujudge в CHJudge и продолжить соревнования на «четвертаковской» системе.

Очный тур 2008 года проводился на CHjudge, но сам эпизод надолго запомнился всем участникам событий.

Анализ причин случившегося показал, что фреймворк подсчета рейтинга использовал алгоритм, вычислительная сложность которого и количество обращений к базе данных квадратично зависели от количества сданных задач. Пока сданных задач было мало, рейтинг вычислялся быстро. Когда количество сданных задач увеличилось, подсчет рейтинга не только занял все процессоры, но и заблокировал все остальные процессы через блокировки СУБД. Поскольку на тренировках количество сданных задач не достигало порога, при котором это происходит, тестирование на тренировках эту проблему не выявило.

В ходе последующего «разбора полетов» был достигнуто соглашение по следующим позициям:

1. Задача разработки полноценной системы автоматизации проведения олимпиад – это задача не студенческого уровня. Хотя привлекать студентов к разработке системы можно и, с педагогической точки зрения, даже нужно, разработка должна вестись под управлением и контролем со стороны взрослых.

2. В процесс разработки системы следует ввести полноценный, хотя, может быть, и упрощенный по сравнению с коммерческой разработкой, контроль качества, включающий отслеживание не только функциональных, но и нефункциональных требований, в первую очередь – производительности и отчуждаемости кода.

3. Разрабатывать новую версию системы с нуля не следует. Необходимо сосредоточиться на доработке существующего работоспособного кода. Нефункциональные требования по модифицируемости и отчуждаемости кода следует обеспечивать за счет поэтапного рефакторинга.

После принятия этих решений, встал вопрос, какую из имеющихся систем, Ujudge или CHJudge, следует принять за «существующий работоспособный код».

Анализ ситуации показал, что платформа Ruby on Rails, строго говоря, не является платформой. Разработчики Ruby и Rails используют политику разработки, известную как «катящееся обновление» (rolling update). Попросту говоря, все изменения в систему вносятся только в последнюю, «текущую» версию. Так, если в коде интерпретатора или библиотек обнаруживается ошибка, то она исправляется только в последней версии. Поэтому все разработчики, использующие Ruby On Rails, вынуждены постоянно синхронизовать среду разработки с основным репозиторием. Устаревших, но поддерживаемых версий,

аналогичных «стабильным» версиям продуктов с открытыми исходными текстами или релизам коммерческих программных продуктов, в Ruby не существует. Фактически, вся платформа представляет собой непрерывную бета-версию.

При этом, разработчики Ruby не очень заботятся о совместимости своей платформы с самой собой. Обновление на новую версию среды исполнения часто ломает существующие программы и требует внесения изменений в их исходный код.

Зафиксировать версию платформы невозможно, так как в ней могут быть обнаружены серьезные ошибки, например, уязвимости с точки зрения безопасности. Эти ошибки будут исправлены только в текущей версии системы, то есть придется обновлять всю среду, а это, в свою очередь, потребует внесения изменения в ваше приложение для устранения интерфейсов, которые разработчики Ruby сочли «устаревшими».

Таким образом, разработка и поддержка приложения на платформе Ruby on Rails также превращается в «катящееся обновление», темп которого задается разработчиками платформы. Для короткоживущих программных проектов, например, для учебных проектов или прототипов это может быть терпимо, но для продуктов с многолетним жизненным циклом абсолютно неприемлемо.

В современном Ruby политика разработки более адекватна требованиям продуктов с длительными жизненными циклами, но здесь описано положение дел, имевшееся в 2008-2009 годах.

Поэтому за базу для дальнейшей разработки была принята система CHjudge. Дальнейшая история этой системы – это уже история системы NSUts.

5. Система NSUts

Одним из основных требований, предъявляемых работодателями к выпускнику ВУЗа, является умение работать в команде. Наилучшим способом обучения навыкам командной работы является участие в реальном программном проекте, особенно если этот проект завершается успехом. Реальный проект, по определению, должен быть нацелен на достижение определенного, нужного заказчику, результата в определенные сроки. Однако подключение студентов к таким проектам сталкивается с рядом проблем, главной из которых является то, что студенты не имеют опыта работы, и, следовательно, проект с их участием с высокой вероятностью может завершиться срывом сроков, превышением бюджета или полной неудачей. Теоретически, эти недостатки можно было бы компенсировать созданием смешанных команд из «взрослых» разработчиков и студентов. Но найти достаточное

количество опытных разработчиков, согласных работать в высоко-рисковом низкобюджетном проекте, практически невозможно. Вопрос о методологиях командной разработки, которые могли бы снизить риски, создаваемые неопытными разработчиками, и, таким образом, повысить вероятность успеха проекта, не является целью этой статьи.

Требования, предъявляемые к автоматизированным системам тестирования знаний, а именно: обеспечение глубокого и адекватного тестирования знаний и навыков, эффективной защиты от мошенничества и простота в эксплуатации для специалистов средней квалификации раскрыты в статье [14]. Безопасность в системе NSUts изложена в статье [3], описание прототипа системы NSUts – в статье [4].

Автоматизированная система NSUts [9] состоит из трех основных подсистем: клиентского ПО, сервера олимпиад (учебных туров) и тестирующего клиента. Клиентское ПО (браузер и среда разработки, в которой пользователь разрабатывает и отлаживает свое решение) установлено на компьютере пользователя и не предоставляется системой, но, как легко понять, принимает активное участие в работе. Тестирующих клиентов в системе может быть несколько. Единственный выделенный узел системы – это сервер.

Рис. 1. Архитектура системы NSUts

Сервер тестирования реализует логику проведения олимпиады (тура) и выполняет следующие функции:

1. Предоставляет веб-интерфейс для взаимодействия участников и членов жюри (преподавателя) с системой тестирования.
2. Автоматизирует управление олимпиадой (туром), а именно, осуществляет:

– сбор и хранение условий задач, тестов и решений;
– обработку и отображение результатов тестирования;
– составление рейтинга мест, которые заняли участники соревнования;
– получение отчётов о проведении олимпиады (тура);
– перетестирование решений участников;
– решение задач организационного плана, таких как регистрация участников, обеспечение обратной связи с жюри и других;
– администрирование олимпиад и туров.

Тестирующий клиент непосредственно осуществляет прогон решения на тестовых данных. Решения участников, полученные сервером, помещаются в очередь решений, откуда они забираются тестирующим клиентом. Тестирующий клиент в процессе обработки решения получает исходный код решения, информацию о необходимом для компиляции решения компиляторе, набор тестов. Исходный код компилируется и запускается в изолирующей среде на наборе тестов, результат работы программы сравнивается с эталонными результатами. Результат тестирования отправляется обратно на сервер, где происходит его обработка, составление рейтингов и так далее. Участники олимпиад и члены жюри взаимодействуют с системой исключительно через веб-интерфейс и не взаимодействуют с тестирующим клиентом напрямую.

Сервер написан на языке Perl с использованием модулей архива CPAN и некоторых утилит командной оболочки (shell для Linux или cmd.exe для Windows). Работа сервера олимпиад осуществляется под управлением веб-сервера apache2 [18]. Хранение данных приложения осуществляется в базе данных MySQL. Сервер тестирования может быть запущен как ОС Linux, так и в ОС Windows.

В настоящее время ведется покомпонентная переработка сервера на архитектуру AJAX, когда сервер отдает пользовательским машинам только данные в формате JSON, а их преобразование в красивый и удобный пользовательский интерфейс производится уже на клиенте, с использованием программ на JavaScript, исполняющихся в браузере. По данным наших измерений [5], это, в сочетании с переписыванием серверной части на PHP, может снизить нагрузку на сервер, например, расходы процессорного времени и оперативной памяти сервера на формирование страницы очереди, в разы или даже в десятки раз. Это может быть очень важно при больших нагрузках. В 2017 году была реализована и внедрена AJAX версия самой загруженной подсистемы сервера, страницы очереди.

Тестирующий клиент также написан на языке Perl с использованием командной оболочки MS Windows cmd.exe посредством использования bat-файлов. Отдельные компоненты

реализованы на языке C с использованием WinAPI. Существуют версии клиента для MS Windows, Linux и Solaris. В настоящее время, главный спрос со стороны организаторов соревнований предъявляется именно на соревнования под Windows, поэтому поддержка клиентов Linux и Solaris приостановлена.

Тестирующий клиент для Windows средствами Win32 API ограничивает вычислительные ресурсы (память, процессорное и астрономическое время) для тестируемой программы. Предпринималось несколько попыток ограничить также права доступа для этой программы, но от них пришлось отказаться из-за неприемлемых накладных расходов по времени.

Время работы решения на одном тесте ограничено несколькими секундами. В последние годы у большинства задач лимит времени на прохождение одного теста составляет 1-2 секунды и редко превышает 10 секунд. Время запуска модуля тестирования должно быть того же порядка величины или меньше, так как при проведении крупных соревнований по программированию тестирующий клиент должен проверить до 5000 решений участников олимпиады на 50-100 тестах — это примерно полмиллиона запусков.

Самое простое решение, доступное в Win32, для запуска процесса с ограниченными полномочиями — это использование системного вызова CreateProcessAsUser. Была разработана версия изолирующей среды, использующая этот механизм, но измерения показали, что работа этого системного вызова занимает 1-2 секунды, что было сочтено неприемлемым.

В 2016 году магистрантом ММФ А.Э. Кимом была разработана и внедрена версия изолирующей среды, работающая как отдельный постоянно запущенный процесс [6]. Это открывает возможность для повышения уровня изоляции: действительно, такой процесс можно запустить из-под учетной записи с ограниченными правами один раз, и накладные расходы на смену учетной записи не будут влиять на каждый запуск задачи. К сожалению, из-за нехватки времени разработчиков, на февраль 2018 года, версия тестирующего клиента, развернутая в системе, этой возможностью не пользуется.

Тестирующие клиенты взаимодействуют с сервером через протокол HTTP, поэтому они могут быть размещены как на той же физической машине, что и сервер, так и на выделенных машинах, физических или виртуальных. В НГУ для тестирования олимпиад используются тестирующие клиенты на выделенных физических машинах.

Отказ от виртуализации обусловлен тем, что виртуализация сопровождается сложно учитываемыми накладными расходами, которые, в свою очередь, могут приводить к перерасходу процессорного времени по сравнению с выделенной физической машиной. В наших условиях это может приводить к ложным вердиктам «превышение лимита

времени». Вопреки распространенному мифу, накладные расходы гипервизоров не ограничиваются производительностью ввода-вывода. Гипервизор участвует во всех операциях управления памятью в гостевой ОС, в частности, при запросах пользовательских процессов на выделение виртуальной памятью и при страничной подкачке. В зависимости от стиля работы программы с памятью, эти накладные расходы могут привести к увеличению астрономического времени работы и процессорного времени по счетчикам гостевой ОС на десятки процентов по сравнению с физической машиной [19].

По аналогичной причине разработчики вынуждены были также отказаться от запуска нескольких клиентов на одной физической машине. По данным измерений [8], сам факт, что на других процессорных ядрах многоядерного процессора что-то исполняется, может приводить к существенному замедлению работы программы, главным образом, из-за конкуренции процессорных ядер за доступ к ОЗУ. Это в примерно равной степени относится как к общему (астрономическому) времени работы программы, так и к процессорному времени, измеренному по счетчикам ОС или по аппаратным счетчикам процессора. При использовании гипертрединга, работа программы может замедляться в несколько раз. Поскольку характер и величина замедления решающим образом зависят не только от характера обращений тестируемой программы к памяти, но и от того, что именно работает на других ядрах, данный эффект никак невозможно контролировать или компенсировать.

Поэтому, как уже говорилось, в системе NSUts все официальные соревнования проходят тестирование на выделенных физических машинах. Поскольку необходимо поддерживать ферму тестирующих клиентов, достаточно мощную для проведения соревнований, эти же клиенты используются и для тренировок, в процессе которых ложные вердикты не так опасны.

6. Заключение

Опыт создания и использования автоматизированной системы проверки заданий по программированию NSUts и ее предшественников позволил выработать требования, предъявляемые к системам такого рода [4]. На их основе разработана архитектура системы, состоящей из трех основных частей: клиентского ПО, сервера олимпиад и тестирующего клиента. Система обеспечивает устойчивую работу под высокими нагрузками и обрабатывает подавляющее большинство нештатных ситуаций, возникающих во время мероприятий. Кроме того, опыт показал, что система достаточно легко адаптируется к изменяющимся условиям (поддержка новых языков программирования, изменения правил соревнований).

Система NSUts была создана для обеспечения проверки решений участников олимпиад по программированию, поэтому главной частью ее использования является проведение олимпиад по программированию всех уровней. Она используется при проведении Открытой Всесибирской олимпиады по программированию им. И.В. Поттосина [17], районных и региональных школьных олимпиад по программированию в Новосибирской области. Например, в первых этапах Всероссийской олимпиады школьников по информатике, проводимых в Новосибирской области с помощью системы NSUts, одновременно участвовало несколько сотен школьников, было проверено свыше тысячи решений. Некоторые студенческие олимпиады собирают около 1000 участников, при этом проверяется в онлайн-режиме более 10000 решений в течение одного тура.

Круглосуточная работа системы NSUts позволила организовать не только поддержку тренировок по программированию, а также и использование ее в учебном процессе для проведения практических занятий по программированию для студентов младших курсов.

Стремительное развитие техники и технологий ставит новые задачи и перед нашей системой, которые приходится решать ежедневно. Поэтому описанная в этой статье история создания системы NSUts еще не закончена.

Список литературы

1. Архив задач с автоматической системой проверки в Екатеринбурге [Электронный ресурс]. URL: http://acm.timus.ru
2. Архив задач с автоматической системой проверки в испанском городе Вальядолиде [Электронный ресурс]. URL: http://acm.uva.es
3. Боженкова Е.Н., Воронков А.Д., Иртегов Д.В., Конышева Е.Н., Черненок С.А., Чурина Т.Г. Модель разграничения прав доступа в системе автоматизированной проверки корректности программных приложений // Вестник НГУ Серия: Информационные технологии. - 2011. - Том 09, Выпуск № 4. - С. 79-85. - ISSN 1818-7900.
4. Боженкова Е.Н., Иртегов Д.В., Киров А.В., Нестеренко Т.В., Чурина Т.Г. Автоматизированная система тестирования NSUts: требования и разработка прототипа // Вестник НГУ Серия: Информационные технологии. - 2010. - Том 08, Выпуск № 4. - С. 46-53. - ISSN 1818-7900
5. Боженкова Е. Н., Иртегов Д. В., Колбин Я. С. Оптимизация производительности веб-интерфейса приложения NSUts средствами динамического HTML // Вестн. Новосиб. гос. ун-та. Серия: Информационные технологии. 2015. Т. 13, вып. 2. С. 13–21.
6. Ким А.Э., Разработка и реализация новой архитектуры тестирующего клиента системы NSUts, магистерская диссертация, ММФ НГУ, 2017, науч. рук. Иртегов Д.В., Чурина Т.Г.

7. Сайт международной олимпиады по программированию ACM-ICPC [Электронный ресурс]. URL: http://icpc.baylor.edu.
8. Свиридов В.С. Измерение и контроль потребления ресурсов программами на машинах с многоядерными процессорами, выпускная квалификационная работа бакалавра, ФИТ НГУ, 2012, науч. рук. Иртегов Д.В.
9. Система NSUts [Электронный ресурс]. URL: https://olympic.nsu.ru/nsuts-new/login.cgi
10. Система olympiads.ru [Электронный ресурс]. URL: http://www.olympiads.ru/
11. Система PCMS2 [Электронный ресурс]. URL: https://neerc.ifmo.ru/school/spb/municipal-participant.html
12. Система Яндекс Контест [Электронный ресурс]. URL: https://contest.yandex.ru/
13. Таблица результатов полуфинала ACM-ICPC 1998 года [Электронный ресурс]. URL: http://neerc.ifmo.ru/past/1997/standings.html.
14. Чурина Т.Г, Иртегов Д.В. Требования к автоматической системе тестирования знаний// Труды VI Международной конференция "Интеллектуальные технологии в образовании, экономике и управлении", декабрь 2009, Воронеж, с. 309-317.
15. I Открытая Всесибирская олимпиада по программированию им. И.В. Поттосина [Электронный ресурс]. URL: http://olympic.nsu.ru/old-site/widesiberia/archive/wso1/2000/index.shtml
16. II Открытая Всесибирская олимпиада по программированию им. И.В. Поттосина [Электронный ресурс]. URL http://olympic.nsu.ru/old-site/widesiberia/archive/wso2/2001/ruls1.shtml
17. XVIII Открытая Всесибирская олимпиада по программированию им. И.В. Поттосина [Электронный ресурс]. URL https://olympic.nsu.ru/widesiberia/2017/news
18. Apache HTTP Server Version 2.2 Documentation [Электронный ресурс] URL: http://httpd.apache.org/docs/2.2/ .
19. [http://ieeexplore.ieee.org/document/6670704/].