ИСП Институт Системного Программирования
Российской Академии наук

ISSN 2079-8156 (Print)
ISSN 2220-6426 (Online)

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том 28, выпуск 4

Volume 28, issue 4

Москва 2016

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН

Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областях системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge. **Proceedings of ISP RAS** is abstracted and/or indexed in:

УДК004.45

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings/

Содержание

Извлечение и анализ информации в современных предприятиях
А.Р. Топчян 7

Масштабируемые учебно-экспериментальные среды для современных предприятий
А.Р. Топчян 29

Генерация функциональных тестов для HDL-описаний на основе проверки моделей
М.С. Лебедев, С.А. Смолов 41

Проверка параметризованных Promela-моделей протоколов когерентности памяти
В.С. Буренков, А.С. Камкин 57

Язык описания шаблонов для генерации тестовых программ для микропроцессоров
А.Д. Татарников 77

Генерация тестовых программ для подсистемы управления памятью MIPS64 на основе спецификаций
А.С. Камкин, А.М. Коцыняк 99

Трансляция вложенных сетей Петри в классические сети Петри для верификации разверток
В.О. Ермакова, И.А. Ломазова 115

Метод оценки эксплуатируемости программных дефектов
А.Н. Федотов 137

Поиск ошибок доступа к буферу в программах на языке C/C++
И.А. Дудина, В.К. Кошелев, А.Е. Бородин 149

Поддержка стандарта OpenMP 4.0 для архитектуры NVIDIA PTX в компиляторе GCC
А.В. Монаков, В.А. Иванишин 169

Модель поведения объектов, подверженных спонтанному изменению, в прецедентном подходе к управлению
В.Н. Юдин, Л.Е. Карпов *183*

Некоторые задачи на графовых базах данных
Р. И. Гуральник *193*

Динамическая компиляция выражений в SQL-запросах для СУБД PostgreSQL
Е.Ю. Шарыгин, Р.А. Бучацкий, Л.В. Скворцов, Р.А. Жуйков, Д.М. Мельник *217*

Обзор современных методов планирования движения
К.А. Казаков, В.А. Семенов *241*

Table of Contents

Information Retrieval and Analysis for a Modern Organization
Artyom Topchyan..... 7

Scalable Sandbox Environments for a Modern Organization
Artyom Topchyan..... *29*

A Model Checking-Based Method of Functional Test Generation for HDL Descriptions
M.S. Lebedev, S.A. Smolov. *41*

Checking Parameterized PROMELA Models of Cache Coherence Protocols
V.S. Burenkov, A.S. Kamkin..... *57*

Language for Describing Templates for Test Program Generation for Microprocessors
A.D. Tatarnikov *77*

Specification-Based Test Program Generation for MIPS64 Memory Management Units
A.S. Kamkin, A.M. Kotsynyak..... *99*

Translation of Nested Petri Nets into Classical Petri Nets for Unfoldings Verification
V.O. Ermakova, I.A. Lomazova. *115*

Method for exploitability estimation of program bugs
A.N. Fedotov..... *137*

Statically detecting buffer overflows in C/C++
I. Dudina, V. Koshelev, A. Borodin *149*

Implementing OpenMP 4.0 for the NVIDIA PTX architecture in GCC compiler
A.V. Monakov, V.A. Ivanishin..... *169*

Model of spontaneously changing object behavior in case control approach
V. N. Yudin, L. E. Karpov..... *183*

Some problems on graph databases
R.I. Guralnik *193*

Dynamic compilation of expressions in SQL queries for PostgreSQL
E.Y. Sharygin, R.A. Buchatskiy, L.V. Skvortsov,
R.A. Zhuykov, D.M. Melnik *217*

An overview of modern methods for motion planning
K.A. Kazakov, V.A. Semenov *241*

Information Retrieval and Analysis for a Modern Organization

Artyom Topchyan <a.topchyan@reply.de>
Yerevan State University,
Alek Manukyan 1, Yerevan, 0025, Armenia

Abstract. With the growing volume and demand for data a major concern for an Organization is to discover what data there actually is, what it contains and how it is being used and by who. The amount of data and the disparate systems used to handle this data increase in their number and complexity every year and unifying these systems becomes more and more complex. In this work we describe an Intelligent search engine system, specifically designed to tackle the problem of information retrieval and sharing in a large multifaceted organization, that already has many systems in place for each Department, which is an integral part of a joint Operational Data Platform(ODP) for data exploration and processing.

Keywords: data-driven projects; information retrieval; streaming processing; mesos; kafka

DOI: 10.15514/ISPRAS-2016-28(4)-1

For citation: Topchyan A.R. Information Retrieval and Analysis for a Modern Organization. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016, pp. 7-28. DOI: 10.15514/ISPRAS-2016-28(4)-1

1. Introduction

With the growing volume and demand for data a major concern for an Organization is to discover what data there actually is, what it contains and how it is being used and by whom. The amount of data and the disparate systems used to handle this data increase in their number and complexity every year. This trend is driven by business and technical demand, which especially stems from the need for more Data-Driven projects [1][2][3]. Data-Driven projects aim at increasing the quality, speed and/or quantity of information gained from Data collected by the Organization. But it is very challenging to move ahead with such projects without access to a defined model to handle data exploration and processing. This leads to most of the project time being spent on actually finding out information about the existing data, finding people involved, data owners and where the data lies, as opposed to actual analysis. As described in our previous work this can be quite costly time and resource and each stage of a Data-Driven project is impacted by this. There is currently no single

accepted approach for tackling such problems, so we described and implemented a joint Operational Data Platform(ODP) for data exploration and processing, which aims to be an end-to-end platform for solving the issue of managing large amounts of data and information about this data by means of scalable automation and information extraction [1]. This platform has been successfully implemented and is actively used by large organizations to implement Data-Driven projects. A high level overview of the entire solution as presented in our previous work is outlined in Fig.1. One of the main components of the platform was the Information Marketplace, which is an intelligent search engine system, specifically designed to tackle the problem of information retrieval and sharing in a large multifaceted organization, that already has many systems in place for each Department. As outlined above, we believe this to be the key challenge when exploring possible use cases for Data Scientists. In this work we outline, the problem definition and technical implementation, in more detail. We will first outline why we believe such a solution is required and how it fits into an existing, ever changing landscape of an organization. Then we will outline in detail what data is used and what and how information is extracted from it in order to build a search index over all project information and data, that an organization has and continuously develops in the future. We will specifically concentrate on the architectural and algorithmic scalability challenges of extracting information from large and varied datasets using existing methods.

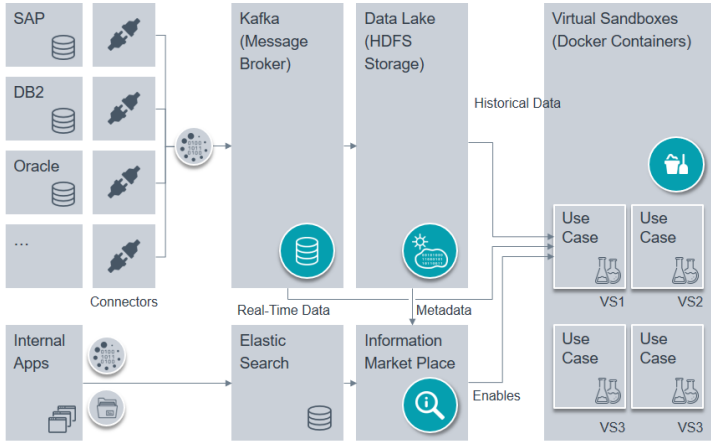


Fig. 1. Operational Data Platform

2. Information Retrieval and Sharing

One of the biggest challenges faced by an Organization when exploring possible use cases for Data Scientists is in experience knowledge sharing and transfer. Large Organizations have a large number of Departments, which vary widely based on their size, project they take on and the way these projects are completed and documented.

This leads a large variety of data sources, column names and documentation being created on the same subject by a large number of stakeholders from different Departments some of whom might not be part of the Organization anymore. This leads to challenges for Data Scientists and the IT Department, which have to identify the relevant information and people or documents describing the data, especially when the project involves more than one data source. The most important consideration here is time spent on actually finding out if data is available and similar issues as opposed to more productive data analysis.

To bridge this issue, there are large undertakings for an organization wide change management process, which pushes for standardization. On a technical level this changes translate to the centralization and standardization of project related documentation as well as rigid data views in a central database. Due to the complexity of the data and the Organization itself an Enterprise Data Warehouses or a Wiki-System cannot directly solve this issue, while satisfying all the requirements on structure and intelligence. This leads to the creation of Organization wide specialty tools, data/knowledge repositories and integration layers providing each Department with access and management capabilities, in order to adapt to the specific requirements of the Organization.

Essentially what this entails are a full organizational restructure to create solution for data and information management. In reality, this is a vast and complex process and can cost a large amount of money and resources from the side of the Organization and in some cases might decrease productivity. Each individual Department has an approach of managing projects and in most cases such a monolithic system allows for less flexibility for individual Departments. This may lead to decreased productivity. It is often unrealistic to expect full and informed cooperation from each Department and its staff in order for each project to be documented, every data column described, every project contributor to be listed and all the interconnections between documents and data of multiple Departments being identified and documented. This is further complicated by the fact, that there is always more data and information each year, so all of the created documentation should be retroactively update periodically. The learning period for a complete change and standardization of such processes and the time required to update all of this information, can bring an entire Department to a halt for an extended period of time.

There are also a large number of technical considerations, such as file format support, performance of the search and indexing. Data Scientists and the Business Department would like to see changes to any information as fast as possible.

To summarize, lets pose a list of requirements we accumulated based on our experience working with teams of Data Scientists at large organizations:

1. **Index decentralized documentation repositories.** Each Department should be able to retain their knowledge repositories with little to no functional changes. Data should be acquired from these repositories with as little effort as possible from the Departments side.

2. **Support a large number of formats.** Each Department should be able to use any binary file format for their project documentation.
3. **Document Metadata has to be preserved.** Such properties as authors, auditors, names, comments should all be extracted and preserved and be uniquely identifiable.
4. **Low Latency Indexing.** Changes to or new documents should be reflected as fast as possible.
5. **Content Indexing.** As a minimum all textual content has to be searchable, there has to be a possibility to later extend this to non-textual content, such as audio and image files.
6. **Author extraction.** Each document must be tagged automatically with the authors of that document. This can be extracted depending on the metadata or by analysis of the document or related documents.
7. **Document summarization.** A short summary of the Document would be very beneficial when exploring a large collection. Giving context knowledge about document can often be use full to understand the content at a glance. This could be implemented by extracting important sentences or keywords.
8. **Context search.** Each document describes a specific context or project in relation to the organization or the Department. This representation of the document if very valuable specifically for finding similar projects, but which are not specifically referring to specific data sources, projects or Departments.
9. **Cluster by Department context.** It is often useful to look at groups of documents referring to specific contexts as opposed to exploring them one by one. This is specifically interesting to see if documents of one Department refer to a context usually associated with another Department.
10. **Data source search.** It should be possible to search for specific data sources referred to in the documents. Meaning every document should point to specific data sources it mentions. Used with the document clustering approach, documents that have no mention of any document can also be tagged.
11. **Data source summarization.** It is very important to present an outline of the data, that is actually available in connection to a Data source a document refers to.
12. **Flexible faceting.** All of this extracted information should be made available as facet views, available when using the tool to explore the organizational information.

13. **Decoupling of functionality.** As we described above, the landscape of the organizational data and systems is always changing and there should be no hard dependencies between specific Departments or types of analysis. If a Department stops producing documentation in a specific format or at all or author information should no longer be stored, the systems should not require a substantial rewrite to remove these functions.

14. **Flexibility for extensions.** Similar to the previous requirement the system should be easily extensible if further analysis is required, such as for example support for extracting information from image data. This should also not invalidate the previous requirement and require a substantial modification to the system. New attributes or indices should be handled transparently.

15. **Performance.** This tool will be used by Data Scientist and the Business Department in their day to day work, this means it should allow for a responsive experience and support many hundreds of concurrent users.

16. **Scalability.** The entire solution entails fairly complex computation as there are potentially tens of gigabytes of documentation (from many Departments) and data meta information (data schemas) as well as terabytes of data that has to be analysed. This means the solution should be scalable in order to provide the similar levels of performance independent of new documentation or data source.

17. **Fault tolerance.** This is more of an exploratory tool and as such full end-to-end correctness is not necessarily required. By correct we mean, that no documents are processed more than once and all results are always consistent throughout the solution. Nevertheless, we need to provide a certain level of fault tolerance thought the system so little to no supervision is required to ring the system to a consistent state. In this case we are looking more at the system being eventually consistent [12].

This is not an exhaustive list, but it outlines the base requirements a system for information retrieval and sharing should fulfill. It is list of complex functional requirements which we need to map to a technical problem definition and implementation.

In the next section we will explore this definition and the architecture as well as the algorithmic implementation of this tool, which we called the Organizational Information Marketplace.

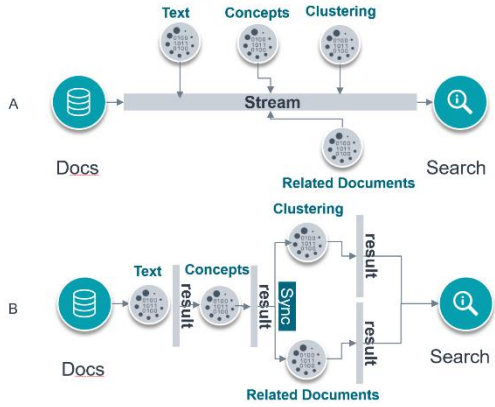


Fig. 2. (a) Data-flow requirements in the context of stream processing. (b) Data-flow requirements in the context of batch processing.

3. Architecture

There are significant architectural and algorithmic considerations when mapping the requirement set outlined in the previous chapter to a technical implementation. To achieve all the requirements for performance this has to be distributed application running on multiple machines, which introduce a large set of benefits and problems.

First let's consider the architectural side of the problem, before outlining how the required information is extracted and modelled. To outline the architecture, we first need to define, what input we have and what required output has to be produced.

Input. Our input is a collections of documents for each Department/data source the organization has. These collections are unbounded and may increase and decrease as well as change over time and so can the data sources. Currently a single collection of documents can be larger than ten gigabytes with documents containing hundreds of words. Additionally, such a system should be ready to process raw organizational data as well. This would be important to analyze the actual data content as opposed to definitions. This can become extremely large with terabytes of data coming every month.

Output. As the output we require the document with additional extracted or calculated metadata information, such as the summary of the document, the semantic/contextual representation of the document, similar documents and others. This information should be exposed through an interface, that supports a rich set of text based queries. As the entire content of the document has to be indexed with all the additional extracted information, we are looking at indexes tens of gigabytes in size per Departments.

Based on the requirements we need to find a way to get from the input to the output as fast as possible, whilst applying a non-fixed number of potentially process

intensive transformation functions in sequence on an unbounded collection of documents. Additionally, the latency of processing, should not be affected by the number of documents in a collection at any point in time, meaning there should be a defined way of scaling each step independent of the others. It is an accepted approach in the Industry to frame this problem in the context of stream processing [5][4] as opposed to a periodic batch process. In stream processing each collection of documents can be represented as a separate stream of documents, which changes constantly over time. Each new element of the stream is processed one by one, out-of-order in parallel. The output of each transformation is represented as an another stream of datum and can be directly consumed by another transformation. This is quite different from a periodic batch approach, where each collection is considered static at the point in time and all transformations have to be applied to the entirety of collection. This is disadvantageous for our case as certain transformations can depend on others, which means a transformation has to be applied on all documents before proceeding to the next step. Because This means we need to either join the transformations together into more monolithic blocks or create large intermediary collections of transformed documents and orchestrate the order of the batch jobs. In our experience, this is highly impractical, as this entails either very large, not flexible transformation steps or a complex scheduling problem. Such problems are very tricky to tune to scale correctly. Not to mention the fact, that this process be definition is high latency, but this of course depends on the number of documents and the complexity of transformations at any point in time. The difference between the two approaches is highlighted in Fig.2. The main issue comes from the necessity to sync between transformations. A transform has to be notified that a previous step has been completed. This introduces complexity and latency. In our approach we settled on decreasing the granularity of data slices and define a stream of data. This in our opinion allows for flexibility of adding and removing steps at essentially any point. As opposed to only passing simple events around [7] we model the data as an unbounded stream, so we can process these elements on-by-one or in larger batches as well without intermediate storage. This greatly simplifies the amount of state that has to be tracked in the entire solution and as a side effect, in our experience, leads to more concise, performant and testable services. But we still need to define an approach for scaling this type of solution and more importantly ensuring a certain amount of automatic fault-tolerance to make the solution as flexibly and low-maintenance as possible.

3.1 Components for Fault Tolerance and Scalability

ThereisasignificantconnectionbetweenFault-ToleranceandScalability. Scalability is usually most directly solved by trying to parallelize the costly process, most often, by launching more instances of the same service and balancing the work between each instance. But according to the CAP theorem, which states that it is impossible for a distributed computer system to simultaneously provide Consistency (all nodes see the same data at the same time), Availability (every request receives a response about

whether it succeeded or failed) and Partition tolerance (the system continues to operate despite arbitrary partitioning due to network failures. To this end the most important architectural block of a streaming processing systems is the representation of the "stream". As we described we need a proven way of storing unbounded amounts of information, which has to be tolerant to failures and scale to a large number of events and services.

3.1.1 Stream component

A widely accepted approach to this is using a distributed message broker, such as Apache Kafka [25], which is a high throughput, distributed and durable messaging system. Each stream can be represented as a topic in Kafka. A Topic is a partitioned log of events. Each partition is an ordered, immutable sequence of messages that is continually appended to—a commit log. The messages in the partitions are each assigned a sequential id number called the offset that uniquely identifies each message within the partition [25]. This is a flexible representation as it allows us to create many topics per stage, which can be consumed by processors in a stage(consumers). Each Kafka consumers is part of a consumer group and Kafka itself is balancing the partitions in the topic over the consumer processes in each group. This means that it is simple to achieve processing scalability just by launching more consumer processes. As data is automatically distributed and consumption is balanced, Availability is straightforwardly addressed by dynamically launching more instances of the service and relying on a balancing mechanism. On the other hand, as described in [9][8] streaming systems are most sensitive to Partitioning and Consistency problems. The generally accepted approach to try and solve this [10] [29] is essentially based on offset tracking. Offset tracking is tracking how far along in the stream the processing service is. If each instance of the service has to make sure it processed a single or batch of event before requesting new events. It would be quite expensive to commit such offsets for every event, so offsets are usually committed is small batches. This is supported as a core functionality within Kafka itself, which offers approaches to store offsets in a separate topic as well as allows for automatic batching of events. Using Kafka, we introduce a scalable and fault tolerant representation of a stream, both on the storage and processing level.

3.1.2. Service Scheduling component

With the described approach each service is only responsible for deciding how many events to process and from which offsets to start and work balancing, storage and transfer are handled by the stream representation. Kafka has no control over the consumer processes as well has no idea how costly each operation or what state(offsets) have to be tracked. This means it is simple to scale processes just by launching more processes, but making sure all events were processed correctly during failures, as well as quickly is up to the consumer itself. To this end we require a defined way for dynamically scaling process as well as restarting and retrying in case

of consumer outage. To solve this issue, we can exploit the fact, that with the proposed way of handling data storage and state, each service is essentially immutable. Which essentially means a crashed service does not need to be restarted or recovered, but we can just start a new copy, which will catch up to the state the previous service was automatically. Coupled with the fact, that we have a large number of machines at our disposable the problem of service fault tolerance and scaling becomes a process scheduling problem. Technically we have a set amount of resources for each stage, each stage takes a certain amount of time to compute and we are trying to reach a certain amount of latency. Using this information, we can approximate how many processes have to run and where in order to achieve the required latency. We require a component, that can decide whether to start, stop or restart processes. This is an accepted approach to handle scaling largescale distributed application and has been successfully implemented in the industry [11] [27] by means of a global resource manager. Such a resource manager is essentially a higher level version of an Operating Systems Kernel, but running on many machines. Applications decide based on their processing time and requirement, what resources they require. The resource manager tries to accommodate this. If the service crashes or the computing node fails, it can be transparently restarted somewhere else assuming the application can transparently handle something like this. Which our proposed approach can. Here we use Apache Mesos, a commonly accepted High-Availability implementation of such a system. Such a system has also proven to be very flexible to new services or any adaption [13]. The system just needs to know what resource and what processes to run.

3.2 View component and Architecture Implementation

The last thing missing from our architecture is the "view “layer. This is where the output of the processing stages is stored in its final form, as well as as any extra data. Based on the outlined requirements we need a flexible and scalable storage systems, that supports a large scale of analytical queries. Considering the fact, that most of our data are documents. It is more efficient to store the output in a Search Engine. In this work we use Elasticsearch [26] due to its proven scalability and performance, specifically in the context of stream processing. Based on this we can define our general purpose architecture for building a scalable and Fault-Tolerant stage based Event Driven application for the purpose of extracting information, context from an organizations documentation and allowing this to be efficiently queried. A high level overview of the architecture is presented in Fig. 3. It should be noted, that in this work we concentrate on the architectural aspect of scaling and fault-tolerance and do not define a novel approach, but more an architectural framework using the contexts from these projects in order to efficiently and accurately solve the problem for this specific use case.

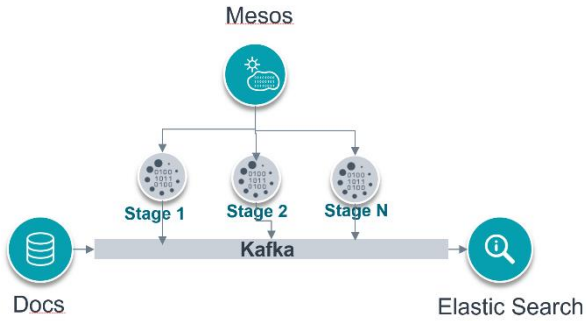


Fig. 3. Information Marketplace Architecture

4. Processing Stages and Implementation

Based on the described architecture we need to map the requirements described in the previous chapters to a set of stages of transformation a single document has to go through and queryable views, which are the end result of these transformations. First lets define the views, which are needed to satisfy the requirements:

- Document Full-Text Search View
- Search View based on Author, Time, File-Format, Department, Data source
- Search View based on extracted keywords, contexts, summaries
- Related document View, based on contexts and keywords

These view are essentially queries against the Elasticsearch database, where the data will be stored and so we will not outline them in much detail outside of the stage definitions. What is required is a full representation of the document, data sources and Departments. Based on this we can define the stages of transformation:

1. Document load from source
2. Document original format to plain text conversion with metadata extraction
3. Map document to specific organizational data source
4. Extract document summary
5. Extract semantic representation of the document(contexts)
6. Find closely related/similar documents
7. Find Departments a data source might refer to
8. Index document

We will now outline in more detail what each stage does and how it is implemented as well as scalability and fault tolerance considerations for each stage.

4.1 Load document

- Input — Directories to watch for files
- Output — Stream of document creation, update, delete

Documents may reside on a filesystem in every Department. We need to detect all the files already there and detect the creation of new files to fulfill the low-latency requirement. We implemented a distributed directory watcher, which watches for filesystem event and emits these events into a Kafka stream. Each watcher keeps a reverse index of the filesystem it is watching. This is required in order to track what files we have already processed, as no duplicates should be processes if possible. The filepath and the last modified date are used as the offset key in the reverse index. The offsets are only committed if the file was detected, and has not been modified in the last few seconds. This is done in order to avoid reading files, which are still being written to. With this approach we ensure fault tolerance and exactly once processing of all files. Because this index is stored as a collection of offsets in a Kafka topic no duplicates or missing documents will be processed during service or full node failures. We implemented these watcher using the Kafka Connect Framework [29], which is a framework tightly coupled with Kafka and offers some utility methods to more simply write data to Kafka. Each watcher writes an event to Kafka containing the source of the event, the nature of the event and the filepath. This stage is then defined as a collection of such watchers accessing multiple organizational sources distributed throughout the organization.

4.2 Document conversion

- Input — Stream of document
- Output — Stream of document metadata and text representations

We need to extract information from a large variety of document formats. As most of the document ion is written we need to convert it to a simple text representation as opposed to a native binary format such as Microsoft Word or PDF. For parsing a large variety of format we use Apache Tika [28]. Tika also allows us to extract a full set of file metadata from the file, such as authors, modified dates, owners and etc. This stage produces three different text representations:

1. HTML bases representation, preserving the formatting, useful for display
2. Plain Text representation
3. Filtered text, based on language stop words are filtered, characters are normalized

As document arrive in a highly asynchronous manner from many data sources in a single stream we can implement a distributed stream processing application running on top of Mesos to convert this representation to Text. Essentially each streaming consumer receives the file, extracts the text and metadata and materializes the result to another stream. Processes can be pre-allocated based on the number of documents

committed and average processing time, this is implemented keeping the main ideas of [14].

4.3. Data source mapping

- Input - Stream of documents
- Output - Stream of documents with Organization data sources labelled

As we outlined in previous sections understanding which data a document is referring can be very beneficial to Data Scientists, when trying to build a project. This info can both be used to find out what the data is about or the reverse, which data matches a certain topic. What we have as input is a lot of documents with no extra information apart from their content and origin. We need to map each document received to one or more data sources. If we had some documentation for each data-source of the organization we could rephrase this issue as a context matching problem, this could be solved by one of our other stages of analysis. But it is very rarely the case, that an Organization has any significant amount of information on every data source they have, quite often the information is there, but has been lost in the thousands of documents scattered across Departmental repositories. This is the aspect of the problem we are trying to solve in this stage. As it stands assuming we have the technical definition of each data source, schemas, column definitions and etc., we need to find instances of documents that contain references on these attributes. A straightforward approach would be to scan through each document and count how many instances of all permutations of all columns are contained inside the document. But if we consider the complexity of such calculations for a typical organization, it becomes a very suboptimal approach. For example, this stage has to function with low latency processing at an organization with dozens of different database systems, which have thousands of tables in total and each table containing potentially hundreds of columns. Such an organization has tens of thousands of pieces of documentation, with a hundreds of documents being modified or created every day. Such documents usually contain hundreds of words on average. This is an obviously suboptimal calculation, which is not practical to compute with low latency. For example, let's say we have 3000 tables with 100 column search and 10000 documents with 500 words each. This would mean for one document we would need to do $500 \times 300000 = 150000000$ not simple comparisons. Another downside of such an approach is, that we would need to keep all data related information in memory as a collection, which would lead to large resource requirements. To this we adopt a method similar to the approach described in [21], which rephrases this issue as a comparison of sets of data. Indeed, we can efficiently represent a document and each data source specification as sets, and then our problem would be to find a set most similar to each incoming document. Of course this would mean, that we are taking the overlap of a document and a full set of data columns, which would never truly fully overlap, but as described in [21] this would give us a good approximate match and we could then analyse which of the matched data source is truly mentioned in the document. For this we build an efficient representation of each data source specification using the Minhash (min-wise

independent permutations locality sensitive hashing scheme) algorithm [20]. 4.3.1. Minhash

The Jaccard similarity coefficient is a commonly used indicator of the similarity between two sets. For sets A and B, the Jaccard similarity is defined as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cup B|}$$

We can phrase this as the ratio of the size of the intersection of A and B to the size of their union. J lies in 0 <= J <= 1. More specifically, if J = 1 the sets are identical, if J = 0 there are no shared members otherwise the value can be interpreted as “the probability that a random element from the union of two sets is also in their intersection” or “the probability that a randomly chosen element chosen from one of the sets is also in the other set.”. This is widely accepted measure of similarity, but it is still quite expensive to compute [21] especially for our case. To this end [20] describes an algorithm to estimate the Jaccard similarity (resemblance) between sets of arbitrary sizes in linear time using a small and fixed memory space. It is also quite suitable to use in a stream processing context due to the low computation time and compact representation. First lets define a matrix representation of the comparison of multiple sets. Let’s define matrix M where m_{ij} is 1 if element i is in set S_j and 0 otherwise. Let’s denote N as the number of all unique elements in each set. Then for N let’s take K random hash functions $\{h_1, h_2, \dots h_k\}$, which map each element to a random unique ID from[N]. Then for k counters $c_1, c_2, \dots, c_k$ we can define the Minhash for the set S as: Then set mj(S) = cj and we can define the Jaccard distance estimate as:

$$JS_k(S_i, S_{\hat{i}}) = \frac{1}{k} \sum_{i=1}^{k=1} I(m_j(S_i) = m_j(S_{\hat{i}}))$$

, where I(σ) = 1 if σ = 1.

This also gives us a compact representation, which we can calculate and store for each data source specification, meaning we only need to calculate the Minhash for each incoming document. But this is still computationally expensive as we still need do the computation at least O(N_{tables}), meaning the query cost increases linearly with respect to the number of data sources. A popular alternative is to use Locality Sensitive Hashing (LSH) index.

4.3.1 Minhash-LSH

One approach to implement LSH is to "hash" each element many times, so that similar items are more likely to be hashed to the same bucket. We can then just look at the pairs, which ended up in the same bucket. The main idea as described in [21] is that most of the dissimilar pairs will never hash to the same bucket, and therefore will never be checked, and the number of false positives is low.

Algorithm 1 Min hash on Set S

```
for i = 1 to N do
    if S(i) = 1 then
        for j = 1 to k do
            if h_j(i) < c_j then
                c_j ← h_j(i)
            end if
        end for
    end if
end for
```

For a Minhash representation computed as described above, an effective way to choose the hash functions is to take the matrix of the representation and partition it in to b partitions with r rows each. Then we use a hash function for each partitions and calculate the mapping to many buckets. Each partitions can use the same has function and keep a separate bucket for each partition. This means columns with the same vector in different bands will not hash to the same bucket. Then we can check each band for matches in each bucket.

Based on the describes algorithms we create a Minhash representation for all data source specifications and build and LSH index over these. This allows us to very efficiently query for the most similar data sources for an incoming document in a stream. We take the 3 top matches and find, which one actually is most referenced inside the document. This is attached to the document as extra metadata and is written to another stream. Due to the implementation this can be easily scaled by launching more instances as the Minhash representation is fairly compact.

4.4 Document summary

- Input — Stream of documents
- Output — Stream of documents with added summary

This stages receives a stream of documents as input and outputs the same document with an additional summarization field attached to the metadata. A text summary in this context is a set of most important phrases/sentences of the document. Such kind of method are very sensitive to stop words, words that do not impact the semantic meaning of the text, such as "and" or " and etc. So this stage relies on the third form of the text representation, which is already filtered and cleaned. As we only have unstructured documents and most organization do not tag or write abstracts for each document, we need to rely on an unsupervised method for text summarization [19]. We summarize the given text, by extracting one or more important sentences from the text. This summarizer is based on the "TextRank" algorithm [18], which was chosen when considering its performance and a simpler implementation for key-sentence extraction vs keywords.

4.4.1 TextRank

In the TextRank algorithm [18], a text is represented by a graph. Each vertex corresponds to a word type or sentence. Vertices v_i and vj are connected with a weight w_{ij}, which corresponds to the number of times the corresponding word types co-occur within a defined window in the text. The main objective of TextRank is to compute the most important vertices of the text. If we denote the neighbors of vertex v_i as Adj(v_i), then the score S(v_i) is computed iteratively using the following formulae:

$$S(v_i) = (1-d) + d \times \sum_{v_j \in Adj(v_i)} \frac{w_{ji}}{\sum_{v_k \in Adj(v_j)} S(v_j)}$$

Vertexes with many neighbors that have high scores will be ranked higher. We are of course not interested in keyword extraction, but sentences based summarization. To this end as shown in [18] it is we can adapt the same algorithm to sentences. Each vertex will represent a sentence and, as "co-occurrence" is not a logical relation between sentences, defining another similarity measures. A similarity of two sentences is defined as a function of how much they overlap. The overlap of two sentences can be determined simply as the number of common tokens $w_1^i \ldots w_N^i$ between the lexical representations of the two sentences S_i and S_j. In [18] it is defined as:

$$Sim(S_i, S_j) = \frac{|\{w_k | w_k \in S_i, w_k \in S_j|\}}{log(|S_i|) + log(|S_j|)}$$

By applying the same ranking algorithm 0.3 and sorting the sentences in reversed order of their score we can find the most important sentences, that represent the document. This algorithm was implemented and applied in a stream of document to produce the output of the stage. The algorithm performance only depends on the length of the documents, and based on our average length of documents, the processing speed satisfies out latency requirements.

4.5 Context extraction

- Input — Stream of documents with extracted information
- Output — Stream of documents with context
- Output — Context Model for each Department

We require a representation of each document in the context of the Department it is generated by, as well as other Departments. To do this we need a model, that captures the concepts most used and referred to by projects in the Department. This would give us a valuable contextual representation of key concepts. This could be achieved by extracting keywords or popular words similar to the methods described in the Document Summarizing stage, but what this kind of approach lacks is the document level representation. We want to consider the context both on the level of separate

concepts as well as groups of these concepts(documents). This would give us a more complete view of the context of the Department as well as allow us to extract, such a contextual representation for any document. Meaning we would see how a document relates to work usually carried out in the Departments. We also want to analyse the extracted context to find similar concepts. Another requirement we have is that any model used should be flexible and require little maintenance to be updated in the future. For this Probabilistic Topic Models are a widely accepted choice [17]. Using these models, we can infer the semantic structure of a collection of document using Bayesian analysis. These models are fairly popular and have been used to solve a wide range of similar problems [23] [24]. In this work we specifically adopted the Latent Dirichlet allocation (LDA) model.

4.5.1 Latent Dirichlet allocation

One of the most popular models is Latent Dirichlet allocation (LDA) [17]. LDA is a model for extracting underlying topical information from unstructured data. LDA models each document as being represent by multiple "topics". The topics are represented as a collection of words, or more specifically a probability distribution over a fixed vocabulary of terms. In LDA documents in a corpus are assumed to be generated from a set of K topics. Each topic k is represented by a Dirichlet distribution over the vocabulary:

$$\beta_k \sim Dir$$

Each document d is also repented as a Dirichlet θd.Eachword w indocument d is then assumed to be generated by drawing a topic index z_{dw} from a multinomial distribution Mult(θ(d)), then choose a word wdn for that topic from Mult ($\varphi_{z_{dw}}$). By assuming this generative process, we can go backwards and estimate these matrices from a collection of documents. In this case a collection per Department, which will generate a model per Department. There are a number of way to train such a model [17] [16]. In this work as we require low latency inference as well as flexibility to do automatic model updates, we adopt the Online Variational Bayes algorithm [16]. This method also has the quality of being static in memory as the entire stream of documents does not have to be loaded fully when required. Mini-Batches of documents are processed to infer these matrices. As with most vibrational algorithms, the processing consists of two steps:

- The E-Step: Given the minibatch of documents, updates to the corresponding rows of the topic/word matrix are computed.
- The M-Step: The updates from the E-Step are blended with the current topic/word matrix resulting in the updated topic/word matrix.

Each document received is added to the model and after that the topics are extracted, this allows us to continuously update the model as well as extract topics from documents. It is more effective to update the model with minibatches [16]. A mini-batch is a small batch of documents. In our case, this is usually set to 100. This is also

efficient for throughput as well as simplified offset tracking. Offsets are committed only after a mini-batch of documents has been successfully processed. After each iteration the model is save to disk and only then the offsets are committed. This means we ensure the update model always contains all the documents and is up to date. The topics are attached to the document with their distribution, this is necessary later on for ranking text matches. A model is produced per Department in order to captured the unique context of each. This means the stage contains at least one instance of each model, but the output of the stage is still a single stream. Related documents as we are extracting the context of each document via the LDA model and attaching this information with the specific weights to the document that is indexed in the Search Engine, it is fairly trivial to find contextually related documents, just by including this field and its weights when executing text searches

4.6 Related documents

As we are extracting the context of each document via the LDA model and attaching this information with the specific weights to the document that is indexed in the Search Engine, it is fairly trivial to find contextually related documents, just by including this field and its weights when executing text searches.

4.7 Departmental context

- Input — Stream of Documents
- Output — Stream of documents with attached "Department" label

If we take the described LDA Topic Model for each Department as a contextual representation of the concepts used, we can use this as the representation of what the Department does in the context of the organization. We essentially have a collection of sets of topics per Department and we want to find which other Departments this document relates to. We can exploit the Minhash based set matching approach we used in the previous stages and apply it to the problem of matching a document to an organization. We can define an LSH index over each set of topics for each organization. Then for any incoming document we need to find the most similar set of topics from each organization a document relates to. Based on this we can extract a very rough estimation of "related Departments".

As we are using the same efficient representation as with the data source mapping stage we can efficiently calculate this with low resource requirements.

4.8 Document indexing

- Input — Stream of documents with extracted information
- Output — Elasticsearch indexes

This is the stage that actually indexes the data into a search engine, in this case Elasticsearch is used to create a reverse token index of the documents.

We implemented these watcher using the Kafka Connect Framework [29]. Each index writes an event from Kafka into Elasticsearch. This stage is then defined as a collection of such indexers accessing multiple sources throughout the organization. The data is keyed with the filenames, event timestamp, and a unique offset is generated via a has function to ensure, that no duplicate data is written into Elasticsearch even during node faults. We can rely on Elasticsearch's idempotent write semantics to ensure exactly once delivery. By setting ids in Elasticsearch documents, the connector can ensure exactly once delivery by simply overwriting the data and not creating new records. You can restart and kill the processes and they will pick up where they left off, copying only new data. As opposed to the data loading stage we can read from and write to Elasticsearch in parallel and so we can launch more services to scale this stage based on amount of input.

5. Conclusion

We proposed a flexible way for discovering data and interconnections of the data, based on metadata, functional descriptions and Documentation, in an automated and intelligent way, while not requiring a full Departmental restructure. We outlined the set of requirements desired by many Organizations in the industry. We described a scalable, fault-tolerant and flexible Architecture as well as technical and algorithmic implementation, that satisfies all these requirements. We believe the approach, proposed solution and its architecture provide a solid basis for implementing similar information retrieval solutions at large Organizations, that are starting to explore Data-Driven projects.

References

[1]. Topchyan A.R. Enabling Data Driven Projects for a Modern Enterprise. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 3, 2016, pp. 209-230. DOI: 10.15514/ISPRAS-2016-28(3)-13

[2]. Rahman, Nayem, and Fahad Aldhaban. "Assessing the effectiveness of big data initiatives."2015 Portland International Conference on Management of Engineering and Technology (PICMET). IEEE, 2015.

[3]. Davenport, Thomas H., and Jill Dych´e. "Big data in big companies."International Institute for Analytics (2013).

[4]. Dunning, Ted, and Ellen Friedman. Streaming Architecture: New Designs Using Apache Kafka and Mapr Streams. O'Reilly Media .2016.

[5]. Marz, Nathan, and James Warren. Big Data: Principles and best practices of scalable real-time data systems. Manning Publications Co, 2015

[6]. Michael Hausenblas and Nathan Bijnens. Lambda Architecture. http://lambda-architecture.net, 2015.

[7]. K. Mani Chandy. vent-Driven Applications: Costs, Benefits and Design Approaches, California Institute of Technology, 2006.

[8]. Akidau, Tyler, et al. "MillWheel: fault-tolerant stream processing at internet scale."Proceedings of the VLDB Endowment 6.11:1033-1044, 2013.

[9]. Zaharia, Matei, et al. "Discretized streams: Fault-tolerant streaming computation at scale."Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013.
[10]. Akidau, Tyler, et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, outof-order data processing."Proceedings of the VLDB Endowment 8.12: 1792-1803, 2015.
[11]. Verma, Abhishek, et al. "Large-scale cluster management at Google with Borg."Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015.
[12]. Boritz, J. "IS Practitioners' Views on Core Concepts of Information Integrity". International Journal of Accounting Information Systems. Elsevier, 2011.
[13]. Netflix. Distributed Resource Scheduling with Apache Mesos. http://techblog.netflix.com/2016/07/distributedresource-scheduling-with.html
[14]. Newell, Andrew, et al. "Optimizing distributed actor systems for dynamic interactive services.". Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016
[15]. Cohen, William, Pradeep Ravikumar, and Stephen Fienberg. "A comparison of string metrics for matching names and records.". Kdd workshop on data cleaning and object consolidation. Vol. 3, 2003
[16]. Hoffman, Matthew, Francis R. Bach, and David M. Blei. "Online learning for latent dirichlet allocation.". Advances in neural information processing systems, 2010
[17]. Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation. "Journal of machine Learning research 3.Jan: 993-1022, 2003
[18]. Mihalcea, Rada, and Paul Tarau. "TextRank: Bringing order into texts. "Association for Computational Linguistics, 2004.
[19]. Hasan, Kazi Saidul, and Vincent Ng. "Conundrums in unsupervised key phrase extraction: making sense of the state-of-the-art. "Proceedings of the 23rd International Conference on Computational Linguistics: Posters. Association for Computational Linguistics, 2010.
[20]. Broder, Andrei Z. "Identifying and filtering near-duplicate documents. "Annual Symposium on Combinatorial Pattern Matching. Springer Berlin Heidelberg, 2000.
[21]. E. Cohen et al. "Finding interesting associations without support pruning. "IEEE Transactions on Knowledge and Data Engineering, vol. 13, no. 1, pp. 64-78, 2001.
[22]. Leskovec, Jure, Anand Rajaraman, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2014.
[23]. Krestel, Ralf, Peter Fankhauser, and Wolfgang Nejdl. "Latent dirichlet allocation for tag recommendation. "Proceedings of the third ACM conference on Recommender systems. ACM, 2009.
[24]. Maskeri, Girish, Santonu Sarkar, and Kenneth Heafield. "Mining business topics in source code using latent dirichlet allocation. "Proceedings of the 1st India software engineering conference. ACM, 2008.
[25]. Apache Kafka. http://kafka.apache.org, 2015.
[26]. Gormley, Clinton, and Zachary Tong. Elasticsearch: The Definitive Guide. "O'Reilly Media, Inc.", 2015.
[27]. Apache Mesos. http://mesos.apache.org, 2015.
[28]. Apache Tika. https://tika.apache.org, 2015.
[29]. Confluent Inc. Kafka-Connect. http://docs.confluent.io, 2015.

Извлечение и анализ информации в современных предприятиях

А.Р. Топчян <a.topchyan@reply.de>
Ереванский государственный университет,
0025, Армения, г. Ереван, ул. А. Манукяна, дом 1

Аннотация. С ростом объема данных и потребности в них одной из основных проблем организаций становится обнаружение природы данных, выявление несомой ими информации и установление того, как и кем они используются. Объем данных и число разнородных систем, используемых для их обработки, растет, данные и системы все время усложняются, и совместное использование этих систем становится все более и более сложным. В этой работе мы описываем интеллектуальную поисковую систему, в основном предназначенную для решения проблемы поиска и обмена информацией в большом многопрофильной организации, в которой уже имеется много действующих систем для каждого отдела. Эта система является неотъемлемой частью совместной оперативной платформы данных (ODP) для исследования и обработки данных.

Ключевые слова: проекты, ориентированные на данные; извлечение информации, потоковая обработа; Mesos; Kafka

DOI: 10.15514/ISPRAS-2016-28(4)-1

Для цитирования: Топчян А.Р. Извлечение и анализ информации в современных предприятиях. Труды ИСП РАН, том 28, вып, 4, 2016, стр. 7-28 (на английском). DOI: 10.15514/ISPRAS-2016-28(4)-1

Список литературы

[1]. Topchyan A.R. Enabling Data Driven Projects for a Modern Enterprise. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 3, 2016, pp. 209-230. DOI: 10.15514/ISPRAS-2016-28(3)-13
[2]. Rahman, Nayem, and Fahad Aldhaban. "Assessing the effectiveness of big data initiatives."2015 Portland International Conference on Management of Engineering and Technology (PICMET). IEEE, 2015.
[3]. Davenport, Thomas H., and Jill Dych´e. "Big data in big companies."International Institute for Analytics (2013).
[4]. Dunning, Ted, and Ellen Friedman. Streaming Architecture: New Designs Using Apache Kafka and Mapr Streams. O'Reilly Media .2016.
[5]. Marz, Nathan, and James Warren. Big Data: Principles and best practices of scalable real-time data systems. Manning Publications Co, 2015
[6]. Michael Hausenblas and Nathan Bijnens. Lambda Architecture. http://lambda-architecture.net, 2015.
[7]. K. Mani Chandy. vent-Driven Applications: Costs, Benefits and Design Approaches, California Institute of Technology, 2006.
[8]. Akidau, Tyler, et al. "MillWheel: fault-tolerant stream processing at internet scale."Proceedings of the VLDB Endowment 6.11:1033-1044, 2013.

[9]. Zaharia, Matei, et al. "Discretized streams: Fault-tolerant streaming computation at scale."Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013.
[10]. Akidau, Tyler, et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, outof-order data processing."Proceedings of the VLDB Endowment 8.12: 1792-1803, 2015.
[11]. Verma, Abhishek, et al. "Large-scale cluster management at Google with Borg."Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015.
[12]. Boritz, J. "IS Practitioners' Views on Core Concepts of Information Integrity". International Journal of Accounting Information Systems. Elsevier, 2011.
[13]. Netflix. Distributed Resource Scheduling with Apache Mesos. http://techblog.netflix.com/2016/07/distributedresource-scheduling-with.html
[14]. Newell, Andrew, et al. "Optimizing distributed actor systems for dynamic interactive services.". Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016
[15]. Cohen, William, Pradeep Ravikumar, and Stephen Fienberg. "A comparison of string metrics for matching names and records.". Kdd workshop on data cleaning and object consolidation. Vol. 3, 2003
[16]. Hoffman, Matthew, Francis R. Bach, and David M. Blei. "Online learning for latent dirichlet allocation.". Advances in neural information processing systems, 2010
[17]. Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation. "Journal of machine Learning research 3.Jan: 993-1022, 2003
[18]. Mihalcea, Rada, and Paul Tarau. "TextRank: Bringing order into texts. "Association for Computational Linguistics, 2004.
[19]. Hasan, Kazi Saidul, and Vincent Ng. "Conundrums in unsupervised key phrase extraction: making sense of the state-of-the-art. "Proceedings of the 23rd International Conference on Computational Linguistics: Posters. Association for Computational Linguistics, 2010.
[20]. Broder, Andrei Z. "Identifying and filtering near-duplicate documents. "Annual Symposium on Combinatorial Pattern Matching. Springer Berlin Heidelberg, 2000.
[21]. E. Cohen et al. "Finding interesting associations without support pruning. "IEEE Transactions on Knowledge and Data Engineering, vol. 13, no. 1, pp. 64-78, 2001.
[22]. Leskovec, Jure, Anand Rajaraman, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2014.
[23]. Krestel, Ralf, Peter Fankhauser, and Wolfgang Nejdl. "Latent dirichlet allocation for tag recommendation. "Proceedings of the third ACM conference on Recommender systems. ACM, 2009.
[24]. Maskeri, Girish, Santonu Sarkar, and Kenneth Heafield. "Mining business topics in source code using latent dirichlet allocation. "Proceedings of the 1st India software engineering conference. ACM, 2008.
[25]. Apache Kafka. http://kafka.apache.org, 2015.
[26]. Gormley, Clinton, and Zachary Tong. Elasticsearch: The Definitive Guide. "O'Reilly Media, Inc.", 2015.
[27]. Apache Mesos. http://mesos.apache.org, 2015.
[28]. Apache Tika. https://tika.apache.org, 2015.
[29]. Confluent Inc. Kafka-Connect. http://docs.confluent.io, 2015.

Scalable Sandbox Environments for a Modern Organization

Artyom Topchyan <a.topchyan@reply.de>
Yerevan State University,
Alek Manukyan 1, Yerevan, 0025, Armenia

Abstract. With the growing volume and demand for data, a major concern for an Organization is the creation of collaborative Data-Driven projects. With the amount of data, number of departments and the development of potential use cases, the complexity of creating Multitenant and Collaborative environments for multidisciplinary teams to work and create productive solutions, is becoming more and more important problem. In this work, we describe an approach to building such an environment with scalability, flexibility and productivity in mind. This solution is an integral part of a joint Operational Data Platform for data exploration and processing at large data driven Organizations.

Keywords: data-driven projects, distributed, multidisciplinary, sandbox, auto-scaling, collaboration, fault-tolerance, isolation, containers, mesos, microservices

DOI: 10.15514/ISPRAS-2016-28(4)-2

For citation: Topchyan A.R. Scalable Sandbox Environments for a Modern Organization. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 4, 2016, pp. 29-40. DOI: 10.15514/ISPRAS-2016-28(4)-2

1. Introduction

With the growing volume and demand for data a major concern for an Organization is to find a well-defined way of enabling large teams to work collaboratively on a large amount of very varied data. This trend is driven by business and technical demand, which especially stems from the need for more Data-Driven projects [1][3][4]. Data-Driven projects aim at increasing the quality, speed and/or quantity of information gained from Data collected by the Organization. But it is very challenging to move ahead with such projects without the need to define models to handle data exploration and processing. As we described in our previous work [1][2], this leads to most of the project time being spent not on data analysis, but finding out information about the existing data, getting access to a usable form of the data and requesting a suitable ICT environment to process this data. This can be quite costly time and resource wise, and each stage of a Data-Driven project is impacted by this.

There is currently no single accepted approach for tackling such problems, so we described and implemented a joint Operational Data Platform(ODP) for data exploration and processing. In [1] we described the building blocks of the platform and went into detail, how the data is collected and stored. In [2]. we described the Information Marketplace, which is an intelligent search engine system, specifically designed to tackle the problem of information retrieval and sharing in a large multifaceted organization, that already has many systems in place for each Department. This platform has been successfully implemented and is actively used by large

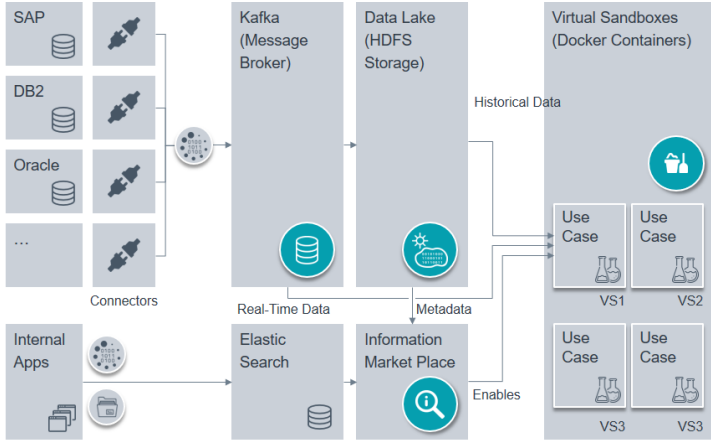


Fig. 1. Operational Data Platform

organizations to implement Data-Driven projects. A high level overview of the entire solution as presented in our previous work is outlined in Fig.1. In this work we will outline the third component of the Platform, which centers on providing a scalable environment to allow for collaborative and multi-developer projects. This approach aims to simplify testing and deployment in order to bring these highly complex projects into production. We will first outline why we believe such a solution is required. Next we will define a set of requirements, such a solution must fulfill. We will then outline the architecture, that aims to address these requirements, as well as how it differs from commonly accepted approaches and the various technical challenges involved in building such a system.

2. Data-Driven project development environment

As highlighted in [1] most Data-Driven projects in the industry often follow a variation of the Cross Industry Standard Process for Data Mining project lifecycle, as described in [5]. Which states that most of these project go through the following steps:

- Business Understanding;

- Data Understanding;
- Data Preparation;
- Modeling;
- Evaluation;
- Deployment.

In [1] we outlined how we address the necessity for enabling and optimizing these individual steps and in [2] we give more detail how to take step one and two and greatly simplify step 3 and 4. But we did not fully consider yet, is in what what context this project is developed. Projects are developed in environments, which first of all usually reflects internal requirements of the project. In our case project vary greatly in their complexity and the amount of data they process. So a environment is required that can generalize their parameters to the main functional requirements of the problem. A very important point to note in this context is that most Organizations require for these project to be developed in a a fully on-premise secure environment, which greatly limits what can be done towards building a scalable solution. This means we need to design the environment to fit the largest possible problem, that would need to be solved, which entails processing terabyters of data from many disparate systems. To this end one practical solution used in the industry is building clusters of many machines, that can handle these types of workloads [7] [8] In a modern Organization this usually means using the Hadoop ecosystem [6] to create a generalized processing environment with support for most data types and approach to analysis. This usually goes hand in hand with a Data Lake implementation [9] or alternatively a more flexible and structured approach as described in [1] Hadoop clusters are immensely powerful and flexible, but suffer from a number of oversights. Hadoop environment usually do not provide a very flexible way for dozens of users to access the systems in parallel. The usual approach for this is to provide an Edge Server [8], which is a secured server, that allows for selective access to certain resources of the cluster. There are usually a set number of such server per cluster and are not necessarily very powerful servers. The idea behind such servers is, that they should only be used to deploy job to the cluster and not carry out any calculation themselves. This is more geared towards Data-Engineers deploying applications and not an analysis environment, where Data Scientists have to explore the data and retrieve results in an ad-hoc manner. For this analysis many routine libraries and tools are often required and this leads to the Edge Servers running out of resources and contain a large number of potentially clashing software as these are shared environments. As more and more users start using these servers this becomes an impractical approach. To address this there are approaches to use vitalization to provide on demand environments for each individual team [11] [10] Due to vitalization overhead multiple teams need to share these machines, which are often over-provisioned and have much more resource than it is required as reconfiguring such machines is usually a lengthy administrative task. Requiring multiple teams to share a virtual environment is also the simplest way to enable collaboration as

individual developers can access the code, results or services of other team members. But this leads to the resource of the machines not being fully utilized and decreased performance when the project requires a large number of services or processing workloads, that do not run on the cluster. For example, prototyping a system similar to the Information Marketplace to analyse and index the documents and data of the Organization, would require, a Search engine, a Database and many other services. Another downside to such approaches is the complexity it brings to testing and deploying such projects, once development has finished. It is unclear what libraries and resources are required to actually run the application outside the development environment. Dependencies, configurations and resource specifications have to be supplied to the operations team, which has to somehow package this into a solution that conforms to the organizations standards and can be run in production. Such Edge Servers are not self-describing and fully rely on the development team to deliver well documented, testable, deployable code. This is often not simple to do due to the multidisciplinary and complex nature of Data-Driven projects. What is required is a self-describing environment, that is simple to ship into production. It is possible to directly deploy the vitalized edge-nodes with all the required software inside them, but this is quite expensive and slow, when working in an on premise environment as such approached are too rigid in their requirements. To summarize let's pose a list of requirements we accumulated based on our experience working with teams of Data Scientists at large organizations:

1. Single-User Secure isolated environments. It is beneficial for users to have fully isolated environments as this simplifies development and later on deployment. Because of this environment need to be fully isolated and secured in accordance with the Organizations requirements.

2. Support a large variety of demanding workloads. Such environment should support all types of workloads, that can be submitted to the Hadoop cluster as well as ran locally inside the environment itself, which is essential for ad-hoc analysis. Such environments should also support being used as servers for tools required by other projects. For example, a standalone Database deployment.

3. Managing Resources in a highly multi-tenant environment. We need flexibility to manage many of such instances efficiently, potentially on very limited hardware. Environments should consist of services based on required resource allocation

4. Collaboration tools.The environment should have in-built tools that enable collaboration. This is very important to enable well documented, testable projects and code.

5. Scalability and Fault-Tolerance. It should be straightforward to scale individual environments up and down on demand, in perspective

automatized. As it is important to ensure uninterrupted work and no loss of work data, such environment should be Fault-Tolerant.

6. Going to Test, Production quickly after development. The environments should be self-descriptive and flexible enough to speed up this process and not introduce more technical hurdles.

7. Flexible access to all stored Data. Because this is an environment for a Data-Driven project, the most important thing is access to all required data, but this access has to be customizable as to provide fun grained access for teams and individual developers.

This is not an exhaustive list, but it outlines the basic requirements such a system must fulfill from the view point of teams working on Data-Driven projects. It is list of complex functional requirements which Project, System Developers and Designers need to map to a technical problem definition and implementation. To this end we propose the analytical sandbox environments, which are generated, isolated environments provided to Data Scientists, Analysts and Engineers so that they can build up their project on the data. It is a fully isolated environment, where the user can install or download any extra tools they require and is accessible via an analytical and console view.

3. On-demand Sandbox environments

There are significant architectural and algorithmic considerations when mapping the requirement set outlined in the previous chapter to a technical implementation. We will outline the technical design of these sandboxes and the necessary components we use to provide the technical functionality.

3.1 Immutable environments

First of all we require access to all the existing cluster resources preferable via the native interfaces, but provide a remote accessible and simple to use environment. As one of the main gaols is to simplify development, this approach should also support some way to distribute source code and dependencies along with the environment. To summarize we require this environment to be stateless and immutable as possible, but still contain all the necessary tools and code when moved. To achieve the flexibility required we just replace it with another instance to make changes or ensure proper behavior. This would allow sandboxes to not only serve as an analytical environment, but as a template for all Data-Driven projects, that ever come to production. To this there has been a large shift in the Industry to use Linux Containerization technology [12] to solve these types of problems.

3.1.1 Linux Containers

LXC (Linux Containers) is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a control host using a single

Linux kernel [12] The Linux kernel provides the cgroups functionality, that allows limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) without the need for starting any virtual machines, and namespace isolation functionality that allows complete isolation of an applications' view of the operating environment, including process trees, networking, user IDs and mounted file systems [12] Such environments can be created based on a defined specification, which allows the environment to inherently be self-describing [12] This essentially would allow us to launch anything we want in completely independent environments. For example, on-demand edge nodes. This approach is very useful for creating immutable architectures, where every component can be replaced at any point in time. It is being used more and more in the Industry and is becoming the standard for running distributed services and applications [15] The use of this type of process management and assuming that each sandbox is fully isolated and immutable allows us to easily reason about Fault-Tolerance and Scalability.

3.2 Fault Tolerance and Scalability

In the context of Immutable containers, service fault tolerance and scaling becomes a process scheduling problem. Technically we have a set amount of resources in total. Each sandbox takes a certain amount of resources, such as CPU, RAM, Disk space and Network. Considering the fact, we need a large amount of resources, but limited amount of Hardware at our disposal, we need to effectively plan where each sandbox can run and how much resource it can actually use. This is a well-known problem in the Industry as is usually tackled by global Resource Manager or planners, such as Apache Mesos [13], Google Kubernetes [16] or Borg [14] and there are well defined patterns of tackling such problems [15] In this work we use Apache Mesos due to its popularity in the Industry and its proven ability to scale. Such a resource manager is essentially a higher level version of an Operating Systems Kernel running on many machines. Applications decide what resources they require based on their processing time and requirement, and the resource manager tries to accommodate this. If the service crashes or the computing node fails, it can be transparently restarted somewhere else assuming the application can transparently handle something like this. This would allow us to efficiently run many sandboxes on very limited hardware by automatically scaling how much resource each one requires as well as making each sandbox Fault-Tolerant by launching another one during outages.

3.3 Collaboration

A central aspect of a solution like this is simplifying collaboration. As we propose isolated environments, this becomes more challenging. The accepted approach using central source code-based collaboration, such as Git and SVN [21], which we also adopt. But it is often challenging to share large files, such as models, test data and dependencies between team members using these systems. To this end we implement a shared network file-system layer between team members of the same project. Each project team gets a filesystem, with individual workspaces for each members. Team

members can then decide how to structure this environment in order for it to be more productive for them. One crucial requirement for such system is Fault-Tolerance. Which is even more crucial due to our implantation of Immutable sandboxes. Any large files generated by the project team should be durable to sandbox and computing node outages. Loosing modelling results can lead to days of lost work time. To address this there are a number of high performance and durable implantation of Network Filesystems being used in the Industry [22] [23] In this work we adopted GlusterFS as our network storage due to its proven performance, simplicity and integration capabilities with systems like Mesos [23]

3.4 Isolation

As we outlined above isolating the sandboxes simplifies management and development. By isolation we are specifically referring to isolation of resources, such as:

- CPU
- RAM
- Filesystem
- Libraries and System tools

Most of these are out-of-the box supported by Linux container technology and are supported and managed by Apache Mesos. On the other hand, as these sandbox environments are not exclusive to Analytical Cases, which produce reports as their output, but Service based applications as well. An example of this would be a Web Service that provides recommendations based on customer input data. Such applications usually run as a web service and most commonly rely on some database system, which might not be provided as part of the Hadoop installation. Such services can be easily launched in a separate sandbox environment. We provide teams of developers with easy access to these services or databases, provided they are part of the same use case. As there might be any number of such services running inside the cluster, this might lead to problems as defined in [20] So in order to simplify the development process we would need full Network isolation as well. To solve this, it is now an accepted approach in the Industry to use software based overlay network solutions, which are computer networks that are built on top of another network [18] [19] This gives us the ability to selectively isolated certain Sandboxes from each other, allowing the users of these sandboxes to launch any service they want in their sandboxes and transparently make it accessible to other team members, without leading to network clashes with other environments. In this solution we use Calico networking, which is well integrated with Mesos environments.

3.5 Security

Due to the level of isolation provided Security becomes an Authorization problem and can be implemented as per requirements from the Organization. A commonly accepted approach in the Industry is to integrate with a central Organization wide

authority, which has all the information about user rights and permissions. In most cases we integrate directly with the Organizations central LDAP, such as Kerberos or Microsoft Active Directory [17] As most Hadoop installation also support this integration we can transparently enable security throughout the solution without introducing a new authorization and authentication concept in the Organization.

4. Architecture

Having discusses the individual building blocks of the solution, we will outline how this translates a technical architecture and implementation as a number of automation services are required to build the necessary platform based on the concepts outlined in the previous section. This architecture is implemented similarly to our previous work and must fulfill the requirements of low-latency and flexible services. To this end we adopted a Microsservice architecture, which a lightweight and flexible approach to implementing Service Oriented Architectures [24] The Sandbox orchestration service is defined with the following components.

4.1. Sandbox Templates

The first required component is a set of templates for sandbox environments. These vary based on the Organizational standards, but these typically contain all the necessary tools to interface with the Hadoop Cluster, run analytics and build applications. For example:

- Secure Shell access
- Ipython Console, R console, Scala Console
- Hive, Hue, Hadoop
- Pyspark, Spark, RSpark

These are common tools and having a repository of such as propose built templates allows to formalize the tool choice and testing and deployment processes during development. New templates can be added based on demand by extending the already existing ones.

4.2 Marathon

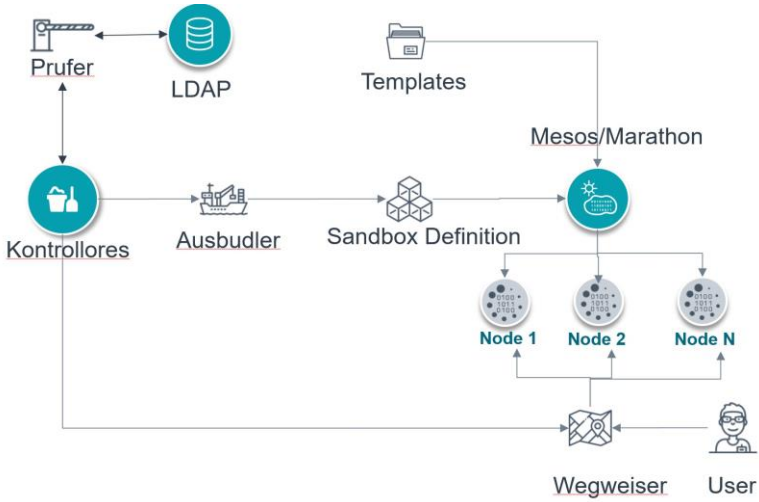
Marathon is a production-grade container orchestration platform for Apache Mesos [25] This allows for API based orchestration of containers based on the specific templates. We use this as our central orchestration platform for all sandboxes.

4.3 Kontrollores

Kontrollores is our central control service. It handles request for creating new teams and sandboxes for them. It provides the following functionality:

- Handle Sandbox creation requests;
- Authorize request;

- Create code repositories;
- Create collaboration file system;
- Initiate Sandbox creation;
- Report status.



Fig, 2. Sandbox microservice architecture

4.4 Ausbudller

Ausbudller Is an interface service to Marathon and handles the actual creation and monitoring of a single sandbox. It validates these request based on the user permissions and quotas stored in the central LDAP.

4.5 Prufer

Prufer handles validation of resource and tools requests based on the users in the team.

4.4 Wegweiser

Wegweiser is the central routing component. As the sandboxes are in isolated environments we require a secure gateway to dynamically route individual users through the internal network to their sandbox. This is achieved by discovering sandbox details based on their definition retrieved from Kontrollores.

The combined architecture defined with these components is outlined in Fig, 2, which also outlines the interconnection of these components and the general flow of execution.

5. Conclusion

We proposed an approach for a scalable, multi-user environment for handling research and development of Data-Driven projects in the context of a large organization operating a Hadoop cluster. We described the necessity for such a solution to handle the complexity of managing multi-user research projects and outlined the technical challenges faced when implementing similar systems. Based on this we outlined the set of requirements desired by many Organizations in the industry and we proposed a scalable, fault-tolerant and flexible Architecture as well as technical implementation, that satisfies all these requirements. We believe the approach, proposed solution and its architecture provide a solid basis for implementing similar systems at large Organizations, that are starting to explore Data-Driven projects.

References

[1]. Topchyan A.R. Enabling Data Driven Projects for a Modern Enterprise. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 3, 2016, pp. 209-230. DOI: 10.15514/ISPRAS-2016-28(3)-13

[2]. Topchyan A.R. Information Retrieval and Analysis for a Modern Organization. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016, pp. 7-28. DOI: 10.15514/ISPRAS-2016-28(4)-1

[3]. Rahman, Nayem, and Fahad Aldhaban. "Assessing the effectiveness of big data initiatives."2015 Portland International Conference on Management of Engineering and Technology (PICMET). IEEE, 2015.

[4]. Davenport, Thomas H., and Jill Dych´e. "Big data in big companies."International Institute for Analytics (2013).

[5]. Shearer C. The CRISP-DM model: the new blueprint for data mining. J Data Warehousing; 5:13—22, 2000

[6]. Tom White. Hadoop: The definitive guide. O'Reilly Media 2012

[7]. Rowstron, Antony, et al. "Nobody ever got fired for using Hadoop on a cluster. "Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing. ACM, 2012.

[8]. Lin, Chia-Feng, et al. "The study and methods for cloud based CDN."CyberEnabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on. IEEE, 2011.

[9]. Alex Gorelik. The Enterprise Big Data Lake: Delivering on the Promise of Hadoop and Data Science in the Enterprise. O'Reilly Media 2016

[10]. Shen, Zhiming, et al. "Cloudscale: elastic resource scaling for multi-tenant cloud systems. "Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011.

[11]. Li, Peng, Lee Toderick, and Joshua Noles. "Provisioning virtualized datacenters through virtual computing lab."2010 IEEE Frontiers in Education Conference (FIE). IEEE, 2010.

[12]. Rami Rosen. "Resource management: Linux kernel namespaces and cgroups"(PDF). cs.ucsb.edu. Retrieved February 11, 2015.

[13]. Apache Mesos. http://mesos.apache.org, 2015.

[14]. Verma, Abhishek, et al. "Large-scale cluster management at Google with Borg."Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015.

[15]. Burns, Brendan, and David Oppenheimer. "Design Patterns for Containerbased Distributed Systems."8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), 2016.
[16]. Burns, Brendan, et al. Borg, Omega, and Kubernetes."Communications of the ACM 59.5: 50-57, 2016.
[17]. Koutsonikola, Vassiliki, and Athena Vakali. "LDAP: framework, practices, and trends."IEEE Internet Computing 8.5: 66-72, 2004
[18]. Project Calico. https://www.projectcalico.org 2015.
[19]. Jain, Raj, and Subharthi Paul. "Network virtualization and software defined networking for cloud computing: a survey." IEEE Communications Magazine 51.11: 24-31, 2013
[20]. JAMES, BD. "Isolating network traffic in multi-tenant virtualization environments."U.S. Patent No. 7,885,276. 8 Feb, 2011.
[21]. Otte, Stefan. "Version Control Systems."Computer Systems and Telematics, 2009.
[22]. FraunhoferFS "FraunhoferFS High-Performance Parallel File System "Computer Systems and Telematics, 2009.
[23]. Klaver, Jeroen; van der Jagt, Roel. Distributed file system on the SURFnet network Report, University of Amsterdam System and Network Engineering, 2010
[24]. Newman, Sam. Building Microservices. "O'Reilly Media, Inc.", 2015.
[25]. Marathon. https://mesosphere.github.io/marathon, 2015.

Масштабируемые учебно-экспериментальные среды для современных предприятий

А.Р. Топчян <a.topchyan@reply.de>
Ереванский государственный университет,
0025, Армения, г. Ереван, ул. А. Манукяна, дом 1

Аннотация. С ростом объема данных и потребностей в них одной из основных проблем организаций становится создание кооперативных, управляемых данными проектов. При возрастании объема данных, числа подразделений и появлении новых потенциальных сценариев использования все более важным является создание корпоративных сред, поддерживающих совместную работу мультидисциплинарных групп для разработки эффективных решений. В этой работе мы описываем подход к построению такой среды, обладающей масштабируемостью, гибкостью и производительности. Это решение является неотъемлемой частью совместной оперативной платформы данных по изучению и обработке больших данных в крупных организацией, ориентированных на обработку данных.

Ключевые слова: проекты, ориентированные на данные; учебно-экспериментальные среды; автомасштабирование; кооперация; отказоустройчивость; изоляция; контейнеры; Mesos; микросервисы.

DOI: 10.15514/ISPRAS-2016-28(4)-2

Для цитирования: Топчян А.Р. Масштабируемые учебно-экспериментальные среды для современных предприятий. Труды ИСП РАН, том 28, вып, 4, 2016, стр. 29-40 (на английском). DOI: 10.15514/ISPRAS-2016-28(4)-2

Список литературы

[1]. Topchyan A.R. Enabling Data Driven Projects for a Modern Enterprise. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 3, 2016, pp. 209-230. DOI: 10.15514/ISPRAS-2016-28(3)-13

[2]. Topchyan A.R. Information Retrieval and Analysis for a Modern Organization. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016, pp. 7-28. DOI: 10.15514/ISPRAS-2016-28(4)-1

[3]. Rahman, Nayem, and Fahad Aldhaban. "Assessing the effectiveness of big data initiatives."2015 Portland International Conference on Management of Engineering and Technology (PICMET). IEEE, 2015.

[4]. Davenport, Thomas H., and Jill Dych´e. "Big data in big companies."International Institute for Analytics (2013).

[5]. Shearer C. The CRISP-DM model: the new blueprint for data mining. J Data Warehousing; 5:13—22, 2000

[6]. Tom White. Hadoop: The definitive guide. O'Reilly Media 2012

[7]. Rowstron, Antony, et al. "Nobody ever got fired for using Hadoop on a cluster. "Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing. ACM, 2012.

[8]. Lin, Chia-Feng, et al. "The study and methods for cloud based CDN."CyberEnabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on. IEEE, 2011.

[9]. Alex Gorelik. The Enterprise Big Data Lake: Delivering on the Promise of Hadoop and Data Science in the Enterprise. O'Reilly Media 2016

[10]. Shen, Zhiming, et al. "Cloudscale: elastic resource scaling for multi-tenant cloud systems. "Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011.

[11]. Li, Peng, Lee Toderick, and Joshua Noles. "Provisioning virtualized datacenters through virtual computing lab."2010 IEEE Frontiers in Education Conference (FIE). IEEE, 2010.

[12]. Rami Rosen. "Resource management: Linux kernel namespaces and cgroups"(PDF). cs.ucsb.edu. Retrieved February 11, 2015.

[13]. Apache Mesos. http://mesos.apache.org, 2015.

[14]. Verma, Abhishek, et al. "Large-scale cluster management at Google with Borg."Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015.

[15]. Burns, Brendan, and David Oppenheimer. "Design Patterns for Containerbased Distributed Systems."8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), 2016.

[16]. Burns, Brendan, et al. Borg, Omega, and Kubernetes."Communications of the ACM 59.5: 50-57, 2016.

[17]. Koutsonikola, Vassiliki, and Athena Vakali. "LDAP: framework, practices, and trends."IEEE Internet Computing 8.5: 66-72, 2004

[18]. Project Calico. https://www.projectcalico.org 2015.

[19]. Jain, Raj, and Subharthi Paul. "Network virtualization and software defined networking for cloud computing: a survey." IEEE Communications Magazine 51.11: 24-31, 2013

[20]. JAMES, BD. "Isolating network traffic in multi-tenant virtualization environments."U.S. Patent No. 7,885,276. 8 Feb, 2011.

[21]. Otte, Stefan. "Version Control Systems."Computer Systems and Telematics, 2009.

[22]. FraunhoferFS "FraunhoferFS High-Performance Parallel File System "Computer Systems and Telematics, 2009.

[23]. Klaver, Jeroen; van der Jagt, Roel. Distributed file system on the SURFnet network Report, University of Amsterdam System and Network Engineering, 2010

[24]. Newman, Sam. Building Microservices. "O'Reilly Media, Inc.", 2015.

[25]. Marathon. https://mesosphere.github.io/marathon, 2015.

A Model Checking-Based Method of Functional Test Generation for HDL Descriptions

M.S. Lebedev <lebedev@ispras.ru>
S.A. Smolov <smolov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Automated test generation is a promising direction in hardware verification research area. Functional test generation methods based on models are widespread at the moment. In this paper, a functional test generation method based on model checking is proposed and compared to existing solutions. Automated model extraction from the hardware design's source code is used. Supported HDLs include VHDL and Verilog. Several kinds of models are used at different steps of the test generation method: guarded action decision diagram (GADD), high-level decision diagram (HLDD) and extended finite-state machine (EFSM). The high-level decision diagram model (which is extracted from the GADD model) is used as a functional model. The extended finite-state machine model is used as a coverage model. The aim of test generation is to cover all the transitions of the extended finite state machine model. Such criterion leads to the high HDL source code coverage. Specifications based on transition and state constraints of the EFSM are extracted for this purpose. Later, the functional model and the specifications are automatically translated into the input format of the nuXmv model checking tool. nuXmv performs model checking and generates counterexamples. These counterexamples are translated to functional tests that can be simulated by the HDL simulator. The proposed method has been implemented as a part of the HDL Retrascope framework. Experiments show that the method can generate shorter tests than the FATE and RETGA methods providing the same or better source code coverage.

Keywords: hardware design; functional verification; static analysis; test generation; guarded action; high-level decision diagram; extended finite state machine; model checking.

DOI: 10.15514/ISPRAS-2016-28(4)-3

For citation: Lebedev M.S., Smolov S.A. A Model Checking-Based Method of Functional Test Generation for HDL Descriptions. *Trudy ISP RAN / Proc. ISP RAS]*, vol. 28, issue 4, 2016, pp. 41-56. DOI: 10.15514/ISPRAS-2016-28(4)-3

1. Introduction

Functional verification is an expensive and time-consuming stage of hardware design process [1]. Due to hardware designs increasing complexity, automated test generation seems to be important and challenging. To avoid design complexity, automated verification methods often utilize mathematical abstractions of system

properties and behavior, so-called *models*. Models can be created manually or automatically extracted from the system's source code. Automated verification methods based on model extraction from the HDL (Hardware Description Language – a collective name for several languages described below) source code are considered in this paper. Models can be based on the following formal descriptions: finite-state machines, decision diagrams, Petri nets [2], etc.

Model checking [3] is an approach to set up the correspondence between the model of the system and formal conditions (specifications). For every specification a model checker tries to produce a *counterexample* – an input stimuli sequence that leads the system into a specification-contradicting state. Counterexample construction is often used for functional test generation purposes.

Proof of equivalence of a model and the corresponding system is an important issue when model checking is used for hardware verification. There is no need in such proof when the model is automatically extracted from the system's source code and translated into the model checker's format.

A method of functional test generation for hardware is proposed in this paper. It is based on automatic extraction of High-Level Decision Diagrams (HLDD) [4] from the system's source code. Synthesizable sets of VHDL [5] and Verilog [6] HDLs are supported. Extracted models are then automatically translated into SMV (Symbolic Model Verifier) language supported by the nuXmv [7] model checker. Extended Finite State Machine (EFSM) transition constraints are used as specifications for model checking. EFSM model is also extracted from the system's source code. Counterexamples built by the nuXmv model checker are then translated into an HDL testbench which can be simulated by an HDL simulator.

The rest of the paper is organized as follows. Section 2 contains a review of functional test generation methods based on model extraction from the HDL source code. In Section 3 basic definitions are given. Section 4 contains HLDD construction and test generation methods. Section 5 is dedicated to the experimental results. Section 6 concludes the paper.

2. Related works

The idea of model extraction from the HDL source code and following test generation is not new. A prototype of CV tool for VHDL description model checking is presented in [8]. The tool's execution process consists of five stages. On the first stage, a VHDL description is parsed and an internal representation is constructed. A Binary Decision Diagram (BDD) based model is built on the second stage. On the third stage a CTL-based specification is parsed. The specification language syntax is described in the paper. On the fourth stage the specification parsing result and the BDD-based model are passed to the CBMC [9] model checker. On the final stage, the model checker output is translated into tests that can be executed by the HDL simulator. It is stated that BDD-based model size plays the key role in the model checking process. Model

size reduction heuristics usage is suggested to avoid state space explosion but no heuristics are offered in the paper.

In [10], extraction of the EFSM model and generation of tests that cover all the model transitions are described (so-called FATE method). This method assumes that the user provides additional information for the tool about signal semantics (for example, which of the signals encodes state). The EFSM extraction process contains several stages of transition structure simplification (embedded conditions elimination, compatible transitions union, dataflow dependency analysis). The test generation method is based on the state graph traversal through random walk and *backjumping* techniques.

In [11] an improved modification of method [10] is proposed. Optimizations described concern path reachability (*weakest precondition* [12] is used instead of the approximate approach) and test filtering tasks. A new functional test generation method called RETGA is also proposed in [11]. This method is based on the algorithm [13] for automated EFSM model extraction from HDL descriptions. The algorithm does not require additional information about signals\variables semantics; for state and clock-like variable detection it uses heuristics based on dataflow dependencies. Experiments have shown that RETGA method produces shorter tests with higher HDL code coverage than FATE and even improved FATE do.

It should be noted, that state graph traversal techniques (that FATE and RETGA methods use) do not guarantee coverage of all the EFSM model transitions. One of the problems concerns *counter* [11] variables that are defined in transition loops and used in transition guards, so an EFSM simulation engine needs to recognize at which value of the counter it is going to finish the loop execution.

3. Basic definitions

Suppose that all models described below run in discrete time that implies clock presence. Clock C is a set of events $\{c_1,\ldots,c_k\}$ where an event $c = \{signal, edge\}$ is a pair, consisting of a one-bit *signal* (so-called *clock pulse*) and a type of registration called *edge* (i.e. *positive edge* when *signal* changes its value from 0 to 1 and *negative edge* otherwise).

Let V be a set of *variables*. A *valuation* is a function that associates a variable $v \in V$ with a value $[v]$ from the corresponding domain. Let Dom_V be a set of all valuations of V. A *guard* is a Boolean function defined on valuations ($Dom_V \to \{true, false\}$). An *action* is a transform of valuations ($Dom_V \to Dom_V$). A pair $\gamma \to \delta$, where γ is a guard and δ is an action, is called a *guarded action* (GA). It is implied that there is a description of every function in some HDL-like language (thus, we can reason about not only semantics, but syntax).

Let guarded actions be *synchronized* [14], if each GA is associated with a clock. A system $\{\langle C^{(i)}, \gamma^{(i)} \to \delta^{(i)}\rangle\}_{i=1,l}$ of synchronized guarded actions can be represented by an oriented acyclic graph $G = (N, E, C)$ called *Guarded Actions Decision Diagram* (GADD). Here N is a set of graph nodes, E is a set of graph edges, and C is a clock.

N contains two non-intersecting subsets: a set N_s of non-terminal nodes n_s that are marked by expressions $\gamma(n_s)$; a set N_t of terminal nodes n_t that are marked by actions $\delta(n_t)$. Graph edges can start from non-terminal nodes only and finish either in terminal or in non-terminal nodes. Edges $e \in E$ are marked by sets $Val(e, n)$ of accepted values $\gamma(n)$ (here edge e is an outgoing edge for the node n, $e \in Out(n)$). The node $n \in N_s$ can have no more than one $e_d \in Out(n)$ that is marked by *default* keyword – it means that for this path in G an expression $\gamma(n)$ equals to a value that does not belong to any marking sets of the other edges outgoing from the node n. Supposing that the GADD contains exactly one root node n_{root} (the node without incoming edges, $In(n_{root}) = \varnothing$), a set of *paths* from the root node to all the terminal nodes produces a system of synchronized guarded actions. For example, the i^{th} path, including $n_1^{(i)},\ldots n_m^{(i)}$ nodes and $e_1^{(i)},\ldots, e_{m-1}^{(i)}$ edges ($n_1^{(i)} \equiv n_{root}$, $n_m^{(i)} \in N_t$, $e_k^{(i)} \in Out(n_u^{(i)}) \cap In(n_{u+1}^{(i)})$, $u = 1,\ldots,m-1$), defines a guarded action with $p_1^{(i)}\ldots p_{m-1}^{(i)}$ ($p_r^{(i)} = \mathrm{AND}(\gamma(n_r) == q)$, $r = 1,\ldots,m-1$, $q \in Val(e_r, n_r)$) conjunction as a guard and $\delta(n_m^{(i)})$ as an action. The guarded action clock is a subset of the GADD clock.

In [4] a definition of a high-level decision diagram (HLDD) is given and is shown that every variable of an HDL description can be represented by a function $v = f(v_1,\ldots,v_n) = f(V)$ in terms of HLDD H_v. Let $Z(v)$ be a finite set of all possible values of a variable $v \in V$. A *High-Level Decision Diagram* for v is an oriented acyclic graph $H_v = (M, \Gamma, V)$ where M is a set of nodes, and Γ is a mapping $M \to 2^M$. Let $\Gamma(m)$ be a set of *subsequent* nodes of the node $m \in M$ and $\Gamma^{-1}(m)$ be a set of *preceding* nodes of the node m. A node m_0 of the graph H_v is said to be *initial* if the set of its preceding nodes is empty: $\Gamma^{-1}(m_0) = \varnothing$. M consists of two non-intersecting subsets: M_n for non-terminal nodes and M_t for terminal nodes. All the non-terminal nodes $m_c \in M_n$ are marked by variables $v(m_c) \in V$ and meet the following condition: $2 \le |\Gamma(m_c)| \le |Z(v(m_c))|$. This means that m_c has at least two subsequent nodes but not more than the number of possible values of $v(m_c)$. All the terminal nodes $m_k \in M_t$ are marked by functions $v(m_k) = f_k(V_k)$, $f_k(V_k) \in F$ ($V_k \subseteq V$). Usually some of these functions are trivial and equal either to variables $v_k \in V$ or to constants. All the edges are marked by sets of accepted values of variables in the same manner as in the GADD definition; the semantics of the default edges is also similar.

On every tick of the clock, the HLDD H_v assigns a value to the target variable v through an *activation* procedure. Starting from the initial node m_0 it calculates values of the variables which mark non-terminal nodes. For a value e of the variable $v(m_c)$, $e \in Z(v(m_c))$, the corresponding edge from the node $m_c \in M$ to the subsequent node $m^e \in \Gamma(m_c)$ is activated. A vector V^t of variable values activates the path $l(m_0, m_k)$ from m_0 to the terminal node m_k marked by the function $f_k(V_k)$ that determines the value of the target variable v.

4. HLDD model construction and test generation method

The proposed test generation method consists of the following steps:

- HDL (VHDL/Verilog) description parsing and GADD model construction.
- HLDD model construction using the GADD model.
- HLDD model and specification translation into the nuXmv model checker input language (SMV model) [16].
- SMV model checking by the nuXmv model checker and translation of counterexamples into HDL tests.

The first step has been implemented in [13] so we will start from the second step. Note that all the actions, which mark the terminal nodes of the GADD model, are represented in the *static single assignment* (SSA) [15] form.

4.1. HLDD model construction

GADD and HLDD models preserve the module structure of the original HDL description. Every HDL description process is represented by a single GADD. The GADD $G = (N, E, C)$ is used as a basis for HLDD construction for every non-input variable of the process. HLDD construction algorithm pseudo code is listed below:

```
proto = new;
for node ∈ N do
  hldd_node = transform_node(node);
  proto.add(hldd_node);
end
copy_edges(E, proto);
for (v : non_input_variables(G)) do
  hldd = proto.keep_assigns(v);
  hldd.add_missing_terminals();
  hldd.transform_identical_assigns();
end
```

At the first step the HLDD prototype *proto* is created. GADD nodes are transformed into HLDD nodes with the help of the *transform_node* method and added to the prototype. Terminal GADD nodes $n_t \in N_t$ are transformed into terminal HLDD nodes $m_k \in M_t$. Every terminal node n_t marked by multiple assignment action $\delta(n_t)$ is transformed into a sequence of nodes. Every node in this sequence is marked by a corresponding single assignment a_k. Every terminal HLDD node is marked by a target variable v_k (which is the left-hand side of a_k) and a function $f_k(V_k)$ (which is the right-hand side of a_k).

Non-terminal GADD nodes $n_s \in N_s$ are transformed into non-terminal HLDD nodes $m_c \in M_n$. Guard γ which marks the node n_s is replaced by a new variable $guard(m_c)$ which marks the node m_c. The new HLDD that contains a single terminal node marked by γ is created for this variable (*create_variable_from_switch* method). GADD edges are transformed into HLDD edges by the *copy_edges* method. The corresponding values are not changed.

Then for every non-input variable *v* the HLDD *hldd* is created which is actually a modified copy of *proto*. The *keep_assigns* method removes from M_t the terminal nodes which are not marked by *v*. After that the *add_missing_terminals* method adds new terminal nodes marked by $f(v) = v$ to the edges which lack the subsequent terminal nodes. This means that the value of *v* does not change if any path to such node is activated. The *transform_identical_assigns* method searches for such non-terminal nodes m_c whose reachable terminal nodes are marked by the same function $f_k(v_k)$, and replaces m_c and its reachable subgraph with the only terminal node marked by $f_k(v_k)$.

Consider an example of the HLDD model construction for a simple VHDL description. This description contains a single module and a single process. The module interface consists of input variables *clk*, *rst*, *x*, *y* and an output variable *res* (all of 1-bit size). The process contains two internal variables: a 1-bit size vector *cnt* and an integer *state* (that can be assigned either 0 or 1). The source code of the process is listed below:

```
process (clk, rst, x, y)
  variable cnt: std_logic;
  variable state: integer range 0 to 1;
begin
if (rst = '1') then
  cnt := '0';
  state := 0;
elsif (clk = '1') then
  if (state = 1) then
    cnt := x or y;
    state := 0;
  elsif (state = 0) then
    cnt := x and y;
    state := 1;
  end if;
  res <= cnt;
end if;
end process;
```

0 shows the GADD model of the process. Non-terminal nodes of the GADD are shown as diamonds and correspond to branch expressions. Terminal nodes are shown as rectangles and correspond to basic blocks. Outgoing edges from the non-terminal nodes are marked by possible values of the branch expressions. Note that the *default* edge on 0 is unreachable because the *state* variable can only take the value of 0 or 1. The clock of the GADD is formed by events of *clk*, *rst*, *x* and *y* signals.

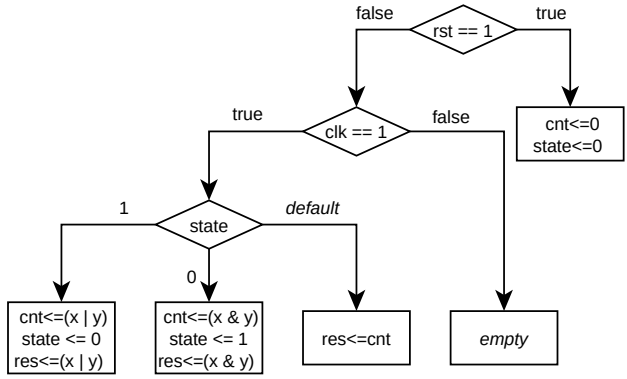


Fig. 1. GADD model.

0 shows the HLDD prototype. Expressions, which mark the non-terminal nodes, are replaced by *guard0*, *guard1*, *guard2* variables.

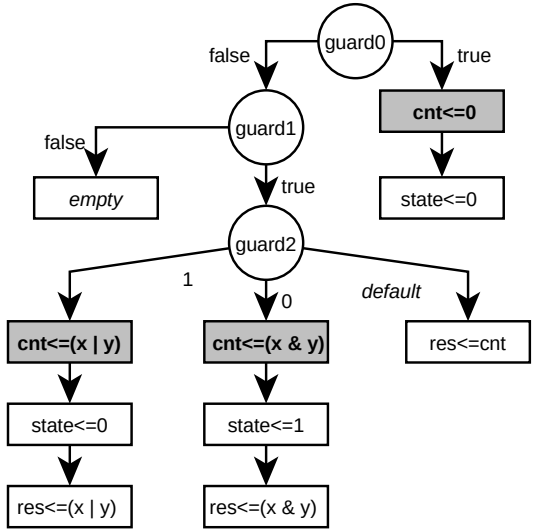


Fig.2. HLDD prototype.

Consider the HLDD construction for the *cnt* variable. Terminal nodes marked by *cnt* are highlighted in grey on 0. Terminal nodes, which are not marked by this variable, are removed. New terminal nodes marked by *cnt* are added to the free non-terminal

node edges (this means that the value of *cnt* does not change on these paths). The final HLDD is represented on 0. Similar diagrams are constructed for the other non-input variables of the HDL description (in our example those are: *state* and *res*).

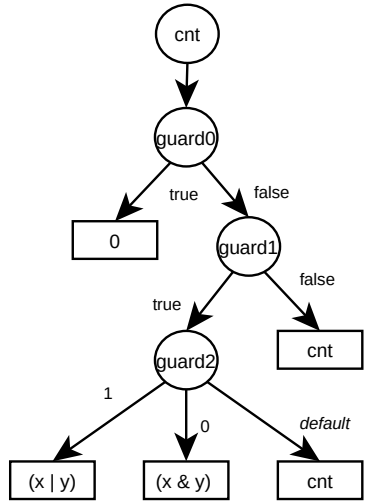


Fig. 3. HLDD of a cnt variable.

4.2. SMV model construction and checking

The constructed HLDD model is translated into an SMV language description. Hardware design module structure is preserved. Any variable constraints (like the range of possible values that is specified for the *state* variable) and their initial values described in the HDL description are added to the SMV model.

Specification construction is based on the EFSM model extracted from the same HDL description. Formal definition of the EFSM model and its extraction algorithm from an HDL description are presented in [13]. Here we provide only the informal definition. Extended finite-state machine is a special case of an ordinary finite-state machine (FSM). It contains sets of inputs, outputs and internal variables. EFSM transitions are marked by guard expressions, which depend on input and internal variable values, and by actions that can change internal and output variable values. A transition can be enabled only if its guard becomes *true*. When a transition is enabled, its action is executed. Specifications used by the proposed method are represented as negations of the EFSM transition guards. Negation is used to make the model checker build a counterexample – a sequence of data states and input stimuli that contradicts the specification (and thus satisfies the corresponding guard).

The nuXmv model checker checks the SMV model along with the specifications. Output counterexamples are translated into a test set aimed at covering reachable EFSM transitions.

Below you can see the HLDD-to-SMV translation result for the *cnt* and *guard0* variables. NuXmv-compatible SMV language format is used. The description consists of the variable declaration section (VAR) and the assignment section (ASSIGN). The *init* construct defines the initial value of a variable. The *next* construct defines the value of a variable in the next model state. The assignment (":=") defines the value of a variable in the current model state. Numeric values in the example are of bit vector type and are represented by "0<type><size>_<value>" construct.

```
VAR
  cnt : word[1];
  guard0 : boolean;
…
ASSIGN
  init(cnt) := 0d1_0;
…
ASSIGN
  next(cnt) :=
    case
      (guard0 = TRUE) : 0d1_0;
      (guard0 = FALSE) :
        case
          (guard1 = TRUE) :
            case
              (guard2 = 0sd32_1) : (x | y);
              (guard2 = 0sd32_0) : (x & y);
              TRUE : cnt;
            esac;
          (guard1 = FALSE) : cnt;
        esac;
    esac;
…
guard0 := (rst = 0d1_1);
guard1 := (clk = 0d1_1);
guard2 := state
```

The example of an SMV specification is listed next:

LTLSPEC ! **F** ((state = 0sd32_0) & (clk = 0d1_1) & !(rst = 0d1_1));

EFSM transition reachability condition consists of the *state* variable constraint (which determines the source state of the transition) and the guard condition depending on the *clk* and *rst* variables.

The nuXmv model checker generates the following counterexample for this specification:

Trace Type: Counterexample
-> **State: 1.1** <-
SAMPLE.process.state = 0sd32_0
SAMPLE.process.cnt = 0ud1_0
SAMPLE.process.guard2 = 0sd32_0
SAMPLE.process.guard1 = FALSE
SAMPLE.process.guard0 = FALSE
SAMPLE.process.res = 0ud1_0
clk = 0ud1_0
y = 0ud1_0
x = 0ud1_0
rst = 0ud1_0
-> State: 1.2 <-
SAMPLE.process.guard1 = TRUE
clk = 0ud1_1

The first state shows the initial values assigned to the variables. The second state shows only the values that have changed. We can see that the second state contradicts the given SMV specification: clk is equal to 1, while the rst and state variables are equal to 0.

5. Experimental results

The proposed test generation method was implemented as a part of the HDL Retrascope 0.2.1 software tool [17]. Java language was used for development along with the Fortress formulae manipulation library [18]. Some HDL descriptions from the ITC'99 benchmark [19] were used for testing of the proposed approach.

The nuXmv model checker supports both symbolic model checking and bounded model checking [21] methods. In some cases, symbolic model checking needed too much time and computer resources because of the state explosion (for example, B04, B10 and B11 designs). Bounded model checking could manage this problem by exploring the model state space only up to some bound. However, bound value affects the model checking results (not all the counterexamples may be obtained at the specified bound). Therefore, in some cases the bound size was iteratively increased in order to get all possible counterexamples.

Generated tests were simulated by the QuestaSim HDL simulator [20]. Test properties (length and source code coverage) were compared to existing test generation methods like FATE [10], RETGA [11] (these methods are based on EFSM model extraction from the HDL descriptions and are targeted at covering the EFSM model transitions) and random test generation.

0contains information about the ITC'99 designs that were used for test generation: their source code size and the corresponding SMV model size (without specifications). Size is given in lines of code.

Table 1. HDL description and SMV model size

Design	HDL	SMV
B01	102	207
B02	70	143
B03	134	637
B04	101	809
B06	127	442
B07	92	370
B08	88	315
B09	100	263
B10	167	755
B11	118	368

0contains the test length information. Test length is given in clock cycles. The length of tests generated by the random generation method corresponds to the point when the test coverage growth stops (maximum length was chosen as 1000000 clock cycles). The sign "-" means that the corresponding method failed to generate tests for the corresponding HDL design.

Table 2. Test length

Design	FATE	RETGA	SMV	Random
B01	115	49	69	300
B02	62	33	47	80
B03	-	-	504	2000
B04	104	36	67	200
B06	198	76	88	700
B07	246	166	249	1000
B08	31	52	31	1000000
B09	19	231	84	1000000
B10	173	135	134	650000
B11	101	721	194	1000000

In 5 of 10 cases tests generated by the proposed method are shorter than tests generated by the FATE method and longer that RETGA tests. Either the rest tests are of comparable length with the leader (RETGA), or tests generated by the FATE method provide lower coverage. Definitive conclusion about the advantages or disadvantages of the proposed method in comparison with the RETGA method cannot be made using the selected HDL description set.

Notice that unlike the FATE and RETGA methods the proposed method is not based on EFSM traversal. So it was able to generate the test for B03 design in contrast to those methods (EFSM extracted from this design is too complex for traversal).

0shows the HDL source code statement coverage in comparison to the FATE, RETGA and random generation methods.

Table 3. Source code statement coverage

Design	FATE	RETGA	SMV	Random
B01	97,14%	100%	100%	100%
B02	100%	100%	100%	100%
B03	-	-	100%	100%
B04	100%	100%	100%	100%
B06	100%	100%	100%	100%
B07	93,93%	93,93%	93,93%	84,85%
B08	81,81%	100%	100%	90,91%
B09	35,29%	100%	100%	61,77%
B10	95,94%	100%	100%	97,29%
B11	69,23%	94,87%	94,87%	87,18%

0shows the HDL source code branch coverage in comparison to the FATE, RETGA and random generation methods.

Table 4. Source code branch coverage

Design	FATE	RETGA	SMV	Random
B01	96,15%	100%	100%	100%
B02	100%	100%	100%	100%
B03	-	-	100%	100%
B04	100%	100%	100%	100%
B06	100%	100%	100%	100%
B07	94,73%	94,73%	94,73%	73,69%
B08	76,92%	100%	100%	84,62%
B09	35,71%	100%	100%	57,15%
B10	90,47%	100%	100%	97,61%
B11	71,87%	96,87%	96,87%	90,63%

The proposed method achieved the same code coverage as the RETGA method at the specified set of HDL descriptions. B07 and B11 HDL description coverage is less than 100% because of the unreachable code in these designs.

6. Conclusion and future work

The functional test generation method based on automated HLDD model extraction and checking with nuXmv is presented in this paper. The main advantage of this method is its flexibility in choosing a test target (through using different kinds of specifications). EFSM transition coverage is presented for comparison to the other test generation methods (FATE, RETGA). Any other specifications can be formulated and checked in order to get a test aimed at covering the corresponding property of a model. The presented implementation of the proposed approach does not produce shorter tests than existing approaches on the chosen hardware design set. Simple optimizations (like test filtering) can be helpful and are going to be implemented in the nearest future.

Future work is focused on applying the method to more complex hardware designs (including Verilog-based). In this case complexity is defined by the number of execution paths in processes and the number of processes and modules in an HDL description. Process decomposition using dataflow analysis methods and predicate abstraction [22] test generation methods are under research now.

Acknowledgment

Authors would like to thank Russian Foundation for Basic Research (RFBR). The reported study was supported by RFBR, the research project number is 15-07-03834.

References

[1]. J. Bergeron. Writing Testbenches: Functional Verification of HDL Models. Springer, 2003. 478 p.
[2]. V.G. Lazarev, E.I. Piil'. Control automata synthesis. Energoatomizdat, Moscow, 1989. 328 p. (in Russian).
[3]. E.M. Clarke, O. Grumberg, and D.A. Peled. Model Checking. MIT Press, Cambridge, 2000. 314 p.
[4]. R.J. Ubar, J. Raik, A. Jutman, M. Jenihhin. Diagnostic modeling of digital systems with multi-level decision diagrams. Design and Test Technology for Dependable Systems-on-Chip, 2011. pp. 92-118.
[5]. IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002), 2009. pp.c1-626.
[6]. IEEE Standard for Verilog Hardware Description Language. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), 2006. pp.0_1-560.
[7]. D. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta. The nuXmv symbolic model checker. Proceedings of the 16th International Conference on Computer Aided Verification (CAV), 2014, № 8559. pp. 334-342.
[8]. D. Deharbe, S. Shankar, E.M. Clarke. Model checking VHDL with CV. Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD), 1998. pp. 508-514.
[9]. CBMC model checker. Available at: http://www.cprover.org/cbmc/
[10]. G. Guglielmo, L. Guglielmo, F. Fummi, G. Pravadelli. Efficient generation of stimuli for functional verification by backjumping across extended FSMs. Journal of Electronic Functional Testing: Theory and Application, 2011, № 27(2). pp. 137-162.
[11]. I. Melnichenko, A. Kamkin, S. Smolov. An extended finite state machine-based approach to code coverage-directed test generation for hardware designs. Trudy ISP RAN / Proc. ISP RAS, 2015, vol. 27, issue 3,. pp. 161-182. DOI: 10.15514/ISPRAS-2015-27(3)-12.
[12]. E. Dijkstra. A Discipline of Programming. Prentice Hall, 1976. 217 p.
[13]. S. Smolov, A. Kamkin. A method of extended finite state machines construction from HDL descriptions based on static analysis of source code. Nauchno-tehnicheskie vedomosti Sankt-Peterburgskogo gosudarstvennogo politehnicheskogo universiteta. Informatika. Telekommunikacii. Upravlenie. [St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunication and Control Systems], № 1(212), 2015. pp. 60-73 (in Russian).

[14]. J. Brandt, M. Gemünde, K. Schneider, S. Shukla, J.-P. Talpin. Integrating system descriptions by clocked guarded actions. Forum on Design Languages, 2011. pp. 1-8.
[15]. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, № 13(4), 1991. pp. 451-490.
[16]. M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta. NuXmv 1.0 User Manual. 2014. pp. 7-44. Available at: https://es-static.fbk.eu/tools/nuxmv/index.php?n=Documentation.Home.
[17]. HDL Retrascope toolkit. Available at: http://forge.ispras.ru/projects/retrascope.
[18]. Fortress library. Available at: http://forge.ispras.ru/projects/solver-api.
[19]. ITC'99 benchmark. Available at: http://www.cad.polito.it/tools/itc99.html.
[20]. QuestaSim simulator. Available at: https://www.mentor.com/products/fv/questa/.
[21]. E. Clarke, A. Biere, R. Raimi, Y. Zhu. Bounded model checking using satisfiability solving. Formal Methods in System Design, 2001, vol. 19, issue 1, pp. 7-34.
[22]. E. Clarke, M. Talupur, H. Veith, D. Wang. SAT based predicate abstraction for hardware verification. Lecture Notes in Computer Science, 2004, vol. 2919, pp. 78-92.

Генерация функциональных тестов для HDL-описаний на основе проверки моделей

М.С. Лебедев<lebedev@ispras.ru>
С.А. Смолов <smolov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Разработка методов автоматической генерации тестов составляет перспективное направление в области верификации цифровой аппаратуры. На текущий момент большое распространение имеют методы генерации функциональных тестов на основе моделей. В данной работе представлен метод генерации функциональных тестов на основе проверки моделей и результаты его сравнения с существующими решениями. В методе используется автоматическое извлечение моделей из исходного кода HDL-описания аппаратуры. Поддерживаются языки VHDL и Verilog. Метод генерации тестов включает автоматическое построение моделей следующих типов: решающие диаграммы системы охраняемых действий (Guarded Action Decision Diagram,GADD), высокоуровневые решающие диаграммы (High-Level Decision Diagrams, HLDD) и расширенные конечные автоматы (Extended Finite-State Machines, EFSM). HLDD-модель используется в качестве функциональной модели. Модель EFSM используется в качестве модели покрытия. Целью тестирования является покрытие всех переходов расширенного конечного автомата. Выбор такого критерия позволяет получить высокое покрытие исходного кода HDL-описания. Из EFSM-модели извлекаются спецификации, основанные на ограничениях на переходы и состояния. Затем спецификации и функциональная модель автоматически транслируются во входной формат инструмента

проверки моделей nuXmv. Инструмент выполняет проверку модели и строит контрпримеры. Контрпримеры транслируются в функциональные тесты, которые могут быть исполнены с помощью HDL-симулятора. Предлагаемый метод был реализован программно в инструменте HDL Rertrascope. Результаты экспериментов показывают, что метод генерирует более короткие тесты, чем методы FATE и RETGA, при обеспечении такого же или лучшего покрытия исходного кода.

Ключевые слова: цифровая аппаратура; функциональная верификация; статический анализ; генерация тестов; охраняемое действие; высокоуровневая решающая диаграмма; расширенный конечный автомат; проверка модели.

DOI: 10.15514/ISPRAS-2016-28(4)-3

Для цитирования: Лебедев М.С., Смолов С.А. Генерация функциональных тестов для HDL-описаний на основе проверки моделей. Труды ИСП РАН, том 28, вып. 4, 2016 г. стр. 41-56 (на английском). DOI: 10.15514/ISPRAS-2016-28(4)-3

Список литературы

[1]. J. Bergeron. Writing Testbenches: Functional Verification of HDL Models. Springer, 2003. 478 p.

[2]. Лазарев В.Г., Пийль Е.И. Синтез управляющих автоматов. Энергоатомиздат, Москва, 1989. 328 с.

[3]. E.M. Clarke, O. Grumberg, D.A. Peled. Model Checking. MIT Press, Cambridge, 2000. 314 p.

[4]. R.J. Ubar, J. Raik, A. Jutman, M. Jenihhin. Diagnostic modeling of digital systems with multi-level decision diagrams. Design and Test Technology for Dependable Systems-on-Chip, 2011. pp. 92-118.

[5]. IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002), 2009. pp.c1-626.

[6]. IEEE Standard for Verilog Hardware Description Language. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), 2006. pp.0_1-560.

[7]. D. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta. The nuXmv symbolic model checker. Proceedings of the 16th International Conference on Computer Aided Verification (CAV), 2014, № 8559. pp. 334-342.

[8]. D. Deharbe, S. Shankar, E.M. Clarke. Model checking VHDL with CV. Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD), 1998. pp. 508-514.

[9]. CBMC model checker. Доступно по ссылке: http://www.cprover.org/cbmc/

[10]. G. Guglielmo, L. Guglielmo, F. Fummi, G. Pravadelli. Efficient generation of stimuli for functional verification by backjumping across extended FSMs. Journal of Electronic Functional Testing: Theory and Application, 2011, № 27(2). pp. 137-162.

[11]. I. Melnichenko, A. Kamkin, S. Smolov. An extended finite state machine-based approach to code coverage-directed test generation for hardware designs. Proceedings of the Institute for System Programming, 2015, № 27(3). pp. 161-182. DOI: 10.15514/ISPRAS-2015-27(3)-12.

[12]. E. Dijkstra. A Discipline of Programming. Prentice Hall, 1976. 217 p.
[13]. С. А. Смолов, А. С. Камкин. Метод построения расширенных конечных автоматов по HDL-описанию на основе статического анализа кода. Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление, 2015, № 1(212), 60–73.
[14]. J. Brandt, M. Gemünde, K. Schneider, S. Shukla, J.-P. Talpin. Integrating system descriptions by clocked guarded actions. Forum on Design Languages, 2011. pp. 1-8.
[15]. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, № 13(4), 1991. pp. 451-490.
[16]. M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta. NuXmv 1.0 User Manual. 2014. pp. 7-44. Доступно по ссылке: https://es-static.fbk.eu/tools/nuxmv/index.php?n=Documentation.Home.
[17]. Fortress library. Доступно по ссылке: http://forge.ispras.ru/projects/solver-api.
[18]. ITC’99 benchmark. Доступно по ссылке: http://www.cad.polito.it/tools/itc99.html.
[19]. QuestaSim simulator. Доступно по ссылке: https://www.mentor.com/products/fv/questa/.
[20]. E. Clarke, A. Biere, R. Raimi, Y. Zhu. Bounded model checking using satisfiability solving. Formal Methods in System Design, 2001, Vol. 19 Iss. 1. pp. 7-34.
[21]. E. Clarke, M. Talupur, H. Veith, D. Wang. SAT based predicate abstraction for hardware verification. Lecture Notes in Computer Science, 2004, Vol. 2919. pp. 78-92.

Checking Parameterized PROMELA Models of Cache Coherence Protocols

[1] *V.S. Burenkov <burenkov_v@mcst.ru>*
[2] *A.S. Kamkin <kamkin@ispras.ru>*
[1] *JSC MCST,*
24 Vavilov str., Moscow, 119334, Russian Federation
[2] *Institute for System Programming of the Russian Academy of Sciences,*
25 Alexander Solzhenitsyn str., Moscow, 109004, Russian Federation

Abstract. This paper introduces a method for scalable verification of cache coherence protocols described in the PROMELA language. Scalability means that resources spent on verification (first of all, machine time and memory) do not depend on the number of processors in the system under verification. The method is comprised of three main steps. First, a PROMELA model written for a certain configuration of the system is generalized to the model being parameterized with the number of processors. To do it, some assumptions on the protocol are used as well as simple induction rules. Second, the parameterized model is abstracted from the number of processors. It is done by syntactical transformations of the model assignments, expressions, and communication actions. Finally, the abstract model is verified with the SPIN model checker in a usual way. The method description is accompanied by the proof of its correctness. It is stated that the suggested abstraction is conservative in a sense that every invariant (a property that is true in all reachable states) of the abstract model is an invariant of the original model (invariant properties are the properties of interest during verification of cache coherence protocols). The method has been automated by a tool prototype that, given a PROMELA model, parses the code, builds the abstract syntax tree, transforms it according to the rules, and maps it back to PROMELA. The tool (and the method in general) has been successfully applied to verification of the MOSI protocols implemented in the Elbrus computer systems.

Keywords: multicore microprocessors, shared memory multiprocessors, cache coherence protocols, model checking, SPIN, PROMELA.

DOI: 10.15514/ISPRAS-2016-28(4)-4

For citation: Burenkov V.S, Kamkin A.S. Checking Parameterized PROMELA Models of Cache Coherence Protocols. *Trudy ISP RAN / Proc. ISP RAS]*, volume 28, issue 4, 2016. pp. 57-76. DOI: 10.15514/ISPRAS-2016-28(4)-4

1. Introduction

Shared memory multiprocessors (*SMP*) constitute one of the most common classes of high-performance computer systems. In particular, it includes multicore

microprocessors, which combine several processors (cores) on a single chip [1]. Nowadays, 8- and 16-core microprocessors are in mass production; hardware vendors have announced forthcoming 48-, 80-, and even 100-core designs. Multicore microprocessors and SMP systems are also designed by Russian companies such as MCST and INEUM, e.g., Elbrus-4C (4 cores, 2014) and Elbrus-8C (8 cores, 2015) [2].

The main problem arising in the development of SMP systems is ensuring *memory coherency*. As each processor contains a local cache, multiple copies of the same data may exist in the system: one copy is in the main memory, and several copies are in the processors' caches. Modification of a copy should cause either the invalidation of the other copies or their consistent modification. This is supported by so-called *cache controllers*, i.e. memory devices connected into a network and cooperating in accordance with a special protocol, so-called *cache coherence protocol* (*CCP*) [3].

Development of cache coherence mechanisms includes two stages: first, design of a CCP; second, its implementation in hardware. The both stages are error-prone; accordingly, methods for protocol verification and methods for hardware verification are in use [4]. Protocol bugs are especially critical and should be revealed before implementing the hardware. The widely recognized method for protocol verification is *model checking* [5]. It is fully automated, but suffers from a principal drawback – it is not scalable due to the *state space explosion* problem. Using the traditional methods for verifying a CCP of a system with four and more processors is impossible (at least, highly problematic) [6].

To overcome the issue and develop scalable verification technologies, researchers utilize *parameterized model checking* [7]. The idea is to construct abstract models that are independent of the number of processors and may be verified with the existing tools. Correctness of the abstract model guarantees correctness of the original one (checking, however, may produce wrong error messages, so-called *false positives*). The proposed approach is also of that type. In contrast to the existing ones, it supports the PROMELA language used in the SPIN model checker [8] and the message passing primitives. The method was successfully used for verifying the CCPs implemented in the Elbrus computer systems [2].

The paper is structured as follows. In Section 2, we analyze the existing approaches to CCP verification. In Section 3, we propose a method for constructing an abstract model out of a PROMELA protocol model. In Section 4, we describe theoretical foundations of the suggested method. In Section 5, we provide a case study on using the method for verifying a MOSI protocol. In Section 6, we summarize our work and outline directions of further research.

2. Related work

As it has been said, classical model checking is inapplicable to CCPs with an arbitrary number of processors. There exists an alternative approach, called *deductive verification*; however, it is hardly automated due to the need of so-called *inductive invariants* [9] and does not provide any diagnostic information if there are errors.

Parameterized model checking seems to be a more promising approach. It is worth mentioning two directions.

First, verification of a parameterized model (in essence, a family of models) can be reduced to the verification of a single model of the family. Corresponding methods are aimed at finding such number N that verification of the model for N components (processors, cache controllers, etc.) is sufficient for proving correctness in general. In [7], such kind of method is presented, and it is reported that $N = 7$ is enough for the protocols having been examined. However, that value is too big to make the method applicable to industrial SMP systems [6].

Second, a model (parameterized model) can be abstracted so as to reduce the state space size (make it independent of the number of components). In [10], a method for abstracting a model from the exact number of *replicated identical components* (e.g., caches in which the cache line is in a given state) is introduced. The technique significantly reduces the state space size; however, the use of a modified version of the Murφ tool complicates its real-life application. A similar idea, called $(0,1,\infty)$-*counter abstraction*, is employed in [11]-[13]. Though the technique seems to be powerful, it often leads to overly detailed abstract models, which makes the approach inapplicable to complex protocols.

In [14], a general method for *compositional verification* is proposed. The idea is to replace a subset of identical components with an abstract one, called *environment*. Such replacement usually leads to false positives, and considerable efforts are required to eliminate them. In [15]-[18], the approach has been adapted to CCPs. The suggested method is based on *syntactical transformations* of Murφ models and *counterexample-guided abstraction refinement* (*CEGAR*). The main drawbacks are as follows:

- Murφ does not support the message passing primitives, which complicates CCP description;
- restrictions on Murφ models of CCPs are not clearly defined;
- the tools are not in open access.

3. Suggested method

The problem to be solved is as follows. Given a PROMELA model of a CCP for some configuration of an SMP system (i.e. a model with a fixed number $n > 2$ of processors), it is required to check the CCP correctness for an arbitrary configuration of the system (i.e. for any $N \geq n$).

Models considered in this paper satisfy the following conditions (obtained from the verification practice and shown to be sufficient for specifying CCPs). The allowed statements are **if**, **do**, **goto**, = (*assignment*), ! (*send*), and ? (*receive*). Each guarded action is placed in an **atomic** block and therefore is executed with no interruption; **else** alternatives are absent. Assignments' right-hand sides contain only primary expressions, i.e. variables and constants; left-hand sides are variables and array elements (an array index is a primary expression). Atomic logic formulae are of the

form $x == c$ or $B(ch)$, where x is a variable (or an array element), c is a constant, ch is a channel, and B is a predicate: **empty**, **full**, etc.

3.1 Model parameterization

From the conceptual point of view, a CCP model consists of an unbounded number of replicated identical processes, so-called *basic processes*, and a fixed number of *auxiliary processes*. Without loss of generality we will assume that there is only one auxiliary process. All processes are enumerated from 0 to N, where N is a parameter: 0 is the identifier of the auxiliary process, while $1, \dots, N$ are the identifiers of the basic processes. All arrays used in the model (arrays of variables and arrays of channels) are of length N and indexed with the identifiers of the basic processes.

To generalize the original model to a parameterized one, the following induction rules are used:

- each condition containing an array is either a conjunction or a disjunction of similar conditions on all array elements:
  - $\varphi\{i/1\} \wedge \dots \wedge \varphi\{i/n\}$ is interpreted as $\forall i \in \{1, \dots, N\}: \varphi$;
  - $\varphi\{i/1\} \vee \dots \vee \varphi\{i/n\}$ is interpreted as $\exists i \in \{1, \dots, N\}: \varphi$;
- each sequence of statements $\alpha\{i/1\}; \dots; \alpha\{i/n\}$ is interpreted as a loop **for** $(i: 1 \,..\, N)$ $\{\alpha\}$.

Here, φ (α) is a formula (statement) containing an index i as a free variable, and $\varphi\{i/t\}$ ($\alpha\{i/t\}$) denotes the result of substitution of t for all occurrences of i in φ (α).

3.2 Assumptions

Let us consider a CCP where request processing is coordinated by a *system commutator* of the *home processor* (the processor that owns the requested data). Accordingly, the PROMELA model contains two process types: $proc$ is a cache controller (a basic process), and $home$ is a home processor's commutator (an auxiliary process). As usual, the CCP model deals with a single cache line.

Broadly speaking, the CCP works as follows. Each $proc$ instance may initiate an operation on the cache line by sending a primary request to the $home$ process. Upon its reception and analysis, $home$ sends *snoop requests* to all processes except for the sender. After snoop reception, a $proc$ sends a response to the sender (data or an acknowledgement that it has completed an action on the cache line). Having collected all of the answers, the sender informs $home$ on the completion of the operation. As soon as the completion message is received, $home$ can accept the next primary request (see Fig. 1).

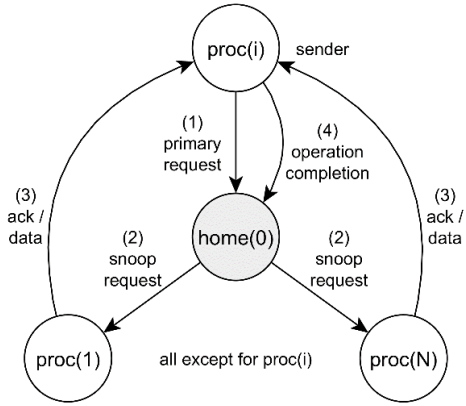


Fig. 1. Generalized scheme of a CCP

It is worth emphasizing that at most one primary request is being processed at each moment of time. It is assumed that values of global variables (e.g., a current sender identifier) are set by $home$ upon reception of a primary request and do not change during its processing.

Each channel can be read by a single process; however, multiple processes are allowed to write into it. A channel is called *simple* if there is only one sender; otherwise, it is called *multiplexed*. Let $C_{S\to r}$ be the set of channels with the reader r and senders from the set S. Channels are divided into three groups (hereinafter, singletons are written without brackets, e.g., $0 \to j$ stands for $\{0\} \to j$):

- $C_* = \bigcup_{j=0}^{N} C_{\{1,\dots,N\}\to j}$ is the set of multiplexed channels of capacity N used by $home$ and $proc$ to receive messages from the basic processes (e.g., a channel over which $home$ receives primary requests, and channels over which processes receive responses);
- $C_{h\to p} = \bigcup_{j=1}^{N} C_{0\to j}$ is the set of simple channels of positive capacity (which is defined by the CCP, but independent of N) used by the basic processes to receive messages from $home$ (e.g., channels over which $home$ transmits snoop requests);
- $C_{p\to h} = \bigcup_{i=1}^{N} C_{i\to 0}$ is the set of simple channels of capacity 1 used by $home$ to receive messages from the basic processes (e.g., channels over which a sender informs $home$ on operation completion).

Messages transmitted via channels are ordered pairs of the form (opc, i), where opc is an operation code, and i is an identifier of the message sender.

A verified CCP property looks as follows:

$$\mathbf{G}\{\forall k, l \in \{1, \dots N\}: (k \neq l) \to \varphi\{i/k, j/l\}\},$$

where **G** is an operator that requires its argument to be true in all reachable states of the model [5]; φ is a formula with two free indices (i and j) that characterizes cache coherency in the corresponding caches. For MOSI protocols [3], φ is as follows:

$$\begin{cases}\neg(cache[i] = M \wedge cache[j] \neq I);\\ \neg(cache[i] = O \wedge cache[j] = O);\end{cases}$$

where $cache$ is an array that stores the cache line states.

3.3 Informal description

The core of the proposed method is syntactical transformation of PROMELA code. The transformations change the process types and retain four processes of $N + 1$: a modified $home$ process ($home_{abs}$), two modified $proc$ processes ($proc_{abs}$), and an environment process representing the rest of the processes ($proc_{env}$). Accordingly, the initialization process of the abstract model is as follows (ABS is a constant distinct from 0, 1, and 2):

```
init {
 atomic {
  run home_abs(0);
  run proc_abs(1);
  run proc_abs(2);
  run proc_env(ABS);
 }
}
```

The length of all arrays is changed from N to 2 (recall that arrays are indexed with the identifiers of the $proc$ processes). Each array access is supplied with the guard $i \leq 2$, where i is the index of the element being accessed.

- On read (in a condition), the atomic formula containing the array access, is replaced with $undef$ (an undefined value) if the index is rejected by the guard:

  $$B(x[i], \dots) \Longrightarrow (i \leq 2 \rightarrow B(x[i], \dots) : undef).$$

  In PROMELA, a formula of the kind $(B \rightarrow t_1 : t_2)$ corresponds to the conditional construct **if** B **then** t_1 **else** t_2 **fi**.

- On write (in an assignment), the assignment to the array is placed inside the selection statement:

  $$x[i] = t \Longrightarrow \textbf{if} :: \textbf{atomic}\,\{i \leq 2 \rightarrow x[i] = t\} :: \textbf{else fi}.$$

  Assignments to the global variables as well as conditions on the global variables remain unchanged.

Channels of the set $C_{h \rightarrow p}$ are represented as an array (let us denote it as ch). Similarly to other arrays, it is truncated to length 2. Each atomic formula over $ch[i]$, where $i > 2$, is replaced with $undef$, while each operation on such a channel is removed. Channels of the sets C_* and $C_{p \rightarrow h}$ are represented by individual variables, not arrays.

Send statements are either unchanged or removed. A statement $ch!m$ in a process type P is removed only in the following cases:

- $ch \in C_{h\to e}$ and $P = home_{abs}$, where $C_{h\to e} = \bigcup_{j=3}^{N} C_{0\to j}$;
  e.g., $home_{abs}$ does not send snoop requests to $proc_{env}$;
- $ch \in C_*$ и $P = proc_{env}$;
  e.g., $proc_{env}$ does not send primary requests / snoop responses.

Receive statements may be left unchanged, modified, or removed. A statement $ch?m$ in a process type P is removed only in the following case:

- $ch \in C_{h\to e}$ and $P = proc_{env}$;
  e.g., $proc_{env}$ does not receive snoop requests.

Modification of $ch?m$ takes place solely in the following case:

- $ch \in C_*$ and $P \in \{home_{abs}, proc_{abs}\}$.

The corresponding transformation replaces a guarded action of the kind **atomic** $\{B \to ch?m\}$ with the following selection statement:

if
:: **atomic** $\{B' \to ch?m\}$
:: **atomic** $\{m.opc = opc_1;\ m.i = ABS\}$
...
:: **atomic** $\{m.opc = opc_k;\ m.i = ABS\}$
fi

where B' is the result of B transformation, and $opc_1, \ldots, opc_k$ are all possible operation codes that may be sent along the channel ch.

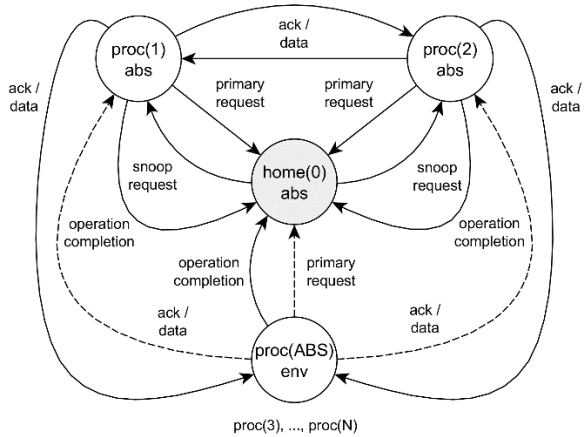


Fig. 2. Abstraction of a CCP model

Fig. 2 provides a simplified view on CCP model abstraction. All processes except for $home(0)$, $proc(1)$, and $proc(2)$ are merged into the environment process

$proc_{env}(ABS)$. Solid arrows represent the unmodified send / receive statements. Dashed arrows correspond to the removed sends / modified receives.

Having performed the above transformations, all logical formulae containing $undef$ (in essence, formulae of Kleene's strong three-valued logic) are transformed into classic logic formulae such that $undef$ in the outer scope is interpreted as $true$. This is achieved by the obvious transformation F:

- $F(\varphi) \Longrightarrow G(\varphi, true)$;
- $G(undef, T) \Longrightarrow T$;
- $G(B, T) \Longrightarrow B$, where B is an atom distinct from $undef$;
- $G(\neg\varphi, T) \Longrightarrow \neg G(\varphi, \neg T)$;
- $G(\varphi \circ \psi, T) \Longrightarrow G(\varphi, T) \circ G(\psi, T)$, where $\circ \in \{\wedge, \vee\}$.

When transforming the PROMELA model, the following optimizations are applied:

- constant propagation and folding;
- dead code elimination.

Here are some simple examples:

- $(i \leq 2) \Longrightarrow true$ in $home_{abs}$ and $proc_{abs}$;
- $(true \wedge B) \Longrightarrow B$ and $(false \wedge B) \Longrightarrow false$;
- **atomic** $\{true \rightarrow \alpha\} \Longrightarrow \alpha$.

It should be said that in general case the abstraction procedure transforms $N + 1$ processes to the $k + 2$ ones, where $k \in \{2, \ldots N - 1\}$: $proc_{abs}$ (in the number k), $home_{abs}$, and $proc_{env}$.

4. Theoretical foundations

4.1 Basic definitions

Let Var be a *set of variables* and $Chan$ be a *set of channels*. $Data = Var \cup Chan$ is referred to as the *set of data*. For each $c \in Chan$, a value $|c| > 0$, called *capacity*, is defined. A *data state* (or *state* for short) is a *valuation* of data, i.e. a mapping s that maps each variable v to the *value* $s(v) \in \mathbb{N}$ and each channel c to the sequence of *messages* $s(c) \in \mathbb{M}^*$ such that $|s(c)| \leq |c|$. The set of all states is denoted by S. A designated state $s_0 \in S$ is called *initial*.

Let us assume that there is a language over the data that includes logic formulae and statements, such as $x = t$ (*assignment*), $c\,!\,m$ (*send*), and $c\,?\,m$ (*read*).

A *guard* is a formula; an *action* is a sequence of statements; a *guarded action* is a pair $\gamma \rightarrow \alpha$, where γ is a guard, and α is an action. The guarded action $true \rightarrow \epsilon$, where ϵ is the empty sequence of statements, is called *empty* and designated as ε. The set of all guarded actions is denoted by Act. A guarded action $\gamma \rightarrow \alpha$ is called *executable* in $s \in S$ iff (if and only if) $s \vDash \gamma$.

A *process graph* (or *process* for short) is a triple $\langle V, v_0, E \rangle$, where V is a *set of vertices*, $v_0 \in V$ is an *initial vertex*, and $E \subseteq V \times Act \times V$ is a *set of edges*.

Process structure is defined by the control statements: **if** (*selection*), **do** (*repetition*), and **goto** (*jump*). Correspondence between code and processes is straightforward and not described here.

A *system* is a set of processes, i.e. $\{\langle V_i, v_{0_i}, E_i\rangle\}_{i=0}^{N}$. Hereinafter, P_i is considered to be a shortcut for $\langle V_i, v_{0_i}, E_i\rangle$. A *configuration* of $\{P_i\}_{i=0}^{N}$ is a pair $\langle l, s\rangle$, where $l: \{0, \dots, N\} \to \bigcup_{i=0}^{N} V_i$ such that $l(i) \in V_i$ for all $i \in \{0, \dots, N\}$, so-called the *control state*, and $s \in S$. The configuration $\langle l_0, s_0\rangle$, where $l_0(i) = v_{0_i}$ for all $i \in \{0, \dots, N\}$, is called *initial*.

The *state space* of a system $\{P_i\}_{i=0}^{N}$ is a triple $\langle C, c_0, T\rangle$, where C is the set of all configurations of the system, c_0 is the initial configuration, and $T \subseteq C \times \left(\{0, \dots, N\} \times (\bigcup_{i=0}^{N} E_i)\right) \times C$ is a *transition relation* such that the following property holds: $\left(\langle l, s\rangle, \left(i, (v, \gamma \to \alpha, v')\right), \langle l', s'\rangle\right) \in T$ iff:

- $l(i) = v$;
- $(v, \gamma \to \alpha, v') \in E_i$;
- $s \vDash \gamma$;
- $l' = (l \setminus \{i \mapsto v\}) \cup \{i \mapsto v'\}$;
- $s' = [\![\alpha]\!](s)$, where $[\![\alpha]\!]: S \to S$ is the semantics of α (actions are assumed to be deterministic).

It is worth mentioning that the restrictions on the transition relation conform to the notion of *asynchronous parallelism*.

A configuration c is called *reachable* in a state space $\langle C, T, c_0\rangle$ iff there is a path in T from c_0 to c. A state s is called *reachable* iff a configuration $\langle l, s\rangle$, for some l, is reachable.

4.2 System abstraction

A *process transformation* (or *transformation* for short) is a function that maps one process to another.

Let $Data_S = (Var_S \cup Chan_S) \subseteq Data$ be a *set of significant data*. States s and s' are called *equivalent* (it is designated as $s \sim s'$) iff $s_{|Data_S} = s'_{|Data_S}$.

A guarded action $\gamma' \to \alpha'$ is referred to as an *abstraction* of a guarded action $\gamma \to \alpha$ in $s \in S$ iff:

- the truth of γ' is determined only by the significant data: for all $s' \in S$ such that $s' \sim s$, $s' \vDash \gamma'$ iff $s \vDash \gamma'$;
- the effect of α' is determined only by the significant data: for all $s' \in S$ such that $s' \sim s$, there holds $[\![\alpha']\!](s') \sim [\![\alpha']\!](s)$;
- γ' is weaker than γ: $s \vDash \gamma \to \gamma'$;
- α' acts similar to α: $[\![\alpha']\!](s) \sim [\![\alpha]\!](s)$.

A set of guarded actions $\{\gamma_i' \to \alpha_i'\}_{i=1}^m$ is referred to as an *abstraction* of a guarded action $\gamma \to \alpha$ in $s \in S$ iff there exists $i \in \{1, \dots, m\}$ such that $\gamma_i' \to \alpha_i'$ is an abstraction of $\gamma \to \alpha$ in s.

A guarded action $\gamma' \to \alpha'$ (a set $\{\gamma_i' \to \alpha_i'\}_{i=1}^m$) is referred to as an *abstraction* of $\gamma \to \alpha$ iff $\gamma' \to \alpha'$ ($\{\gamma_i' \to \alpha_i'\}_{i=1}^m$) is an abstraction of $\gamma \to \alpha$ in all states.

An *abstraction function* is a mapping $f: Act \to 2^{Act}$ such that for all $\gamma \to \alpha \in Act$, $f(\gamma \to \alpha)$ is an abstraction of $\gamma \to \alpha$. The abstraction function $I(\gamma \to \alpha) \equiv \{\gamma \to \alpha\}$ is called *trivial*.

It should be emphasized that this view to abstraction is a bit simplified. An abstraction function should take into account context of a guarded action (the process edge, the process, and the model). Thus, it is assumed that each guarded action contains the context information.

Let $P = \langle V, v_0, E \rangle$ be a process, f be an abstraction function, V' be some set, and $R: V \to V'$ be a mapping. An *abstraction* of P induced by f and R is the process $f(P, R) = \langle V', R(v_0), E' \rangle$, where E' is defined as follows:

- if $(v, \gamma \to \alpha, u) \in E$ and $f(\gamma \to \alpha) = \{\gamma_i' \to \alpha_i'\}_{i=1}^m$, then $\{(R(v), \gamma_i' \to \alpha_i', R(u))\}_{i=1}^m \subseteq E'$;
- no other edges belong to E'.

An abstraction $f(P, R)$, where R is a bijection, is referred to as a *bijective abstraction*.

Besides transforming individual processes, there are of interest transformations that merges several processes into one. Let us consider a particular kind of such transformations, where processes to be merged are identical.

Given a system $\{P_i\}_{i=0}^N$, the following denotations can be introduced ($i \in \{0, \dots, N\}$):

- Use_i is the set of variables read by P_i;
- Def_i is the set of variables assigned by P_i;
- $Var_i = Use_i \cup Def_i$ is the set of variables of P_i;
- Var_{L_i} is the set of *local variables* of P_i (we do not define the set Var_{L_i} assuming that it is provided);
- $Var_G = Var \setminus (\cup_{i=0}^N Var_{L_i})$ is the set of *global variables*.

Similarly, the following sets of channels (including the *sets of local channels* and the *set of global channels*) can be defined: In_i, Out_i, $Chan_i$, $Chan_{L_i}$, and $Chan_G$. In addition,

- $Data_i = Var_i \cup Chan_i$ is the set of data of P_i;
- $Data_{L_i} = Var_{L_i} \cup Chan_{L_i}$ is the set of *local data* of P_i;
- $Data_G = Var_G \cup Chan_G$ is the set of *global data*.

Processes are called *identical* if they can be transformed one another by renaming their local data. More formally, processes P_i and P_j are called identical if there are a bijection $R: V_i \to V_j$ and a bijection $r: Data_{L_i} \to Data_{L_j}$ such that $R(v_{0_i}) = v_{0_j}$ and

$(v, \gamma \to \alpha, u) \in E_i$ iff $\big(R(v), r(\gamma \to \alpha), R(u)\big) \in E_j$, where $r(\gamma \to \alpha)$ is the result of renaming the local data in $\gamma \to \alpha$ in accordance with r.

Let $\{P_i\}_{i=k_1}^{k_2}$ be a system of identical processes, $Data_S \cap \big(\bigcup_{i=k_1}^{k_2} Data_{L_i}\big) = \emptyset$ (the processes' local data are insignificant), g be an abstraction function, V' be some set, and $R: V_{k_1} \to V'$ be a mapping. The process $g\big(P_{k_1}, \dots, P_{k_2}; R\big) = g\big(P_{k_1}, R\big)$ is called a *unifying abstraction* of $\{P_i\}_{i=1}^{k}$ induced by g and R.

The definition needs to be clarified. Provided that the processes $\{P_i\}_{i=k_1}^{k_2}$ operate simultaneously, there are control states that cannot be represented by a single vertex of the abstraction $g\big(P_{k_1}, \dots, P_{k_2}; R\big)$. Thus, a unifying abstraction may appear to be inadequate. Let us assume that each process can be either *active* or *passive*, and it is prohibited two or more processes to be active simultaneously. Besides, the passive mode is organized as the following loop:

- a request is received;
- the local data are updated;
- a response is sent;
- the control is returned to the initial vertex.

Let $V(E')$ be the set of all vertices of the edges from E'.

A process $P = \langle V, v_0, E_A \cup E_P \rangle$ is referred to as a *bimodal process* with the set of *active edges* E_A and the set of *passive edges* E_P iff $E_A \cap E_P = \emptyset$ and the graph $\langle V(E_P), E_P \rangle$ is strongly connected.

Given a bimodal process $P = \langle V, v_0, E_A \cup E_P \rangle$, the following denotation can be introduced: $V_A = V(E_A)$ and $V_P = V(E_P)$ (generally speaking, $V_A \cap V_P \neq \emptyset$).

The process $g(P, R) = \langle V', v_0', E' \rangle$, where g is an abstraction function, and $R: V \to V'$ is a mapping, is called a *serializing abstraction* of P iff R satisfies the following properties:

- $R(v) = v_0'$ for all $v \in V_P \setminus V_A$;
- $R: V_A \to V'$ is a bijection;

and E' is defined as follows:

- if $(v, \gamma \to \alpha, u) \in E_A$ and $g(\gamma \to \alpha) = \{\gamma_i' \to \alpha_i'\}_{i=1}^{m}$, then $\{(R(v), \gamma_i' \to \alpha_i', R(u))\}_{i=1}^{m} \subseteq E'$;
- $(v_0', \varepsilon, v_0') \in E'$ (so-called *ε-self loop*);
- no other edges belong to E';

and for every $(v, \gamma \to \alpha, u) \in E_P$, the empty guarded action ε is an abstraction of $\gamma \to \alpha$, i.e. α depends on and affects solely insignificant data.

The nature of serializing abstraction is removing all passive edges and replacing them with the ε-self loop $(v_0', \varepsilon, v_0')$. Being applied to identical bimodal processes, such abstraction makes them unimodal and serializable (at most one process is operating, i.e. being in a non-initial state, at each moment of time) and allows constructing an adequate unifying abstraction.

Let $M = \{P_i\}_{i=0}^{N}$ be a system where all processes, except maybe $\{P_i\}_{i=0}^{k}$, for some $k \in \{0, \dots, N\}$, are identical and bimodal; $Data_S$ be significant data; V_i', where $i \in \{0, \dots, k+1\}$, be some sets; $R_i: V_i \to V_i'$ be some mappings; f_i, where $i \in \{0, \dots, k\}$, and g be abstraction functions; at that, $f_i(P_i, R_i)$ are bijective abstractions, while $g(P_{k+1}, \dots, P_N; R_{k+1})$ is a serializing abstraction. Then, the system

$$M' = \{f_i(P_i, R_i)\}_{i=0}^{k} \cup \{g(P_{k+1}, \dots, P_N; R_{k+1})\}$$

is called an *abstraction* of M. A process $f_i(P_i; R_i)$, where $i \in \{0, \dots, k\}$, is called an *abstraction of the process* P_i. The process $g(P_{k+1}, \dots, P_N; R_{k+1})$ is called an *abstraction of the environment*.

Statement. Let $M = \{P_i\}_{i=0}^{N}$ and $M' = \{P_i'\}_{i=0}^{k+1}$ be, respectively, a system and its abstraction. Given an arbitrary state s, if s is reachable in the state space of M, then there is a state s' reachable in the state space of M' such that $s' \sim s$.

Proof. Let $ABS = k + 1$. This denotation is introduced to emphasize that the abstraction of the environment, the process $P_{ABS}' = P_{k+1}'$, generalizes not only the process P_{k+1}, but also the processes $P_{k+2}, \dots, P_N$.

A configuration $\langle l', s' \rangle$ of M' is said to *conform* to a configuration $\langle l, s \rangle$ of M iff the following conditions are satisfied:

- $l'(i) = R_i\big(l(i)\big)$ for all $i \in \{0, \dots, k\}$;
- if $l'(ABS) = R_{ABS}(v_{0_{ABS}})$, then $l(i) = v_{0_i}$ for all $i \in \{k+1, \dots, N\}$;
- if $l'(ABS) \neq R_{ABS}(v_{0_{ABS}})$, then there is only one index $i \in \{k+1, \dots, N\}$ such that $l'(ABS) = R_i\big(l(i)\big)$;
- $s' \sim s$.

Let us consider a path in the state space of M starting with $\langle l_0, s_0 \rangle$:

$$\pi = \left\{\Big(\langle l_j, s_j \rangle, \Big(i_j, \big(v_j, \gamma_j \to \alpha_j, v_{j+1}\big)\Big), \langle l_{j+1}, s_{j+1} \rangle\Big)\right\}_{j=0}^{m-1}.$$

Here, $i_j \in \{0, \dots, N\}$ is a process index; $v_j = l_j\big(i_j\big) \in V_{i_j}$ and $v_{j+1} = l_{j+1}\big(i_j\big) \in V_{i_j}$ are the process's vertices connected with the edge labelled by $\gamma_j \to \alpha_j$; $s_j \vDash \gamma_j$ and $s_{j+1} = [\![\alpha_j]\!](s_j)$ for all $j \in \{0, \dots, m-1\}$.

Our goal is to show that, in the state space of M', there is a path π' of the same length as π such that each configuration of π' conforms to the corresponding configuration of π:

$$\pi' = \left\{\Big(\langle l_j', s_j' \rangle, (i_j', (v_j', \gamma_j' \to \alpha_j', v_{j+1}')), \langle l_{j+1}', s_{j+1}' \rangle\Big)\right\}_{j=0}^{m-1}.$$

Obviously, existence of such a path implies that there is a state s_m' reachable in the state space of M' such that $s_m' \sim s_m$. Let us consider how to construct π'.

Induction basis. The initial configuration $\langle l_0', s_0' \rangle$ certainly conforms to $\langle l_0, s_0 \rangle$: $v_{0_i}' = l'(i) = R_i\big(l(i)\big) = R_i(v_{0_i})$ for all $i \in \{0, \dots, N\}$.

Inductive step. Given an arbitrary index $q \in \{0, \dots, m-1\}$, we will show that if the configuration $\langle l'_q, s'_q \rangle$ conforms to $\langle l_q, s_q \rangle$, then there are a process of M' (let us denote its index as i'_q) and an edge $\left(v'_q, \gamma'_q \to \alpha'_q, v'_{q+1}\right)$ of that process such that $\langle l'_{q+1}, s'_{q+1} \rangle = \langle \left(l'_q \setminus \{i'_q \mapsto v'_q\}\right) \cup \left\{i'_q \mapsto v'_{q+1}\right\}, [\![\alpha']\!](s'_q) \rangle$ (see the definition of the state space) conforms to $\langle l_{q+1}, s_{q+1} \rangle$. There are two cases:

- $i_q \in \{0, \dots, k\}$;
- $i_q \in \{k+1, \dots, N\}$.

Case 1. If $i_q \in \{0, \dots, k\}$, let $i'_q = i_q$: the transition is executed by the process $P'_{i_q} = f_{i_q}(P_{i_q}, R_{i_q})$.

The edge $\left(v_q, \gamma_q \to \alpha_q, v_{q+1}\right)$ of the process P_{i_q} is abstracted to the set of edges $\left\{\left(R_{i_q}(v_q), \gamma_q^{(i)} \to \alpha_q^{(i)}, R_{i_q}(v_{q+1})\right)\right\}_{i=1}^{t}$, where $f_{i_q}\left(\gamma_q \to \alpha_q\right) = \left\{\gamma_q^{(i)} \to \alpha_q^{(i)}\right\}_{i=1}^{t}$. Among them, there is selected an edge whose label, $\gamma'_q \to \alpha'_q$, is an abstraction of $\gamma_q \to \alpha_q$ in s_q. Such an edge always exists (see the definition of the process abstraction). We need to proof that the chosen edge belongs to the state space of M' and the configuration $\langle l'_{q+1}, s'_{q+1} \rangle$ conforms to $\langle l_{q+1}, s_{q+1} \rangle$. It is sufficient to proof the following statements:

- $s'_q \vDash \gamma'_q$;
- $[\![\alpha']\!](s'_q) \sim [\![\alpha]\!](s_q)$.

The first of them can be deduced from the facts that $s_q \vDash \gamma_q$ (otherwise, the state space of M would not include the transition under consideration), $\gamma'_q \to \alpha'_q$ is an abstraction of $\gamma_q \to \alpha_q$ in s_q, and $s'_q \sim s_q$ (the induction assumption). Obviously, $s_q \vDash \gamma_q$ and $s_q \vDash \gamma_q \to \gamma'_q$ lead to $s_q \vDash \gamma'_q$, which, in couple with $s'_q \sim s_q$, leads to $s'_q \vDash \gamma'_q$. The second statement is an implication of the facts that $\gamma'_q \to \alpha'_q$ is an abstraction of $\gamma_q \to \alpha_q$ in s_q and $s'_q \sim s_q$.

Case 2. If $i_q \in \{k+1, \dots, N\}$, let $i'_q = ABS$: the transition is executed by the process $P'_{ABS} = g(P_{k+1}, \dots, P_N; R_{ABS})$. There are two subcases:

- the edge $\left(v_q, \gamma_q \to \alpha_q, v_{q+1}\right)$ is active;
- the edge $\left(v_q, \gamma_q \to \alpha_q, v_{q+1}\right)$ is passive.

Subcase 2.1. If the edge is active, then, by definition of configuration conformance, $l'(ABS) = R_{i_q}(v_q)$. In P'_{ABS}, there is selected an edge between $R_{i_q}(v_q)$ and $R_{i_q}(v_{q+1})$ whose label is an abstraction of $\gamma_q \to \alpha_q$ in s_q. Such an edge always exists (active edges are abstracted in a usual way). The further proof is similar to that in Case 1.

Subcase 2.2. If the edge is passive, then $R_{i_q}(v_q) = R_{i_q}(v_{q+1}) = R_{i_q}\left(v_{0_{i_q}}\right) = v'_{0_{ABS}}$. In P'_{ABS}, there is selected an edge $(v'_{0_{ABS}}, \varepsilon, v'_{0_{ABS}})$. Conformance of the configuration follows from the facts that passive edges do not depend on sufficient data and do not affect them.

Conclusion. Given an arbitrary path π in the state space of M, there is a path π' in the state space of M' such that the ending state of π' is equivalent to the ending state of π.

Q.E.D.

Corollary. Let $M = \{P_i\}_{i=0}^{N}$ and $M' = \{P_i'\}_{i=0}^{k+1}$ be, respectively, a system and its abstraction. Given an arbitrary formula φ over significant data, if φ is true (false) in all states reachable in the state space of M', then φ is true (false) in all states reachable in the state space of M.

4.3 Model transformation

This section defines abstraction functions used for protocol model transformation. The description is not quite formal: rigorous definition requires, first, formalization of the PROMELA semantics and, seconds, usage of formalisms for describing code transformations. Nevertheless, we believe that the explanations below are sufficient for formalizing and automating the abstraction procedure.

Let $M = \{P_i\}_{i=0}^{N}$ and $M' = \{P_i'\}_{i=0}^{k+1}$ be, respectively, a system (referred to as an *original model*) and its abstraction (referred to as an *abstract model*).

Let us recall that each message circulating in the model includes the sender's identifier. A state of a channel being written by $\{P_i\}_{i=k+1}^{N}$, as well as messages being read from the channel may contain identifiers from the set $\{k+1, \dots, N\}$. In the abstract model, there are no such identifiers: they are mapped to ABS (usually, $ABS = k+1$). The definition of state equivalence should be modified so as not to distinguish between i and ABS if $i \in \{k+1, \dots, N\}$.

Another issue is as follows. State of a channel's buffer is not of importance until a message is read. The idea is to ignore some messages (in particular, messages written by $\{P_i\}_{i=k+1}^{N}$). In this case, a send statement can be replaced with ε. To preserve the abstraction properties, each read from the channel should be supplied (as alternative behavior) with the assignments of all possible values that could be sent via the channel by the removed statement to the message variable.

To be more precise, the definition of state equivalence should take into account the following considerations:

- given a channel $c \in C_*$, an abstract state s' is (quasi) equivalent to a state s (state is a sequence of messages) iff s' is produced from s by removing all messages with identifiers from $\{k+1, \dots, N\}$;
- the channels from $C_{h\to e} = \bigcup_{j=k+1}^{N} C_{0\to j}$ are insignificant (every two states of a channel are equivalent);
- an abstract state s' of the channels $C_{e\to h} = \bigcup_{i=k+1}^{N} C_{i\to 0}$ (as a whole) is equivalent to a state s iff there is $i \in \{k+1, \dots, N\}$ such that for each $c \in C_{i\to 0}$, the state $s'(c')$, where c' is a channel that corresponds to c in P_{ABS}, is produced from $s(c)$ by replacing i with ABS while the remaining channels

are empty in both states.

The suggested approach implies the following restrictions on the input model:

- $Data_S = Data \setminus \left(\bigcup_{i=k+1}^{N} Data_{L_i}\right)$;
- for each $i \in \{1, \dots, N\}$, there holds $Chan_i = Chan_{A_i} \cup Chan_{P_i}$, where $Chan_{A_i}$ and $Chan_{P_i}$ are the sets of channels used, respectively, in the active and passive modes, and:
  - $Chan_{A_i} \cap Chan_{P_i} = \emptyset$;
  - $Chan_{A_i} \subseteq Chan_G$ $(Chan_{A_i} = C_{\{1,\dots,N\}\to 0} \cup C_{i\to 0})$;
  - $Chan_{P_i} \subseteq Chan_{L_i}$ $(Chan_{P_i} = C_{0\to i} \cup \left(\bigcup_{j=1}^{N} C_{\{1,\dots,N\}\to j}\right))$;
- the only channel predicate in use is **empty** (behavior does not depend on the number of messages in the channels' buffers);
- there are no dependencies via variables between the processes $\{P_i\}_{i=1}^{N}$ (all dependencies are via messages);
- each guarded action is closed under data dependencies via variables;
- there are no data dependencies from the local data (control dependencies from the local data are allowed).

$M' = \{P_i'\}_{i=0}^{k+1} = \{f_i(P_i, R_i)\}_{i=0}^{k} \cup \{g(P_{k+1}, \dots, P_N; R_{k+1})\}$, the abstract model, is constructed as follows (the description below can be viewed as a definition of the mappings R_i and the abstraction functions f_i and g). Initially, each process P_i', where $i \in \{0, \dots, k+1\}$, is isomorphic to P_i: $P_i' = I\left(P_i, R_{0_i}\right)$, where I is the trivial abstraction function, while $R_{0_i}: V_i \to V_i'$ is a bijection. Then, the following transformations are applied to $P'_{ABS} = P'_{k+1}$ and the rest of the processes:

- all passive edges of P'_{ABS} are removed and replaced with the ε-self loops;
- when removing a passive edge whose action contains a read from some channel c (a write to some channel c):
  - in $\{P_i'\}_{i=0}^{k}$, for all $j \in \{k+1, \dots, N\}$, all writes to c_j (all reads from c_j), where c_j is a channel of P_j that corresponds to c (the processes are identical), are removed;
  - when removing a read of a message m:
    - in the guards dependent on m, the minimal subformulae dependent on m are replaced with $undef$;
- the active edges of P'_{ABS} are processed as follows:
  - all assignments to the local variables are removed;
  - when removing an assignment to a local variable x:
    - in the guards dependent on x, the minimal subformulae dependent on x are replaced with $undef$;
  - each read from a global channel c is not modified:
    - in $\{P_i'\}_{i=0}^{k}$, writes to c are not modified;

- each write to a global channel c is removed:
  - in $\{P'_i\}_{i=0}^{k}$, each read $c\,?\,m$ is supplemented with the alternatives $\{m = v_j\}_{j=1}^{t}$, where $\{v_j\}_{j=1}^{t}$ contains all possible values that P'_{ABS} can send via c.

Statement. The processes $\{f_i(P_i, R_i)\}_{i=0}^{k}$ (constructed as it is described above) are bijective abstractions, while the process $g(P_{k+1}, \dots, P_N; R_{k+1})$ is a serializing abstraction. Thus, M' is an abstraction of M.

As the description is informal, the statement is given without a proof. It should be noticed that the abovementioned method has been implemented in a tool prototype. Given a PROMELA model, the tool parses the code, builds the abstract syntax tree, transforms it according to the rules, and maps it back to PROMELA.

5. Case study

The tool and the underlying method were used to verify the MOSI family CCPs implemented in the Elbrus computer systems. The developed PROMELA model supports memory accesses of the types Write Back, Write Through, and Write Combined. The experiments were performed on Intel Core i7-4771 with a clock rate of 3.5 GHz. The verified properties are as follows:

- $\mathbf{G}\{\neg(cache[1] = M \wedge cache[2] = M)\}$;
- $\mathbf{G}\{\neg(cache[1] = O \wedge cache[2] = O)\}$;
- $\mathbf{G}\{\neg(cache[1] = M \wedge cache[2] \in \{O, S\})\}$.

Table 1 and Table 2 show time and memory resources consumed for checking the property (1), respectively, on the original model ($n = 3$) and on the abstract one. Note that in the case $n = 3$ abstraction preserves the number of processes: $home(0)$, $proc(1)$, and $proc(2)$ are replaced with their abstract counterparts, while *proc*(3) is replaced with $proc_{env}(ABS)$.

Table 1. Resources required for checking the original model

SPIN optimization	State space size	Memory consumption	Verification time
$Absent$	5.1×10^6	682 Mb	9 s
$COLLAPSE$	5.1×10^6	328 Mb	15 s

Table 2. Resources required for checking the abstract model

SPIN optimization	State space size	Memory consumption	Verification time
$Absent$	2.2×10^6	256 Mb	3.7 s
$COLLAPSE$	2.2×10^6	108 Mb	6.2 s

The tables show that even for $n = 3$ there is a gain in state space size and memory consumption. Meanwhile, correctness of the abstract model implies correctness of the

original one for any $n \geq 3$. It is shown that the suggested approach reduces verification of the parameterized CCP model to visiting and testing $\sim 10^6$ states, which requires ~ 100 Mb of memory.

6. Conclusion

SMP computer systems utilize complicated caching mechanisms. To ensure that multiple copies of the same data are kept up-to-date, CCPs are employed. Errors in the CCPs and their implementations may cause data corruption and system hanging. This explains why CCP verification methods are of high value and importance.

The main problem arising in CCP verification is state explosion. In this paper, we have proposed an approach to overcome the issue and make verification scalable. The method having been described is aimed at transforming a CCP PROMELA model so as the result is independent of the number of processors and can be verified by the SPIN model checker on a regular basis. The approach was successfully applied to the MOSI family CCPs implemented in the Elbrus computer systems.

In the future, we are planning to extend the method with CEGAR, to develop an open-source tool for syntactical transformations of PROMELA models (a prototype is already available), and to create a unified model-based technology for checking CCPs and verifying memory management units.

References

[1]. Patterson D.A., Hennessy J.L. Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 2013. 800 p.

[2]. Kim A.K., Perekatov V.I., Ermakov S.G. Microprocessors and computer systems of the Elbrus familty. SPb.: Piter, 2013. 272 p. (in Russian).

[3]. Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011. 195 p.

[4]. Kamkin A.S., Petrochenkov M.V. A system to support formal methods-based verification of coherence protocol implementations. Voprosy radioehlektroniki. Ser. EVT. [Issues of radio electronics], 2014, issue 3, pp. 27-38 (in Russian).

[5]. Clarke E.M., Grumberg O., Peled D.A. Model Checking. MIT Press, 1999. 314 p.

[6]. Burenkov V.S. An analysis of the SPIN model checker applicability to cache coherence protocols verification. Voprosy radioehlektroniki. Ser. EVT [Issues of radio electronics], 2014, issue 3, pp. 126-134 (in Russian).

[7]. Emerson E.A., Kahlon V. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, 2003, pp. 247-262.

[8]. Holzmann, G.J. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 2003, 608 p.

[9]. Park S., Dill D.L. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. Annual ACM Symposium on Parallel Algorithms and Architectures, 1996, pp. 288-296.

[10]. Ip C.N., Dill D.L. Verifying Systems with Replicated Components in Murphi. International Conference on Computer Aided Verification, 1996, pp. 147-158.

[11]. Pnueli A., Xu J., Zuck L. Liveness with $(0, 1, \infty)$-Counter Abstraction. International Conference on Computer Aided Verification, 2002, pp. 107-122.

[12]. Clarke E., Talupur M., Veith H. Environment Abstraction for Parameterized Verification. Verification, Model Checking, and Abstract Interpretation, 2006. LNCS, vol. 3855, pp. 126-141.
[13]. Clarke E., Talupur M., Veith H. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems. International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 33-47.
[14]. McMillan K. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. Conference on Correct Hardware Design and Verification Methods, 2001, pp. 179-195.
[15]. Chou C.-T., Mannava P.K., Park S. A Simple Method for Parameterized Verification of Cache Coherence Protocols. Formal Methods in Computer-Aided Design, 2004. LNCS, vol. 3312, pp. 382-398.
[16]. Krstic S. Parameterized System Verification with Guard Strengthening and Parameter Abstraction. International Workshop on Automated Verification of Infinite-State Systems, 2005.
[17]. Talupur M., Tuttle M.R. Going with the Flow: , pp. 1-8.
[18]. O'Leary J., Talupur M., Tuttle M.R. Protocol Verification Using Flows: An Industrial Experience. Formal Methods in Computer-Aided Design, 2009, pp. 172-179.

Проверка параметризованных PROMELA-моделей протоколов когерентности памяти

[1] *В.С. Буренков <burenkov_v@mcst.ru>*
[2] *А.С. Камкин <kamkin@ispras.ru>*
[1] *АО «МЦСТ»*
119334, Россия, г. Москва, ул. Вавилова, 24.
[2] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, 25

Аннотация. В статье представлен метод масштабируемой верификации PROMELA-моделей протоколов обеспечения когерентности памяти. Под масштабируемостью понимается независимость затрат на верификацию (прежде всего, машинного времени и памяти) от числа процессоров в системе. Метод состоит из трех основных шагов. На первом шаге в модель протокола, созданную для определенной конфигурации системы (для конкретного числа процессоров), вводится параметр, представляющий число процессоров в системе. Для этого используются простые индуктивные правила, что возможно только при определенных допущениях на вид протокола. На втором шаге построенная параметризованная модель абстрагируется от числа процессоров. Для этого над присваиваниями, выражениями и коммуникационными действиями модели совершается ряд синтаксических преобразований. На третьем шаге полученная абстрактная модель верифицируется с помощью инструмента SPIN обычным образом. Помимо описания метода, в статье приводится доказательство его корректности:

утверждается, что предложенная схема абстракции является консервативной в том смысле, что любой инвариант (свойство истинное во всех достижимых состояниях) абстрактной модели является инвариантом исходной модели (свойства-инварианты — это именно те свойства, которые представляют интерес при верификации протоколов обеспечения когерентности памяти). Предложенный метод был воплощен в прототипе инструмента, который разбирает код на языке PROMELA, строит дерево абстрактного синтаксиса, преобразует его по заданным правилам и отображает обратно в PROMELA код. Инструмент (и метод в целом) был успешно использован при верификации протоколов семейства MOSI, разработанных в АО «МЦСТ» и реализованных в вычислительных комплексах «Эльбрус».

Ключевые слова: многоядерные микропроцессоры, мультипроцессоры с разделяемой памятью, протоколы когерентности памяти, проверка моделей, SPIN, PROMELA.

DOI: 10.15514/ISPRAS-2016-28(4)-4

Для цитирования: Буренков В.С., Камкин А.С. Проверка параметризованных PROMELA-моделей протоколов когерентности памяти. Труды ИСП РАН, том 28, вып. 4, 2016 г. стр. 57-76 (на английском). DOI: 10.15514/ISPRAS-2016-28(4)-4

Список литературы

[1]. Patterson D.A., Hennessy J.L. Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 2013. 800 p.
[2]. Ким А.К., Перекатов В.И., Ермаков С.Г. Микропроцессоры и вычислительные комплексы семейства «Эльбрус». Спб.: Питер, 2013. 272 с.
[3]. Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011, 195 p.
[4]. Камкин А.С., Петроченков М.В. Система поддержки верификации реализаций протоколов когерентности с использованием формальных методов. Вопросы радиоэлектроники. Серия ЭВТ, 2014, вып. 3, стр. 27-38.
[5]. Clarke E.M., Grumberg O., Peled D.A. Model Checking. MIT Press, 1999, 314 p.
[6]. Буренков В.С. Анализ применимости инструмента SPIN к верификации протоколов когерентности памяти. Вопросы радиоэлектроники. Серия ЭВТ, 2014. вып. 3, стр. 126-134.
[7]. Emerson E.A., Kahlon V. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, 2003, pp. 247-262.
[8]. Holzmann, G.J. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 2003, 608 p.
[9]. Park S., Dill D.L. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. Annual ACM Symposium on Parallel Algorithms and Architectures, 1996, pp. 288-296.
[10]. Ip C.N., Dill D.L. Verifying Systems with Replicated Components in Murphi. International Conference on Computer Aided Verification, 1996, pp. 147-158.
[11]. Pnueli A., Xu J., Zuck L. Liveness with (0, 1, ∞)-Counter Abstraction. International Conference on Computer Aided Verification, 2002, pp. 107-122.

[12]. Clarke E., Talupur M., Veith H. Environment Abstraction for Parameterized Verification. Verification, Model Checking, and Abstract Interpretation, 2006. LNCS, vol. 3855, pp. 126-141.
[13]. Clarke E., Talupur M., Veith H. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems. International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 33-47.
[14]. McMillan K. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. Conference on Correct Hardware Design and Verification Methods, 2001, pp. 179-195.
[15]. Chou C.-T., Mannava P.K., Park S. A Simple Method for Parameterized Verification of Cache Coherence Protocols. Formal Methods in Computer-Aided Design, 2004. LNCS, vol. 3312, pp. 382-398.
[16]. Krstic S. Parameterized System Verification with Guard Strengthening and Parameter Abstraction. International Workshop on Automated Verification of Infinite-State Systems, 2005.
[17]. Talupur M., Tuttle M.R. Going with the Flow: , pp. 1-8.
[18]. O'Leary J., Talupur M., Tuttle M.R. Protocol Verification Using Flows: An Industrial Experience. Formal Methods in Computer-Aided Design, 2009, pp. 172-179.

Language for Describing Templates for Test Program Generation for Microprocessors

A.D. Tatarnikov <andrewt@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Test program generation and simulation is the most widely used approach to functional verification of microprocessors. High complexity of modern hardware designs creates a demand for automated tools that are able to generate test programs covering non-trivial situations in microprocessor functioning. The majority of such tools use test program templates that describe scenarios to be covered in an abstract way. This provides verification engineers with a flexible way to describe a wide range of test generation tasks with minimum effort. Test program templates are developed in special domain-specific languages. These languages must fulfill the following requirements: (1) be simple enough to be used by verification engineers with no sufficient programming skills; (2) be applicable to various microprocessor architectures and (3) be easy to extend with facilities for describing new types of test generation tasks. The present work discusses the test program template description language used in the reconfigurable and extensible test program generation framework MicroTESK being developed at ISP RAS. It is a flexible Ruby-based domain-specific language that allows describing a wide range of test generation tasks in terms of hardware abstractions. The tool and the language have been applied in industrial projects dedicated to verification of MIPS and ARM microprocessors.

Keywords: microprocessors; functional verification; test program generation; test templates; domain-specific languages.

DOI: 10.15514/ISPRAS-2016-28(4)-5

For citation: Tatarnikov A.D. Language for Describing Templates for Test Program Generation for Microprocessors. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016. pp. 77-98. DOI: 10.15514/ISPRAS-2016-28(4)-5

1. Introduction

Functional verification is acknowledged to be the bottleneck in microprocessor design cycle. According to various estimates, it accounts for more than 70% of overall project time and resources. In the current industrial practice, function verification mainly relies on *test program generation (TPG)* which is done by special automation tools [1]. Generated *test programs (TP)* are instruction sequences aimed to trigger

certain events in the microprocessor design under verification. TPG tools are aimed to provide a high level of test coverage by applying a rich set of generation methods. As modern microprocessors are getting more and more complex, new more advanced methods emerge. A common problem for TPG tool developers is how to overcome the complexity and make it easy to apply the growing set of methods to a wide range of microprocessor designs.

One of possible ways to increase the flexibility of a TPG tool is to separate generation logic from descriptions of test cases. This method is known as *template-based* generation. The key idea of the method is that test programs are generated on the basis of abstract descriptions called test program templates or *test templates (TTs)*. The method helps generate high-quality tests directed towards specific situations or classes of situations. TTs specify methods to be used for constructing instruction sequences and constraints on instruction operand values which must be satisfied to make certain events to fire. Test data are generated by finding random solutions to the given constraint systems. Such approach is usually referred to as *constraint-based* random generation [2]).

The *template-based* approach is implemented in a number of TPG tools including MicroTESK [3], a reconfigurable [4] and extensible [5] TPG framework being developed at ISP RAS. The framework uses *formal specifications* to construct TPG tools for specific microprocessor designs. A constructed TPG tool is separated into two main components: (1) an architecture-independent test generation core and (2) an architecture specification, or a model. The approach called *model-based* [1] helps significantly reduce the efforts to support a new microprocessor architecture by reusing the core. The core is designed as a set of generation engines which can be easily extended with plugins implementing new TPG methods. Test programs are generated by processing TTs that describe verification tasks in terms of the model and the generation methods implemented by the core.

This paper describes the test template description language (TTDL) used in MicroTESK. This is a domain-specific language implemented as a set of Ruby [6] libraries, which is easy adaptable to changing configurations. Facilities for describing instruction calls for a specific ISA are dynamically added and are based on information provided by the model. Also, the MicroTESK TTDL provides a rich set of facilities for describing verification tasks which are common for all microprocessor configurations. When MicroTESK is extended with new TPG methods, support for these features is added in the TTDL by providing new Ruby libraries.

The rest of the paper is divided into five sections. Section 2 contains a brief survey of the existing TPG tools that follow the template-based approach. Section 3 formulates the requirements for a TTDL imposed by MicroTESK that led to creating the described TTDL. Section 4 provides a detailed description of the architecture and facilities of the MicroTESK TTDL. Section 5 contains a case study of applying the TTDL for describing test cases in industrial projects. Section 6 discusses the results and outlines directions of future research and development.

2. Related work

Functional verification has always been a major issue for the research community. Over the last decades, a lot of TPG methods and tools have emerged. The template-based approach described in this paper has been applied in a number of tools developed by different teams. This section gives an overview of the most significant of existing TPG tools and discusses strong and weak points of their TTDLs.

IBM Research has been one of the major contributors in the field of TPG for microprocessors during the last decades. Genesys-Pro [1], one of their most recent tools, uses TTs to describe TPG tasks as constraint satisfaction problems (CSP) [2] and generates test data by solving these CSPs. Constraints can be used to specify such aspects of functionality as boundary conditions, exceptions, cache hits/misses, etc. The TTDL used by Genesys-Pro is a completely impendent domain-specific language which provides a rich set of features. The language features it offers can be divided into four groups: (1) basic instruction statements, (2) sequencing-control statements, (3) standard programming constructs, and (4) constraint statements. By combining these constructs, users can compose complex TTs with a degree of randomness varied from completely random to completely directed. The main advantage of the language is that it is designed for describing test scenarios and it does not confuse verification engineers with any unnecessary programming constructs. At the same time, being not based on existing languages, it does not take advantage of well-tried constructs that can help organize TTs into reusable libraries. This can be important as industrial testbenches usually contain thousands lines of code. Also, it is unclear how easy the language can be extended with new constructs for describing new types of TPG tasks.

Another company that has made a significant contribution in development of TPG tools is Obsidian Software (now acquired by ARM) [7]. Their tool RAVEN (Random Architecture Verification Engine) [8] generates random and directed tests based on TTs. Test templates are focused on coverage grids and use constraints to formulate specific coverage goals. There is no detailed information available on this technology. It is known that TTs can be either generated by the tool's GUI or created as text. The language must suit well for the TPG tasks that can be accomplished with RAVEN. However, the question whether it is suitable for more general tasks stays open.

Also, Samsung Electronics created a TPG framework called RDG (Random Diagnostics Generator) [9] for testing reconfigurable processors. It uses TTs created in the C++ language to specify instructions that will be used in a TP and constraints on their input values that should be satisfied in order to meet testing goals. This approach takes advantage of power and performance of C++, but requires solid programming skills which are not common for verification engineers.

Finally, MicroTESK [3] version 1.0 used TTs written using Java libraries [10]. This is not convenient as verification engineers are forced to deal with Java abstractions such as classes and interfaces, which are not related to verification tasks. Moreover, details of language implementation must be hidden from users in order to be able to change it without breaking existing TTs. This motivated to create a new domain-specific language for the new version of MicroTESK.

3. Requirements for TTDL

Requirements for a TTDL can be divided in two groups: (1) general requirements for a TTDL; (2) requirements related to integration into the MicroTESK framework. Let us first consider the general requirements that are common for all TTDLs. A TTDL used to describe scenarios for random and directed tests must provide facilities:

1) to describe instructions calls and data definitions using syntax similar to the one used in assembly code;
2) to manage memory allocations in the same way as in the assembly language;
3) to fill memory with data generated according to specific rules;
4) to compose instruction sequences using a wide range of methods (random, combinatorial, etc.) and to merge these sequences;
5) to specify random values and the degree of their randomness described by distributions;
6) to select instructions at random with the specified degree of randomness;
7) to specify constraints on instruction arguments;
8) to describe initialization code that places generated test data to proper registers or memory addresses;
9) to specify code of self-checks that check validity of the resulting state of the microprocessor;
10) to describe exception handlers;
11) to specify conditions for generating different code depending on the context;
12) to insert comments and custom text into generated TPs;
13) to reuse existing TTs and their parts;
14) to split generated TPs into multiple files.

This list is not complete, but it is enough to conclude that the TTDL must be a domain-specific language that provides constructs for the listed facilities.

Another important consideration is that it must be integrated into MicroTESK. First of all, MicroTESK is written in Java and its generation engines operate with Java objects. Therefore, the result of TT processing must be a hierarchy of Java objects that then will be passed to TPG engines. The front-end of a TTDL processor can be implemented using two approaches: (1) creating a Java-based parser for the new language or (2) reusing an existing Java-based parser for one of the popular programming languages. A crucial requirement for the second approach is that the language must be easy to extend with new domain-specific constructs.

Now let us consider the requirements imposed by reconfigurability and extensibility of MicroTESK:

1) *Reconfigurability* means that it can be applied to microprocessors with different ISAs. Consequently, facilities used to describe instruction calls must be changeable. Ideally, they must be added dynamically depending on

the information provided in the model that describes the configuration of the design under verification.

2) *Extensibility* means that the set of supported TPG methods can be extended by adding plugins implementing new methods. Often it will require adding new constructs in the TTDL. Thus, it must be possible to dynamically add language constructs depending on the installed plugins.

In other words, a crucial requirement for the MicroTESK TTDL is the ability to dynamically change the set of supported language constructs. Obviously, changes in the tool configuration must not involve modification of the TTDL processor. Creating a flexible language processor from scratch is a challenging task. A simpler solution would be to reuse a parser of an existing language.

Having considered several possible alternatives, it was decided to use JRuby [11], a Java-based implementation of the Ruby language, as a front-end of the TTDL processor. Ruby was selected because of its support for *metaprogramming* [12], which allows adding new language features at runtime. Thus, the created TTDL combines basic programming constructs provided by the Ruby core with constructs for describing TTs provided by MicroTESK. The TTDL front-end is implemented as a set of Ruby libraries that define language facilities for the above mentioned requirements. Facilities that depend on the current configuration are dynamically added using metaprogramming.

It is also worth mentioning that scripting languages like Ruby are quite popular among verification engineers, who often use them to create in-house test generators. So, another advantage of using Ruby is that it can make the TTDL easier to learn.

4. TTDL Description

4.1 Language Processor Architecture

The job of the TTDL processor is to build a hierarchy of Java objects describing a TT and to pass it to the MicroTESK generation engines for further processing. The TTDL processor is divided into a *Ruby-based front-end* and *Java-based back-end*. The back-end is implemented as set of factories for creating Java objects that correspond to specific entities of a TT. The front-end is represented by Ruby libraries that provide language constructs for describing these entities and perform interaction with the back-end to build corresponding Java objects. In other words, a language feature is defined by a Ruby module that specifies its syntax and a Java module that describes corresponding entities and provides means of constructing them. New language features can be supported by providing corresponding modules.

The TTDL contains features that are configuration dependent. This includes facilities for describing instruction calls, which are determined by the model built by MicroTESK from ISA specifications. These language features are managed by a special Ruby module that uses metaprogramming to define corresponding constructs at runtime based on the information provided by the model.

4.2 Test Template Structure

A TT is a program in Ruby executed by MicroTESK with the help of JRuby to build Java objects that formulate tasks for the TPG engines implemented by the tool core. More technically, it is a subclass of the *Template* base class provided by the MicroTESK library. All domain-specific language constructs are implemented as methods of this class. The *Template* class is not monolithic, it unites a set of Ruby modules responsible for various features into a single class. Language extensions are also implemented as modules to be included in the base class. Configuration-specific methods are dynamically defined when the class is loaded.

The listing below shows the structure of a TT class:

```
require ENV['TEMPLATE']
class MyTemplate < Template
  def initialize
    super
    # Initialize settings here
  end

  def pre
    # Place your initialization code here
  end

  def post
    # Place your finalization code here
  end

  def run
    # Place your testing task description here
  end
end
```

The first line imports the *Template* base class from the location specified by the *TEMPLATE* environment variable. The exact location depends on the configuration and is determined automatically.

Classes describing TTs define four methods:

- *initialize* - configures TT settings if there is a need to override the default;
- *pre* - defines ISA-specific constructs and specifies initialization code to be inserted in the beginning of TPs;
- *post* - specifies finalization code to be inserted in the end of TPs;
- *run* - contains descriptions of test cases to be generated.

The methods will be filled with constructs described further.

4.3 Managing Memory Allocation

It may be required to place code and data sections of generated TPs at specific memory locations. The assembly language provides special directives to accomplish this task. The TTDL offers similar constructs. An important note is that MicroTESK

simulates TPs in the process of their generation. Consequently, these constructs not only specify directives to be placed into TPs, but also manage memory allocation in the simulator.

The TTDL provides the following methods for managing addresses, which are applicable to both code and data sections:

- *align* - aligns the allocation address by the amount n passed as an argument, which by default means 2n bytes.
- *org* - sets the allocation origin, which is required to increase the allocation address. It is possible to set an *absolute* or *relative* origin. The former can be specified as org n and means an offset by n bytes from the base virtual address. The latter can be specified as *org :delta=>n* and means an offset by n bytes from the most recent allocation address.
- *label* - associates the specified label with the current address.

The listed methods rely on the following TT settings:

- *align_format* - specifies textual format for the align directive;
- *org_format* - specifies textual format for the org directive;
- *base_virtual_address* - specifies the base virtual address for memory allocation;
- *base_physical_address* - specifies the base physical address for memory allocation;
- *alignment_in_bytes* - specifies how the alignment amount should be interpreted.

The first four settings are initialized with default values in the *initialize* method of the *Template* base class as shown below and can be changed in the current TT class:

```
@org_format = ".org 0x%x"
@align_format = ".align %d"
@base_virtual_address = 0x0
@base_physical_address = 0x0
```

The last setting is implemented as a method that can be overridden to change its behavior:

```
def alignment_in_bytes(n) 2 ** n end
```

4.4 Defining Random Distributions

Many TPG tasks involve selection based on random distribution. The TTDL provides the following methods to define random distributions:

- *range* - creates an object describing a range of values and its weight, which are specified by the *value* and *bias* attributes. Values can be one of the following types:
  - *single* value;

- *range* of values;
- *array* of values;
- *distribution* of values.

The *bias* attribute can be skipped which means default weight. Default weights are used to specify an even distribution based on ranges with equal weights.

- *dist* - creates an object describing a random distribution from a collection of ranges.

The code below illustrates how to create weighted distributions for integer numbers:

```
simple_dist = dist(
  range(:value => 0, :bias => 25),          # Value
  range(:value => 1..2, :bias => 25),       # Range
  range(:value => [3, 5, 7], :bias => 50)  # Array
  )
composite_dist = dist(
  range(:value=> simple_dist, :bias => 80), # Distribution
  range(:value=> [4, 6, 8], :bias => 20)       # Array
  )
```

4.5 Describing Data Definitions

Data definitions are based on assembler-specific directives, which are not described by the microprocessor model and, therefore, must be configured in TTs. The configuration information includes textual format of the directives and mappings between data types used by the assembler and the microprocessor model. Data directives are configured using the *data_config* construct, which must be placed in the pre method. Here is an example:

```
data_config(:text=>".data", :target=>"MEM") {
  define_type :id=>:byte, :text=>".byte", :type=>card(8)
  define_type :id=>:half, :text=>".half", :type=>card(16)
  define_type :id=>:word, :text=>".word", :type=>card(32)
  define_space :id=>:space, :text=>".space", :fillWith=>0
  define_ascii :id=>:ascii, :text=>".ascii", :zero=>false
  define_ascii :id=>:asciiz, :text=>".asciiz", :zero=>true
}
```

The *data_config* method has the following parameters:

- *text* - specifies the textual format of a directive that marks the beginning of a data section;
- *target* - specifies the memory array defined in the model to which data will be placed during simulation;
- *base_virtual_address* (optional, *0* by default) - specifies the base virtual address for data sections.

Distinct data directives are configured using special methods that must be called inside the *data_config* block. All of these methods share two common parameters: *id* and *text*. The first specifies the keyword to be used in a TT to address the directive and the second specifies how it will be printed into the TP. Here is the list of methods:

- *define_type* - defines a directive to allocate memory for a data element of the data type specified by the *type* parameter;
- *define_space* - defines a directive to allocate memory filled with a default value specified by the *fillWith* parameter;
- *define_ascii_string* - defines a directive to allocate memory for an ASCII string terminated or not terminated with zero depending on the *zero* parameter.

The above example defines directives *byte*, *half*, *word*, *ascii* (non-zero terminated string) and *asciiz* (zero terminated string) that place data in the memory array *MEM* defined in the microprocessor model.

Once data directives have been configured, data sections can be defined using the *data* construct. Data definitions can be of two kinds depending on the context:

1) *Global data* that are available to all test cases generated from the given TT. They are defined in the root of the *pre* or *run* methods. Global data are placed into the simulator's memory during initial processing of a TT.
2) *Test case level data* that are defined and used by specific test cases. Such data are placed into the simulator's memory when the test case is being generated.

The data method has two optional parameters:

- *global* - a flag that states that the data definition should be treated as global regardless of the context.
- *separate_file* - a flag that specifies whether the generated data definitions should be placed into a separate source code file.

Here is an example of a data definition:

```
data(:global => true, :separate_file => false) {
  org 0x00001000
  label :byte_values
  byte 1, 2, 3, 4
  label :word_values
  word 0xDEADBEEF, 0xBAADF00D
}
```

The above code defines global data: four byte values and two word values. Memory is allocated at offset *0x00001000*. Data values are aligned by their size (1 and 4 bytes). Labels *byte_values* and *word_values* point at the beginning of the byte and the word arrays correspondingly.

4.6 Describing Instruction Calls

To describe instruction calls, the TTDL provides runtime methods that are defined using the metaprogramming facilities of Ruby on the basis of information provided by the model. Methods have the same names and parameters as operations describing corresponding instructions, which are defined in ISA specifications. Operations use parameters of three kinds:

1) *Immediate values* that represent constants.
2) *Addressing modes* that encapsulate logic of reading and writing data to memory resources. Usually they provide access to registers or memory.
3) *Operations* that specify operations to be performed as a part of execution of the current operation. They are used to describe complex instructions composed of several operations (e.g. VLIW instructions).

For example, a call to the add instruction from the MIPS ISA [13], which adds two general-purpose registers *t0 ($8)*, *t1 ($9)* and *t2 ($10)* described by the *reg* addressing mode, can be specified in the following way:

```
add reg(8), reg(9), reg(10)
```

The TTDL supports creating *aliases* for addressing modes and operations invoked with certain arguments. Aliases help make TTs more human-readable. They are created by defining Ruby functions with corresponding names. The code below shows how to create aliases for the registers from the previous example:

```
def t0 reg(8) end
def t1 reg(9) end
def t2 reg(10) end
```

Now the arguments of the add instruction can be specified using alises:

```
add t0, t1, t2
```

Also, the TTDL provides the *pseudo* function that can be used to specify calls to *pseudo instructions* that do not have corresponding operations in ISA specifications. They print user-specified text, but are not simulated by the generator. Here is an example:

```
pseudo 'syscall'
```

4.7 Defining Groups

Addressing modes and operations can be organized into *groups*. Groups are used when it is required to randomly select an addressing mode or operation from the specified set. Groups can be defined in ISA specifications or in TTs. To do this in TTs, the *define_mode_group* and *define_op_group* functions are used. Both functions take the *name* and *distribution* arguments that specify the group name and the distribution used to select its items.

For example, the code below defines an instruction group called *alu* that contains instructions *add*, *sub*, *and*, *or*, *nor*, and *xor* selected randomly according to the specified distribution:

```
alu_dist = dist(
  range(:value => 'add', :bias => 40),
  range(:value => 'sub', :bias => 30),
  range(:value => ['and', 'or', 'nor', 'xor'], :bias => 30)
)
define_op_group('alu', alu_dist)
```

The following code specifies three calls that use instructions randomly selected from the *alu* group:

```
alu t0, t1, t2
alu t3, t4, t5
alu t6, t7, t8
```

4.8 Describing Instruction Call Sequences

Instruction call sequences are described using block-like structures. Each block specifies a sequence or a collection of sequences. Blocks can be nested to construct complex sequences. The algorithm used for sequence construction depends on the type and the attributes of a block.

An individual instruction call is considered a primitive block describing a single sequence that consists of a single instruction call. A single sequence that consists of multiple calls can be described using the *sequence* or the *atomic* construct. The difference between the two is that an atomic sequence is never mixed with other instruction calls when sequences are merged. The code below demonstrates how to specify a sequence of three instruction calls:

```
sequence {
  add t0, t1, t2
  sub t3, t4, t5
  or t6, t7, t8
}
```

A collection of sequences that are processed one by one can be specified using the *iterate* construct. For example, the code below describes three sequences consisting of one instruction call:

```
iterate {
  add t0, t1, t2
  sub t3, t4, t5
  or t6, t7, t8
}
```

Sequences can be combined using the *block* construct. The resulting sequences are constructed by sequentially applying the following engines to sequences returned by nested blocks:

- *combinator* - builds combinations of sequences returned by nested blocks. Each combination is a tuple of length equal to the number of nested blocks.
- *permutator* - modifies combinations returned by combinator by rearranging some sequences.

- *compositor* - merges (multiplexes) sequences in a combination into a single sequence preserving the initial order of instructions calls in each sequence.
- *rearranger* - rearranges sequences constructed by compositor.
- *obfuscator* - modifies sequences returned by rearranger by permuting some instruction calls.

Each engine has several implementations based on different methods. It is possible to extend the list of supported methods with new implementations. Specific methods are selected by specifying corresponding block attributes. When they are not specified, default methods are applied. The format of a block structure for combining sequences looks as follows:

```
block(
  :combinator => 'combinator-name',
  :permutator => 'permutator-name',
  :compositor => 'compositor-name',
  :rearranger => 'rearranger-name',
  :obfuscator => 'obfuscator-name') {
  # Block A. 3 sequences of length 1: {A11}, {A21}, {A31}
  iterate { A11; A21; A31 }
  # Block B. 2 sequences of length 2: {B11, B12}, {B21, B22}
  iterate { sequence { B11, B12 }; sequence { B21, B22 } }
  # Block C. 1 sequence of length 3: {C11, C12, C13}
  iterate { sequence { C11; C12; C13 } }
}
```

The default method names are: *diagonal* for combinator, *catenation* for compositor, and *trivial* for permutator, rearranger and obfuscator. Such a combination of engines describes a collection of sequences constructed as a concatenation of sequences returned by nested blocks. For example, sequences constructed for the block in the above example will be as follows: *{A11, B11, B12, C11, C12, C13}*, *{A21, B21, B22, C11, C12, C13}* and *{A31, B11, B12, C11, C12, C13}*.

4.9 Specifying Test Situations

Test situations are associated with specific instruction calls and specify methods used to generate their input data. There is a wide range of data generation methods implemented by various data generation engines. Test situations are specified using the *situation* construct. It takes the situation name and a map of optional attributes that specify situation-specific parameters. For example, the following line of code causes input registers of the add instruction to be filled with zeros:

```
add t1, t2, t3 do situation('zero') end
```

When no situation is specified, a default situation is used. This situation places random values into input registers. It is possible to assign a custom default situation for individual instructions and instruction groups with the *set_default_situation* function. For example:

```
set_default_situation 'add' do situation('zero') end
```

Situations can be selected at random. The selection is based on a distribution. This can be done by using the *random_situation* construct. For example:

```
sit_dist = dist(
  range(:value => situation('add.overflow')),
  range(:value => situation('add.normal')),
  range(:value => situation('zero')),
  range(:value => situation('random', :dist => int_dist))
  )
add t1, t2, t3 do random_situation(sit_dist) end
```

Unknown immediate arguments that should have their values generated are specified using the "_" symbol. For example, the code below states that a random value should be added to a value stored in a random register and the result should be placed to another random register:

```
addi reg(_), reg(_), _ do situation('random') end
```

4.10 Selecting Registers

Unknown immediate arguments of addressing modes are a special case and their values are generated in a slightly different way. Typically, they specify register indexes and are bounded by the lenght of register arrays. Often such indexes must be selected from a specific range taking into account previous selections. For example, registers are allocated at random and they must not overlap. To be able to solve such tasks, all values passed to addressing modes are tracked. The allowed value range and the method of value selection are specified in configuration files. Values are selected using the specified method before the instruction call is processed by the engine that generates data for the test situation. The selection method can be customized by using the *mode_allocator* function. It takes the allocation method name and a map of method-specific parameters. For example, the following code states that the output register of the add instruction must be a random register which is not used in the current test case:

```
add reg(_ mode_allocator('free')), t0, t1
```

Also, the TTDL allows customizing the allowed range for selected values. It is possible to exclude some elements from the range by using the *exclude* attribute or to provide a new range by using the *retain* attribute. For example:

```
add reg(_ :exclude=>[1, 5, 7]), t0, t1
add reg(_ :retain=>8..15), t0, t1
```

Addressing modes with specific argument values can be marked as free using the *free_allocated_mode* function. To free all allocated addressing modes, the *free_all_allocated_modes* function can be used.

4.11 Describing Preparators

Preparators describe instruction sequences that place data into registers or memory accessed via the specified addressing mode. They are inserted into TPs to set up the

initial state of the microprocessor required by test situations. It is possible to overload preparators for specific cases (value masks, register numbers, etc). Preparators are defined in the *pre* method using the *preparator* construct, which uses the following parameters describing conditions under which it is applied:

- *target* - the name of the target addressing mode;
- *mask* (optional) - the mask that should be matched by the value in order for the preparator to be selected;
- *arguments* (optional) - values of the target addressing mode arguments that should be matched in order for the preparator to be selected;
- *name* (optional) - the name that identifies the current preparator to resolve ambiguity when there are several different preparators that have the same target, mask and arguments.

It is possible to define several *variants* of a preparator which are selected at random according to the specified distribution. They are described using the *variant* construct. It has two optional parameters:

- *name* (optional) - identifies the variant to make it possible to explicitly select a specific variant;
- *bias* - specifies the weight of the variant, can be skipped to set up an even distribution.

Here is an example of a preparator what places a value into a 32-bit register described by the *REG* addressing mode and two its special cases for values equal to *0x00000000* and *0xFFFFFFFF*:

```
preparator(:target => 'REG') {
  variant(:bias => 25) {
    data {
      label :preparator_data
      word value
    }
    la at, :preparator_data
    lw target, 0, at
  }
  variant(:bias => 75) {
    lui target, value(16, 31)
    ori target, target, value(0, 15)
  }
}
preparator(:target => 'REG', :mask => '00000000') {
  xor target, zero, zero
}
preparator(:target => 'REG', :mask => 'FFFFFFFF') {
  nor target, zero, zero
}
```

Code inside the *preparator* block uses the *target* and *value* functions to access the target addressing mode and the value passed to the preparator.

The TTDL provides the *prepare* function to explicitly insert preparators into TPs. It can be used to create composite preparators. The function has the following arguments:

- *target* - specifies the target addressing mode;
- *value* - specifies the value to be written;
- *attrs* (optional) - specifies the preparator name and the variant name to select a specific preparator.

For example, the following line of code places value *0xDEADBEEF* into the *t0* register:

```
prepare t0, 0xDEADBEEF
```

4.12 Describing Self-Checks

TPs can include self-checks that check validity of the microprocessor state after a test case has been executed. These checks are instruction sequences inserted in the end of test cases which compare values stored in registers with expected values. If the values do not match control is transferred to a handler that reports an error. Expected values are produced by the MicroTESK simulator. Self-check are described using the *comparator* construct which has the same features as the *preparator* construct, but serves a different purpose. Here is an example of a comparator for 32-bit registers and its special case for value equal to *0x00000000*:

```
comparator(:target => 'REG') {
  prepare target, value
  bne at, target, :check_failed
  nop
}
comparator(:target => 'REG', :mask => "00000000") {
  bne zero, target, :check_failed
  nop
}
```

4.13 Describing Test Cases

A TP can be described by the following formula: $\Pi = \Pi_{start} \cdot \{\langle \pi_{start}, x_i, \pi_{stop} \rangle\}_{i=1,n} \cdot \Pi_{stop}$, where:

- Π_{start} is a TP prologue that consists of instructions aimed for microprocessor initialization;
- $\langle \pi_{start}, x_i, \pi_{stop} \rangle$ is a test case that specifies an individual stimulus and consists of:
  - π_{start} is a test case prologue that performs all necessary preparations for the test case;

- x_i is a test case action that contains the main code of the test case;
- π_{stop} is a test case epilogue that performs finalization actions for the test case such as self-checks.

- Π_{stop} is a TP epilogue that consists of instructions aimed for microprocessor finalization;
- *n* is the number of test cases in a TP.

The TTDL provides means of describing each part of a TP. Π_{start} and Π_{stop} are described in the *pre* and *post* methods of a TT class correspondingly. Test cases are specified in the *run* method.

Test cases are described by block constructs specifying one or more sequences of instruction calls. Each sequence is a separate test case. It is possible to process a block multiple times. This makes sense when sequences use randomization. In this case, it results different test cases based on the same description. For example, the code below describes five test cases based on the same sequence of three calls. Input data for the calls are generated at random and will be different for all test cases.

```
def run
  sequence {
    add t0, t1, t2
    sub t3, t4, t5
    or t6, t7, t8
  }.run 5
end
```

π_{start} that contains preparators for input registers and π_{stop} that contains self-checks will be generated by the tool automatically. Also, it is possible to specify additional prologue and epilogue for test cases. They will be inserted between automatically generated prologue and epilogue and main code of the test cases. They are specified using the *prologue* and *epilogue* blocks nested into the sequence block. The syntax looks like this:

```
sequence {
  prologue { ... }
  ...
  epilogue { ... }
}.run n
```

When instruction sequences are merged by nesting blocks, prologue and epilogue of nested blocks wrap sequences returned by these blocks.

Test cases can be processed by different TPG engines. A specific engine can be selected by passing the *engine* parameter to the block construct that describes the test cases.

4.14 Describing Exception Handlers

TPs must contain handlers of exceptions that may occur during their execution. Exception handlers are described using the *exception_handler* construct. This

description is also used by the MicroTESK simulator to handle exceptions. Separate exception handlers are described using the *section* construct nested into the *exception_handler* block. The *section* function has two arguments: *org* that specifies the handler's location in memory and *exception* that specifies names of associated exceptions. For example, the code below describes a handler for the *IntegerOverflow*, *SystemCall* and *Breakpoint* exceptions, which resumes execution from the next instruction:

```
exception_handler {
  section(:org =>0x380, :exception => ['IntegerOverflow', 'SystemCall', 'Breakpoint']) {
    mfc0 ra, cop0(14)
    addi ra, ra, 4
    jr ra
    nop
  }
}
```

4.15 Printing Text

TPs are printed in textual form to source code files. The printed text includes various supplementary messages such as comments and separators. They are generated by MicroTESK engines or specified by users in TTs. The format of printed text is set up using the following settings:

- *sl_comment_starts_with* - starting characters for single-line comments. Default value is "//".
- *ml_comment_starts_with* - starting characters for multi-line comments. Default value is "/*".
- *ml_comment_ends_with* - terminating characters for multi-line comments. Default value is "*/".
- *indent_token* - indentation token. Default value is "\t".
- *separator_token* - token used in separator lines. Default value is "=".

The settings are initialized with default values in the *initialize* method of the *Template* class can be redefined in the *initialize* method of a TT.

The TTDL provides functions for printing custom text messages. Text messages are printed either into the generated source code or into the simulator log. Here is the list of supported functions:

- *newline* - adds the new line character into the TP;
- *text* - adds text into the TP;
- *trace* - prints text into the simulator execution log;
- *comment* - adds a comment into the TP;
- *start_comment* - starts a multi-line comment;
- *end_comment* - ends a multi-line comment.

The *text*, *trace* and *comment* functions print formatted text. They take a format string and an array of objects to be printed, which can be constants or memory locations. To specify locations to be printed (registers, memory), the *location* function should be used. It takes the name of the memory array and the index of the selected element. For example, the code below prints a constant value and a value stored in a register in the hexadecimal format:

```
text 'Constant: 0x%X', 0xDEADBEEF
text 'Register: 0x%X', location('GPR', 8)
```

5. Case Study

MicroTESK and its TTDL have been applied in industrial projects to generate TPs for MIPS64 [13] and ARMv8 [14] microprocessors. Table 1 provides characteristics of the MIPS64 and ARMv8 specifications used to configure MicroTESK for generating TPs for these designs.

Table 1. Industrial application of the proposed TTDL and supporting tool

Project	**MIPS64**	**ARMv8**
Number of instructions	102	207
ISA specification size (lines of code)	70	143
MMU specification size (lines of code)	134	637
Efforts (person-months)	101	809

Created tests include:

- tests for arithmetical instructions;
- tests for floating-point instructions;
- tests for branch instructions;
- tests for memory access instructions.

To describe tests for branch and memory instruction, the TTDL was extended with additional constructs based on existing ones. The language was evolving in the process of working on the projects. Some features were changed and some were added. A number of language features came as requirements from customers. The approach based on using dynamic languages such as Ruby to create TTDLs has proved its flexibility. The TTDL allowed describing test cases in a format which is maximally close to assembly language for corresponding microprocessors. This allows verification engineers to concentrate on verification problems instead of issues related to the use of a specific programming language.

5. Conclusion

A concept of a TTDL for a reconfigurable and extensible TPG framework has been considered. The proposed solution was implemented in the MicroTESK [3] framework. The developed TTDL is based on the Ruby [6] language and uses its metaprogramming facilities to dynamically add configuration-dependent language

constructs. The language is integrated into MicroTESK, which is a Java-based tool, with the help of JRuby [11]. Facilities of the TTDL can be extended by adding new Ruby libraries.

Directions for further research and development are to apply the described principles to create TTDLs based on other programming languages. First of all, it is Python and its Java-based implementation called Jython. It provides facilities similar to those of Ruby and is also popular among verification engineers. For this reason, it would be advantageous to provide a Python-based version of the TTDL for those who are more comfortable with this language.

Another task is development of a TTDL based on C++. It will be a part of a large research project dedicated to *on-line* generation. An on-line TPG tool is represented by a binary image with basic functions of an operating system, which is loaded directly to a microprocessor chip where it generates and executes test stimuli. The tool will be created by MicroTESK from C++ libraries based on formal specifications. For further unification of TPG tools, it is important that TTs for on-line generation are developed using the same principles.

References

[1]. A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. Design & Test of Computers, 2004. pp. 84–93.

[2]. Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus and G. Shurek. Constraint-Based Random Stimuli Generation for Hardware Verification. AI Magazine, Volume 28, Number 3, 2007, pp. 13–30.

[3]. MicroTESK page. http://forge.ispras.ru/projects/microtesk

[4]. A. Kamkin, E. Kornykhin, D. Vorobyev. Reconfigurable Model-Based Test Program Generator for Microprocessors. International Conference on Software Testing, Verification and Validation Workshops, 2011. pp. 47–54.

[5]. A.S. Kamkin, T.I. Sergeeva, S.A. Smolov, A.D. Tatarnikov, M.M. Chupilko. Extensible Environment for Test Program Generation for Microprocessors. Programming and Computer Software, 40(1), 2014. pp. 1-9.

[6]. Ruby site: http://www.ruby-lang.org

[7]. E.A. Poe. Introduction to Random Test Generation for Processor Verification. Obsidian Software, 7 pp, 2002.

[8]. RAVEN test program generator. Available at: http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc

[9]. Seonghun Jeong, Youngchul Cho, Daeyong Shin, Changyeon Jo, Yenjo Han, Soojung Ryu, Jeongwook Kim, and Bernhard Egger. Random Test Program Generation for Reconfigurable Architectures. 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, 6 p.

[10]. A. Kamkin. Test Program Generation for Microprocessors. Trudy ISP RAN / Proc. ISP RAS], vol. 14, part 2, 2008, pp. 23-64 (in Russian).

[11]. JRuby site: http://www.jruby.org

[12]. Flanagan D., Matsumoto Y. The Ruby Programming Language. OReilly Media, Sebastopol, 2008.
[13]. MIPS64TM Architecture For Programmers. Volume II: The MIPS64TM Instruction Set, Document Number: MD00087, Revision 2.00, June 9, 2003.
[14]. ARM Architecture Reference Manual. ARM DDI 0487A.f, ARM Corporation, 2015. 5886 p.

Язык описания шаблонов для генерации тестовых программ для микропроцессоров

А.Д. Татарников <andrewt@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Генерация тестовых программ на языке ассемблера и проверка корректности результатов их выполнения является наиболее широко применяемым подходом к функциональной верификации микропроцессоров. Данная задача решается при помощи специальных автоматизированных средств, называемых генераторами тестовых программ. Высокая сложность современных электронных устройств создает потребность в автоматизированных средствах, способных генерировать тестовые программы, покрывающие нетривиальные ситуации в их работе. Большинство таких средств используют в качестве входных данных шаблоны тестовых программ, которые позволяют описывать тестовые сценарии в абстрактном виде. Такой подход предоставляет инженерам-верификаторам возможность описывать широкий спектр задач генерации, затрачивая минимальные усилия. Шаблоны тестовых программ разрабатываются на специальных предметно-ориентированных языках. Такие языки должны удовлетворять следующим требованиям: (1) они должны быть достаточно простыми для использования инженерами-верификаторами, не обладающими серьезными навыками программирования; (2) они должны быть применимы для широкого спектра микропроцессорных архитектур и (3) они должны быть легко расширяемы для поддержки описания новых типов задач генерации. В данной работе рассматривается язык описания шаблонов тестовых программ, который был создан для расширяемой среды генерации тестовых программ MicroTESK, разрабатываемой в ИСП РАН. Это гибкий предметно-ориентированный язык, основанный на языке Ruby, который позволяет описывать широкий набор задач генерации в терминах абстракций цифровой аппаратуры. Среда генерации MicroTESK и язык описания тестовых шаблонов успешно применяются в промышленных проектах по верификации микропроцессоров на базе архитектур MIPS и ARM.

Ключевые слова: микропроцессоры; функциональная верификация; генерация тестовых программ; тестовые шаблоны; предметно-ориентированные языки.

DOI: 10.15514/ISPRAS-2016-28(4)-5

Для цитирования: Татарников А.Д. Язык описания шаблонов для генерации тестовых программ для микропроцессоров. Труды ИСП РАН, том 28, вып. 4, 2016, стр. 77-98. DOI: 10.15514/ISPRAS-2016-28(4)-5

Список литературы

[1]. A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. Design & Test of Computers, 2004. pp. 84–93.

[2]. Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus and G. Shurek. Constraint-Based Random Stimuli Generation for Hardware Verification. AI Magazine, Volume 28, Number 3, 2007, pp. 13–30.

[3]. Инструмента MicroTESK. http://forge.ispras.ru/projects/microtesk

[4]. A. Kamkin, E. Kornykhin, D. Vorobyev. Reconfigurable Model-Based Test Program Generator for Microprocessors. International Conference on Software Testing, Verification and Validation Workshops, 2011. pp. 47–54.

[5]. Камкин А.С., Сергеева Т.И., Смолов С.А., Татарников А.Д., Чупилко М.М. Расширяемая среда генерации тестовых программ для микропроцессоров. Программирование, № 1, 2014, стр. 3-14.

[6]. Язык Ruby: http://www.ruby-lang.org.

[7]. E.A. Poe. Introduction to Random Test Generation for Processor Verification. Obsidian Software, 7 pp, 2002.

[8]. Инструмент RAVEN: http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc.

[9]. Seonghun Jeong, Youngchul Cho, Daeyong Shin, Changyeon Jo, Yenjo Han, Soojung Ryu, Jeongwook Kim, and Bernhard Egger. Random Test Program Generation for Reconfigurable Architectures. 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, 6 p.

[10]. А.С. Камкин. Генерация тестовых программ для микропроцессоров. Труды ИСП РАН, 14(2), 2008. С. 23-63.

[11]. Интерпретатор JRuby: http://www.jruby.org.

[12]. Flanagan D., Matsumoto Y. The Ruby Programming Language. OReilly Media, Sebastopol, 2008.

[13]. MIPS64TM Architecture For Programmers. Volume II: The MIPS64TM Instruction Set, Document Number: MD00087, Revision 2.00, June 9, 2003.

[14]. ARM Architecture Reference Manual. ARM DDI 0487A.f, ARM Corporation, 2015. 5886 p.

Specification-Based Test Program Generation for MIPS64 Memory Management Units

A.S. Kamkin <kamkin@ispras.ru>
A.M. Kotsynyak <kotsynyak@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. In this paper, a tool for automatically generating test programs for MIPS64 memory management units is described. The solution is based on the MicroTESK framework being developed at the Institute for System Programming of the Russian Academy of Sciences. The tool consists of two parts: an architecture-independent test program generation core and MIPS64 memory subsystem specifications. Such separation is not a new principle in the area: it is applied in a number of industrial test program generators, including IBM's Genesys-Pro. The main distinction is in how specifications are represented, what sort of information is extracted from them, and how that information is exploited. In the suggested approach, specifications comprise descriptions of the memory access instructions, loads and stores, and definition of the memory management mechanisms such as translation lookaside buffers, page tables, table lookup units, and caches. A dedicated problem-oriented language, called MMUSL, is used for the task. The tool analyzes the MMUSL specifications and extracts all possible instruction execution paths as well as all possible inter-path dependencies. The extracted information is used to systematically enumerate test programs for a given user-defined test template and allows exhaustively exercising co-execution of the template instructions, including corner cases. Test data for a particular program are generated by using symbolic execution and constraint solving techniques.

Keywords: microprocessor, memory management unit, caching, address translation, formal specification, test program, test program generation, MIPS64.

DOI: 10.15514/ISPRAS-2016-28(4)-6

For citation: Kamkin A.S., Kotsynyak A.M. Specification-Based Test Program Generation for MIPS64 Memory Management Units. *Trudy ISP RAN /Proc. ISP RAS*, 2016, vol. 28, issue 4, pp. 99-114. DOI: 10.15514/ISPRAS-2016-28(4)-6

1. Introduction

A computer memory is known to be a complex hierarchy of data storage devices varying in volume, latency and price. In addition to registers and main memory,

microprocessors include a multi-level cache memory and address translation buffers. The set of devices responsible for handling memory accesses is referred to as a *memory subsystem* or a *memory management unit* (*MMU*). Being one of the key microprocessor components, the memory subsystem is strongly required to be correct and reliable. Due to the complicated structure of the memory, the number of situations that can occur in processing load and store instructions is huge; this makes it improbable to verify the subsystem «manually».

It is widely accepted that *test program generation* (*TPG*) is an essential approach to microprocessor verification [3]. The problem is how to overcome the complexity and at the same time provide acceptable test coverage. It is a fallacy that (naive) random TPG is a good way to optimize testing [4]. A better solution, we think, is a *specification-based approach* [3]. A TPG tool consists of two components: (1) an architecture-independent test generation *core* and (2) an architecture *specification*, or *model*. The approach reduces the efforts to create a generator by reusing the core – the only thing one needs to develop is a specification.

There exist a number of tools implementing the paradigm mentioned above [3], [5], [6]. However, only few of them are distributed under open licenses. ISP RAS's MicroTESK [7] is one of those few. The tool uses a dialect of the nML language[8] for specifying *instruction set architectures* (*ISA*) and an extensible set of dedicated languages for specifying particular microarchitectural features, including, first of all, a memory management. In this work, we would like to share our experience in creating a MicroTESK-based TPG for verifying MIPS64 MMUs [1], [2].

The remainder of this paper is divided into four sections. Section 2 contains a brief survey of the existing approaches to TPG for MMUs. Section 3 presents the MicroTESK framework and its facilities aimed at MMU specification and testing. Section 4 studies application of the TPG approach for MIPS64 MMU. Section 5 discusses the results of the work and outlines directions of future research and development.

2. Related work

There are several TPG tools based on formal specifications of memory subsystems. IBM's DeepTrans [9] uses a dedicated specification language. Address translation is depicted as a *directed acyclic graph* (*DAG*) whose vertices correspond to the process stages and whose edges relate to the transitions between the stages. A path from the source of the DAG to the sink defines a particular *situation* in the address translation. Such situations can be referred from high-level descriptions of *test programs* (*TPs*), so-called *test templates* (*TTs*). The latter are processed by Genesys-Pro [3], which formulates constraints on instruction operands, solves them and transforms the solutions into the instruction sequences. The major advantage of the approach is the use of the highly developed languages for modeling MMUs and describing TTs. A possible disadvantage is that the tool seems not to be able to automatically extract MMU-related *dependencies* between instructions.

In [10], the Java language coupled with a special library is used to model MMUs. As in DeepTrans, the situations correspond to the paths in the DAG describing the MMU. For example, {*TLB(va).hit, TLB(va).entry.V, ¬L1(pa).hit*}: there is a hit in the *translation lookaside buffer* (*TLB*); the matched entry is valid; there occurs a miss in the *first-level cache* (*L1*). In addition, the approach provides facilities for specifying MMU-related dependencies between instructions. For example, {*TLB* ↦ ¬*tagEqual*, *L1* ↦ *indexEqual*}: instructions access different TLB entries; data are mapped onto the same set of L1. TTs are constructed automatically by combining situations and dependencies for short sequences of instructions. Building TTs and creating TPs is done by MicroTESK (version 1) [7]. The strength of the approach is systematic TT enumeration that takes into consideration instruction execution paths as well as dependencies between instructions. The principal weakness is underdeveloped specification facilities.

3. MicroTESK Framework

MicroTESK (version 2.3 or higher) [11] combines the advantages of the approaches presented in [9] and [10]. The tool inputs are ISA specifications in nML [8], MMU specifications in MMUSL (MMU Specification Language) and TTs in Ruby [12]. The basic principles of MicroTESK are close to ones implemented in Genesys-Pro [3]. The specifications are analyzed to extract *testing knowledge* (situations and dependencies), which is used to generate TPs from the given TTs as well as to systematically enumerate TTs. More information on the tool can be found in [13] and [14]. Here we provide a brief introduction to MicroTESK by the example of an MIPS64 MMU [2].

3.1 ISA Specifications

ISA specifications include definitions of *data types*, *constants*, *registers*, *access modes*, *memories* and *instructions*. Here comes an example (a fragment of the MIPS64 specification), where there are listed three data types, BYTE, SHORT and DWORD.

```
type BYTE = card(8)       // unsigned 8-bit vectors
type SHORT = int(16)      // signed 16-bit vectors
type DWORD = card(64) // unsigned 64-bit vectors
```

Registers of the same type are grouped into arrays. Register access logic is encapsulated in so-called *modes*, which, besides other things, define *assembly format* (**syntax**) and *binary encoding* (**image**) of the registers. The following example declares an array GPR, consisting of thirty two 64-bit registers, designates a *stack pointer* alias SP = GPR[29], and defines a mode REG aimed at accessing those registers.

```
reg GPR [32, DWORD]              // Array of 32 DWORD Registers
reg SP[DWORD] alias = GPR[29] // Stack Pointer Alias

mode REG (i: card(5)) = GPR[i]   // One-to-One Mapping
  syntax = format("r%d", i)      // Assembly Format
  image = format("%5s", i)       // Binary Encoding
  number = i                     // Custom Attribute
```

Like a group of registers, a memory unit is represented as a plain array. In the example below, an array MEM is interpreted as a *physical memory* comprised of 2^{36} bytes. *Virtual memory* issues such as address translation, caching, and the like are specified separately with the use of a dedicated language (see the next section).

```
mem MEM[2 ** 36, BYTE] // Physical Memory Array
```

The attributes of instructions include **syntax**, **image** and **action**. Actions of load and store instructions are described in an intuitive manner by reading or writing data from or to the array representing the physical memory. Here is a specification of the *Load Byte* instruction (LB), which derives an address from a base register (base) with given offset (offset), loads a byte from the memory, and writes it to a register (rt).

```
op LB (rt: REG, offset: SHORT, base: REG)
  syntax = format("lb %s, %d(%s)", rt.syntax, offset, base.syntax)
  image = format("100000%5s%5s%16s", base.image, rt.image, offset.image)
  action = { rt = MEM[base + offset]; }
```

Notwithstanding MEM is interpreted as the physical memory, it is accessed through virtual addresses – an access triggers the address translation mechanisms and other MMU logic.

3.2 MMU Specifications

Being rather simple, nML does not have adequate facilities to describe MMUs. For this purpose, a special MMUSL language is used. MMU specifications include *address types*, *memory segments*, *buffers*, and *control logic* for handling loads and stores. In the following example, address type, VA, is declared. It is a structure with single field – address itself.

```
address VA(vaddress : 64)
```

A memory segment is considered as a mapping from a set of addresses of some type to a set of addresses of another type. An example given below defines a segment

XKPHYS that maps a VA of the given set (**range**) to the physical address (PA). The segment performs flat translation with no use of TLBs and tables (**read**).

```
segment XKPHYS (va: VA) = (pa: PA)
  range = (0x8000000000000000, 0xbfffFFFFffffFFFF)
  read = {
    pa.paddress = va.vaddress<35..0>;
    pa.cca = va.vaddress<61..59>;
  }
```

Buffers (TLBs, cache units, page tables, etc.) are specified with the following parameters: the *associativity* (**ways**), the *number of sets* (**sets**), the *entry format* (**entry**), the *index calculation function* (**index**), the *tag calculation function* (**tag**) and the *data eviction policy* (**policy**). Their meaning passes current among microprocessors designers. Here comes a sample description of TLB. It is accessed by VAs. The keyword **register** means that the buffer is *mapped to the registers* and can be accessed from the ISA specifications.

```
register buffer TLB (va: VA)
  sets = 1 // Fully associative buffer
  ways = 64
  entry = (R: 2, VPN2: 27, ASID: 8, PageMask: 16, G: 1, …)
  tag = va<39..13>
```

Processing of memory access instructions is specified by requesting the segments and buffers. The syntax is similar to nML though allows using such constructs as B(A).hit (the buffer B contains an entry for the address A), E=B(A) (the entry for the address A is read from the buffer B and assigned to E), B(A)=E (the entry E for the address A is written to the buffer B), and the like. Here is a fragment of the MIPS64 MMU specification. It contains two attributes, **read** and **write**, which, respectively, define logic of loads and stores.

```
mmu MMU (va: VA) = (data: DATA_SIZE)
  var pa: PA;
  var line: DATA_SIZE;
  var l1Entry: L1.entry;
  read = {
    pa = TranslateAddress(va); // Address Translation
```

```
  if IsCached(pa.cca) == 1 then
   if L1(pa).hit then // L1 Cache Access
    l1Entry = L1(pa);
    line = l1Entry.DATA;
   else
    line = MEM(pa);
    l1Entry.TAG = pa.paddress<...>; // L1 Cache Update
    l1Entry.DATA = line;
    L1(pa) = l1Entry;
   endif;
  else
   line = MEM(pa);
  endif;
  data = line;
 }
 write = { … }
```

3.3 TPG Approach

The MicroTESK TPG approach is based on TTs written in Ruby [12]. In general terms, the process is as follows [14]. A TT describing a microprocessor verification scenario is given to MicroTESK. The tool processes the TT and builds a series of *symbolic TPs*, where abstract *situations* and *dependencies* (often in the form of *constraints*) are used instead of specific values. Each symbolic TP is instantiated with appropriate *test data* (*TD*). The resultant TP is supplemented with *preparation code* that initializes the registers, the buffers, and the memory.

TTs are allowed to use modes and instructions defined in the specifications as well as special TPG constructs (*blocks*, *situations*, etc) [14]. More technically, a TT is a subclass of the Template base class provided by the MicroTESK library. In the example below, MmuTemplate is a subclass of Mips64BaseTemplate, which, in turn, is a subclass of Template. The entry point is method **run**. This method declares a *block* of two instructions, LD and SD, to be processed with the dedicated *memory engine*. The situation **access** guides TPG by specifying constraints and biases for the MMU variables and buffers. The denotation **reg(_)** means any instance of the mode REG, i.e. any GPR.

```
class MmuTemplate < Mips64BaseTemplate
 def run
  block(:engine => "memory", ...) {
   ld reg(_), 0x0, reg(_)
    do situation("access", hit("L1"), ...) end
```

```
    sd reg(_), 0x0, reg(_)
   }
  end
end
```

Let us consider how TPG for MMUs is organized. Parsing specifications results in two entities: an *interpreter*, which is a part of the *instruction set simulator* (*ISS*), and a *symbolic representation* in the form of a labeled DAG. The DAG is traversed, and all possible *execution paths* are extracted. An execution path describes processing of a single memory request and finishes either with a *memory access* or with an *exception* (alignment fault, TLB refill event, etc.). Paths are composed of transitions. Each transition is supplied with a *guard*, i.e. a condition that enables the transition, and an *action* to be performed; it can also be labeled with a *buffer* being used in the guarded action. Here is a fragment of the execution path in MMU (see above) represented in a hypothetical language.

```
path PATH(va: VA) = (data: DATA_SIZE)
 transition {
  guard = TRUE
  action = {} // Go to TranslateAddress(va)
 } …
 transition {
  guard = L1(pa).hit
  action = { l1Entry = L1(pa); line = l1Entry.DATA; }
  buffer = L1
 } …
```

Given two execution paths, the tool can extract possible *dependencies* between them. A dependency is a map from the set of buffers common for the given paths to the set of *conflict types*. More formally, let p_1 and p_2 be execution paths, C be a non-empty set of conflict types, and $B(p)$ be the set of buffers used in a path p. A dependency between p_1 and p_2 is a map $d: B(p_1) \cap B(p_2) \to C$. The set C is supposed to include the following elements and their negations:

- *indexEqual* – access to the same set of the buffer;
- *tagEqual* – access to the same entry of the buffer;
- *tagEvicted* – access to the recently evicted entry.

Given a TT, symbolic TPs are systematically enumerated. The main, but not the only, approach supported by MicroTESK is *combinatorial generation*. Symbolic TPs are constructed by selecting all relevant execution paths for the TT's instructions and producing all satisfiable dependencies for each combination of the paths. To avoid combinatorial explosion, special *heuristics* are used, including factorization of the

paths and limitation of the depth of the dependencies. Among them, a *buffer-event factorization* is frequently used. Let p be a path, and $event_p : B(p) \to \{ hit, miss \}$ be the induced map of the buffers to the events. Two paths, p_1 and p_2, are *equivalent*, if $B(p_1) = B(p_2)$ and for each $b \in B(p_1)$, $event_{p_1}(b) = event_{p_2}(b)$ holds. During TPG, the equivalence classes are enumerated, while their representatives are randomized.

Symbolic TP is a pair $\langle \{p_i\}_{i=1}^{n}, \{d_{ij}\}_{i,j=1(i<j)}^{n} \rangle$, where p_i is an execution path, and d_{ij} is a dependency between p_i and p_j. To produce a TP from a symbolic TP, appropriate TD are required, including addresses of the instructions, entries of the buffers being accessed (except *replaceable* ones, such as caches) and sequences of addresses to be used to load or evict data to or from the replaceable buffers. Formally, TD are a tuple $\langle \{addr_i\}_{i=1}^{n}, \{entry_i\}_{i=1}^{n}, load, evict \rangle$, where $addr_i(a)$ is an address of the type a used in the path p_i, $entry_i(b$ is an entry of the buffer b accessed by the path p_i, $load(b, s$ is a sequence of addresses to load data to the set s of the buffer b and, finally, $evict(b, s)$ is a sequence of addresses to evict data from the set s of the buffer b.

Here is an approximation of the TD generation algorithm implemented in MicroTESK's *memory engine*. The following denotations are used: : $d_j(b, c)$ is the minimal i, such that $1 \leq i < j$ and $d_{ij}(b) = c$, or a special value $\epsilon \notin N$ if there are no such i; $addr_j(b)$ is equivalent to $addr_j(a_b)$, where a_b is the address type of the buffer b; $tag_b(addr)$ and $index_b(addr)$ are, respectively, the tag and the index extracted from $addr$ by using the corresponding functions of the buffer b; $newAddr_b(tag, index, \dots)$ is an address constructed from tag, $index$, and, probably, some other information; $newEntry_b(id, index)$ is an empty entry of the buffer b with specified id and $index$; given a buffer b, its state s, and $index$, $victim_b(s, index)$ is a tag to be evicted. Other functions will be briefly explained further below.

```
forall j ∈ {1,…,n} do
 addr_j ← Solver.constructAddresses(p_j)
 forall b ∈ B(p_j) do
  if d_j(b, tagEqual) ≠ ϵ then
   i ← d_j(b, tagEqual)
   addr_j(b) ← newAddr_b(tag_b(addr_i(b)), index_b(addr_j(b)),…)
  else if d_j(b, indexEqual) ≠ ϵ then
    i ← d_j(b, indexEqual)
    tag_new ← Allocator.allocTag(b, index_b(addr_i(b)))
    addr_j(b) ← newAddr_b(tag_new, index_b(addr_i(b)),…)
  endif
 endfor

 forall b ∈ B(p_j) do
  if b.policy ≠ none then
```

```
    if event_{p_j}(b) = hit then
     load(b, index) ← load(b, index) · {addr_j(b)}
    else
     forall k ∈ {1, …, b.ways} do
      tag_new ← Allocator.allocTag(b, index_b(addr_j(b)))
      addr_evict ← newAddr_b(tag_new, index_b(addr_j(b)), …)
      evict(b, index) ← evict(b, index) · {addr_evict}
     endfor
    endif
   else
    if event_{p_j}(b) = hit then
     if d_j(b, tagEqual) ≠ ϵ then
      i ← d_j(b, tagEqual)
      entry_j(b) ← entry_i(b)
     else
      id_new ← Allocator.allocEntryId(b, index_b(addr_j(b)))
      entry_{j(b)} ← newEntry_b(id_new, index_b(addr_j(b)))
     endif
    endif
   endif
  endfor
endfor

state ← Interpreter.observeState()
loads ← Loader.prepareLoads(load, evict)
state ← Interpreter.execMmu(loads, state)

forall j ∈ {1, …, n} do
 forall b ∈ B(p_j, a) do
  if d_j(b, tagReplace) ≠ ϵ then
   i ← d_j(b, tagReplace)
   addr_j(b) ← newAddr_b(Tag_evict(b, p_i), index_b(addr_j(b)), …)
  endif
  if event_{p_j}(b) = miss then
   Tag_evict(b, p_j) ← victim_b(state, index_b(addr_j(b)))
  endif
  state ← Interpreter.execBuffer(b, {addr_j(b)}, state)
 endfor
endfor

forall j ∈ {1, …, n} do
 entry_j ← Solver.constructEntries(p_j)
endfor
```

Generator exploits several auxiliary components: *Solver*, *Allocator*, *Interpreter* and *Loader*. *Solver* performs symbolic execution of a given path and constructs required entities (addresses, entries, etc.) by calling constraint solvers. Interface with solvers is provided by Fortress library [15]. It supports *SMT solvers*, such as Z3 [16] and CVC4 [17], as well as *in-house solvers* aimed at particular tasks. *Allocator*

chooses buffer indices, tags and other address fields taking into account user-defined constraints (e.g., forbidden memory regions). The default strategy is to allocate a new index or a new tag for a given index on every request. This allows avoiding undesirable dependencies between instructions. $Interpreter$ simulates accesses to buffers and predicts data evictions. The results of the predictions are used to satisfy $tagEvicted$ conflicts. $Loader$ prepares a sequence of accesses so as to fulfill hit and $miss$ requirements. The default strategy is as follows. Buffers are handled in reverse order; for every buffer b and every set s, $evict(b, s)$ and $load(b, s)$ are added to the sequence.

Finally, TD are transformed to the ISA-specific preparation code. For this job, the tool needs to know what instructions have to be used to set up addresses and entries. Such information is provided in TTs in the form of so-called *preparators*. Technically, a preparator is a piece of code that defines a sequence of instructions to reach a certain goal. Given a register type (to be more precise, an access mode), there usually exists a family of preparators differing in patterns of loaded values. For example, Mips64BaseTemplate contains the following preparator for loading a 32-bit value into a GPR via the mode REG.

```
preparator( :target => "REG", :mask => "00000000xxxxxxxx") {
  ori target, target, value(16, 31)
  dsll target, target, 16
  ori target, zero, value(0, 15)
}
```

For each buffer, there should be a preparator to write an entry into it. A preparator for MIPS64 DTLB is given below.

```
buffer_preparator(:target => "DTLB") {
  ori t0, zero, address(48, 63)
  dsll t0, t0, 16
  ori t0, t0, address(32, 47)
  dsll t0, t0, 16
  ori t0, t0, address(16, 31)
  dsll t0, t0, 16
  ori t0, t0, address(0, 15)
  lb t0, 0, t0
}
```

4. MIPS64 MMU Case Study

The most challenging part of developing a specification-based TPG tool for a microprocessor is MMU specification. Speaking of MIPS64, the following things have been specified [2]: address spaces, a TLB entry format, and an address translation procedure. Additionally, we have described a two-level write-through cache memory.

MicroTESK's MMUSL has allowed specifying MIPS64 MMU in quite a compact way (approximately 220 lines of code). The specifications involve a TLB (JTLB and DTLB), two-level cache memory buffers (L1 and L2) and memory segments (kseg0, kseg1, xkphys, and useg). On the base of the ISA [18] and MMU [2] specifications, 18 memory access instructions and several auxiliary instructions to access the TLB and the cache have been defined. Description of a single instruction makes up approximately 10 lines of nML code on average.

Table 1. Complexity of MIPS64 MMU Specification

	Min	**Max**	**Average**
Number of Transitions if an Execution Path	7	52	38
Number of Variables in a Path Formula	3	76	49
Number of Execution Paths of an Instruction	76		

Table 1 contains numeric data on MIPS64 MMU execution path complexity. While the complexity is relatively low (average path consists of less than 40 transitions and comprises less than 50 variables), only very short TTs can be processed by exhaustive enumeration of symbolic TPs. In more complicated cases, heuristics become of crucial importance. E.g., the buffer-event factorization gives only 9 path equivalence classes, enabling systematic enumeration of longer sequences of memory accesses. Generation of more complicated TPs is done with the help of constrained random generation. This requires verification engineers to explicate their knowledge in the form of constraints and biases.

This is an ongoing project, and some useful information, such as test coverage, is not available at the moment. Though it is worth considering the lessons learned. We found it convenient to use domain-specific languages (DSLs) for specifying ISAs and MMUs. The use of DSLs, first, eases extraction of testing knowledge and, second, simplifies learning of the TPG tool. On the other hand, it seems that dynamic programming languages, such as Ruby and Python, suit well for describing TTs. Such languages can be easily extended with TPG constructs. Our negative experience is mostly connected with low performance of the tool. Constraint solving needs to be optimized. As the authors of [19], we believe that specialized solvers will help.

5. Conclusion

TPG is a widely-accepted approach to microprocessor verification, including, in particular, MMU verification. State-of-the-art MMUs are extremely complex devices comprising multi-level address translation and caching. Naive approaches to automated TPG for MMUs – meaning, first of all, random generation techniques – are highly improbable to reach high level of test coverage in reasonable time. Specification-based TPG, in our opinion, is one of the most promising directions in the area. Since 1990s, it has been successfully applied to microprocessor testing and verification, e.g., in IBM [3], and it continues to evolve.

The MicroTESK team [7] contributes its mite to the evolution of the specification-based approach. Our goal is to create an open-source, extensible and reconfigurable TPG framework [13], [14]. Different versions of MicroTESK, including the one described in [10], have been applied to several industrial microprocessors and allowed to reveal a large number of critical bugs, which had not been detected by randomly generated TPs.

The proposed solution is based on ISA specifications in nML [8] and MMU specifications in MMUSL. ISA specifications formally describe microprocessor instructions, while MMU specifications define memory segments and buffers. MicroTESK is able to automatically extract testing knowledge from the specifications and to exploit it for TPG. TTs are created with the help of Ruby [12]. To generate TD, symbolic execution and constraint solving techniques are intensively used.

The work is still in progress, and a number of things need to be done. The most priority task is a performance optimization of the constraint solving. Another task is to extend the approach to multicore designs and multiprocessor systems. The main challenge here is to create a unified technology that would include formal verification of cache coherence protocols, unit-level verification of MMUs, and system-level TPG.

References

[1]. MIPS64™ Architecture For Programmers. Volume 1: Introduction to the MIPS64™ Architecture. Revision 6.01. MIPS Technologies Inc. 2014. 148 p.

[2]. MIPS64™ Architecture For Programmers. Volume 3: MIPS64™/microMIPS64™ Privileged Resource Architecture. Revision 6.03. MIPS Technologies Inc. 2015. 368 p.

[3]. A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. Design & Test of Computers, 2004. pp. 84-93.

[4]. R.L. Glass. Facts and Fallacies of Software Engineering. Addison-Wesley Professional, 2002. 224 p.

[5]. T. Li, D. Zhu, Y. Guo, G. Liu, S. Li. MA2TG: A Functional Test Program Generator for Microprocessor Verification. Euromicro Conference on Digital System Design, 2005. pp. 176-183.

[6]. A. Kamkin, A. Tatarnikov. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. Spring/Summer Young Researchers Colloquium on Software Engineering, 2012, pp. 64-69.

[7]. MicroTESK tool. http://forge.ispras.ru/projects/microtesk
[8]. M. Freericks. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993.
[9]. A. Adir, L. Fournier, Y. Katz, A. Koyfman. DeepTrans – Extending the Model-based Approach to Functional Verification of Address Translation Mechanisms High-Level Design Validation and Test Workshop, 2006. pp. 102-110.
[10]. D. Vorobyev, A. Kamkin. [Test program generation for memory management units of microprocesssors]. Trudy ISP RAN /Proc. ISP RAS, vol. 17, 2009. pp. 119-132 (in Russian).
[11]. A. Kamkin, A. Protsenko, A. Tatarnikov. An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation Mechanisms Trudy ISP RAN, 27(3), 2015. pp. 125-138.
[12]. Ruby programming language. http://www.ruby-lang.org
[13]. A. Kamkin, E. Kornykhin, D. Vorobyev. Reconfigurable Model-Based Test Program Generator for Microprocessors International Conference on Software Testing, Verification and Validation Workshops, 2011. pp. 47-54.
[14]. A.S. Kamkin, T.I. Sergeeva, S.A. Smolov, A.D. Tatarnikov, M.M. Chupilko. Extensible Environment for Test Program Generation for Microprocessors. Programming and Computer Software, 40(1), 2014, pp. 1-9.
[15]. Fortress library. http://forge.ispras.ru/projects/solver-api
[16]. Z3 SMT solver. http://github.com/Z3Prover/z3
[17]. CVC4 SMT solver. http://cvc4.cs.nyu.edu
[18]. MIPS64™ Architecture For Programmers. Volume 2: The MIPS64™ Instruction Set Reference Manual. Revision 6.04. MIPS Technologies Inc. 2015. 551 p.
[19]. Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek. Constraint-Based Random Stimuli Generation for Hardware Verification AI Magazine, 28(3), 2007. pp. 13-30.

Генерация тестовых программ для подсистемы управления памятью MIPS64 на основе спецификаций

А.С. Камкин <kamkin@ispras.ru>
А.М. Коцыняк <kotsynyak@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В данной работе описан инструмент автоматической генерации тестовых программ для подсистем управления памятью микропроцессоров с архитектурой MIPS64. Предлагаемое средство базируется на среде MicroTESK, разрабатываемой в Институте системного программирования РАН. Инструмент состоит из двух частей: архитектурно независимого ядра генерации тестовых программ и спецификации подсистемы памяти MIPS64. Такое разделение не является новым — аналогичный подход применяется в промышленных генераторах, в том числе в Genesys-Pro, разрабатываемом в исследовательском подразделении компании IBM. Основные различия между инструментами состоят в форме представления спецификаций, типе

извлекаемой из них информации и способах использования этой информации для построения тестов. В предлагаемом подходе спецификации включают в себя описания инструкций доступа к памяти (инструкций чтения и записи) и описания механизмов управления памятью, таких как буфер трансляции адресов, таблица страниц, устройство аппаратного поиска по таблице страниц, кэш-память. Для спецификации такого рода механизмов (устройств) разработан проблемно-ориентированный язык, названный MMUSL. Инструмент анализирует MMUSL-спецификации и извлекает все возможные пути исполнения инструкций (варианты обработки запросов к подсистеме памяти) и все возможные зависимости между этими путями (конфликты использования устройств). Извлеченная информация используется для систематического перебора тестовых программ для заданного пользователем тестового шаблона и позволяет исчерпывающим образом исследовать совместное исполнение группы инструкций, включая разного рода граничные случаи. Тестовые данные для тестовых программ (значения адресов, содержимое буферов и т.п.) генерируются с использованием техник символического исполнения и решения ограничений.

Ключевые слова: микропроцессор, подсистема памяти, кэширование, трансляция адресов, формальная спецификация, тестовая программа, генератор тестовых программ, MIPS64.

DOI: 10.15514/ISPRAS-2016-28(4)-6

Для цитирования: Камкин А.С., Коцыняк А.М. Генерация тестовых программ для подсистемы управления памятью MIPS65 на основе спецификаций. *Труды ИСП РАН*, 2016, том 28, вып. 4, с. 99-114 (на английском). DOI: 10.15514/ISPRAS-2016-28(4)-6

Список литературы

[1]. MIPS64™ Architecture For Programmers. Volume 1: Introduction to the MIPS64™ Architecture. Revision 6.01. MIPS Technologies Inc. 2014. 148 p.
[2]. MIPS64™ Architecture For Programmers. Volume 3: MIPS64™/microMIPS64™ Privileged Resource Architecture. Revision 6.03. MIPS Technologies Inc. 2015. 368 p.
[3]. A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. Design & Test of Computers, 2004. pp. 84-93.
[4]. R.L. Glass. Facts and Fallacies of Software Engineering. Addison-Wesley Professional, 2002. 224 p.
[5]. T. Li, D. Zhu, Y. Guo, G. Liu, S. Li. MA2TG: A Functional Test Program Generator for Microprocessor Verification. Euromicro Conference on Digital System Design, 2005. pp. 176-183.
[6]. A. Kamkin, A. Tatarnikov. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. Spring/Summer Young Researchers Colloquium on Software Engineering, 2012, pp. 64-69.
[7]. Инструмент MicroTESK. http://forge.ispras.ru/projects/microtesk
[8]. M. Freericks. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993.

[9]. A. Adir, L. Fournier, Y. Katz, A. Koyfman. DeepTrans – Extending the Model-based Approach to Functional Verification of Address Translation Mechanisms High-Level Design Validation and Test Workshop, 2006. pp. 102-110.
[10]. Д. Воробьев, А. Камкин. Генерация тестовых программ для подсистемы управления памятью микропроцессора. Труды ИСП РАН, 17, 2009, с. 119-132
[11]. A. Kamkin, A. Protsenko, A. Tatarnikov. An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation Mechanisms Trudy ISP RAN, 27(3), 2015. pp. 125–138.
[12]. Язык программирования Ruby. http://www.ruby-lang.org
[13]. A. Kamkin, E. Kornykhin, D. Vorobyev. Reconfigurable Model-Based Test Program Generator for Microprocessors International Conference on Software Testing, Verification and Validation Workshops, 2011. pp. 47-54.
[14]. A.S. Kamkin, T.I. Sergeeva, S.A. Smolov, A.D. Tatarnikov, M.M. Chupilko. Extensible Environment for Test Program Generation for Microprocessors. Programming and Computer Software, 40(1), 2014, pp. 1-9.
[15]. Библиотека Fortress. http://forge.ispras.ru/projects/solver-api
[16]. SMT-решатель Z3. http://github.com/Z3Prover/z3
[17]. SMT-решатель CVC4. http://cvc4.cs.nyu.edu
[18]. MIPS64™ Architecture For Programmers. Volume 2: The MIPS64™ Instruction Set Reference Manual. Revision 6.04. MIPS Technologies Inc. 2015. 551 p.
[19]. Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek. Constraint-Based Random Stimuli Generation for Hardware Verification AI Magazine, 28(3), 2007. pp. 13-30.

Трансляция вложенных сетей Петри в классические сети Петри для верификации разверток

В.О. Ермакова <ermakovavo@gmail.com>
И.А. Ломазова <ilomazova@hse.ru>
National Research University Higher School of Economics,
20 Myasnitskaya St., Moscow, 101000, Russia

Аннотация. Вложенные сети Петри являются одним из удобных формализмов для моделирования и анализа поведения распределенных мультиагентных систем. Они естественным образом представляют структуру мультиагентных систем, так как фишки в системной сети сами являются классическими сетями Петри и могут иметь автономное поведение. Мультиагентные системы являются системами с высоким уровнем параллелизма. При верификации таких систем методами проверки модели (model checking) возникают серьезные трудности, связанные с взрывным ростом числа промежуточных состояний системы (state-space explosion problem). Для решения этой проблемы в литературе был предложен подход, основанный на построении развертки поведения системы. Ранее была изучена применимость разверток для верификации вложенных сетей Петри и предложен метод построения разверток для безопасных консервативных вложенных сетей Петри. В этой работе предлагается другой метод построения разверток для безопасных консервативных вложенных сетей Петри, основанный на трансляции таких сетей в классические сети Петри. Для классических сетей Петри затем применяются стандартные методы построения разверток. Также в работе обсуждаются сравнительные достоинства двух подходов.

Ключевые слова: мультиагентные системы; верификация; сети Петри; вложенные сети Петри; развертки.

DOI: 10.15514/ISPRAS-2016-28(4)-7

Для цитирования: Ермакова В.О., Ломазова И.А. Трансляция вложенных сетей Петри в классические сети Петри для верификации разверток. Труды ИСП РАН, том 28, вып, 4, 2016, стр. 115-136. DOI: 10.15514/ISPRAS-2016-28(4)-7

1. Введение

Мультиагентные системы активно изучаются уже в течение нескольких десятилетий. Они используются в различных практических областях, таких как искусственный интеллект, облачные сервисы, грид системы, системы

встроенной реальности с интерактивными агентами среды, сборе информации, кооперации мобильных агентов и коммуникации.

Мультиагентные системы сложны из-за их распределенной структуры. Они состоят из взаимодействующих агентов, имеющих общую среду и автономное поведение. Когда разрабатывается такая система, важно проверить, будет ли она отвечать необходимым требованиям. Для сложных распределенных систем обычно сначала строят и анализируют (верифицируют) модель системы и только после проверки корректности модели переходят к этапу реализации.

Одним из формализмов, успешно представляющих поведение распределенных систем, являются сети Петри. Однако из-за плоской структуры классических сетей Петри они оказываются неудобны для моделирования сложных мультиагентных систем. Для таких систем используется специальные расширения сетей Петри, в частности, вложенные сети Пети [1]. Вложенные сети Петри естественно представляют структуру и поведение мультиагентных систем, так как фишки в системной сети сами являются сетями Петри и имеют собственное поведение.

Для проверки свойств сетей Петри часто используются методы верификации, основанные на проверке моделей (model checking). Основная идея этого подхода заключается в построении графа достижимости и проверке свойств на полученном графе. Однако при использовании этого метода для верификации высоко параллельных систем возникает серьезная проблема, связанная с размером графа достижимости системы. Эта проблема известна как проблема взрывного роста числа состояний (state-space explosion problem) – число промежуточных состояний системы растет экспоненциально от числа независимых параллельных агентов.

Одним из решений этой проблемы является проверка свойств системы не на графе достижимости, а на так называемой развертке (unfolding) ее поведения [2,3]. Ранее в работе [4] было показано, как теория разверток может быть применена для верификации вложенных сетей Петри, а именно, для безопасных консервативных вложенных сетей Петри было дано определение развертки и описан алгоритм ее построения. Было доказано, что для вложенных сетей Петри выполняется фундаментальное свойство разверток и, следовательно, развертки вложенных сетей могут быть использованы для верификации консервативных вложенных сетей Петри так же, как классические развертки используются для верификации классических сетей Петри.

В этой работе описывается другой способ построения разверток для безопасных консервативных вложенных сетей Петри. Консервативность означает, что сетевые фишки, представляющие агентов, не могут быть уничтожены или созданы, но могут изменять свое положение в системной сети, а также менять своё внутреннее состояние, т.е. количество агентов в системе не меняется. Безопасность означает, что в каждой позиции системной сети может одновременно находиться не более одной сетевой фишки (агента).

Мы показываем, что для любой безопасной консервативной вложенной сети Петри можно построить эквивалентную классическую сеть Петри, а затем применить метод построения разверток для классических сетей. Полученная в результате развертка будет изоморфна развертке вложенной сети Петри, построенной методом, описанным в [4].

1.1 ***Сравнение с другими исследованиями***

Вложенные сети Петри широко используются в моделировании распределенных систем [5,6,7], последовательных и реконфигурируемых системах [8,9,10], верификации протоколов [11], координации сенсорных сетей с мобильными агентами [12], инновационных архитектурах космических систем [13], распределенных вычислениях [14].

В литературе было предложено несколько методов для поведенческого анализа вложенных сетей Петри, среди них композициональные методы для проверки ограниченности и живости вложенных сетей Петри [15], трансляция вложенных сетей Петри в раскрашенные сети Петри и верификация их с помощью CPNtools [16], верификация подкласса рекурсивных вложенных сетей Петри с помощью SPIN [17].

Подход, основанный на построении развертки, и проблема взрывного роста числа состояний подробно описаны в литературе. Начало разработкам в области построения разверток для классических сетей Петри было положено в [18]. К. МакМилан [2] был первым, кто использовал развертки для верификации. Он представил концепцию конечных префиксов разверток и показал применимость этого подхода для верификации асинхронных цепей.

Исходный алгоритм МакМиллана был использован для решения проблемы выполнимости перехода – проверить, может данный переход сработать или нет. Этот алгоритм применим также для проверки наличия дедлоков и для решения некоторых других проблем. Позже улучшения алгоритма были представлены в [19,20,21]. Имеются также работы по применению разверток для верификации высокоуровневых сетей Петри [22], алгебр процессов [23] и М-сетей [22].

Общий подход для отсечения разверток с сохранением информации в конечном префиксе развертки предложен в [24,25]. Этот метод основан на понятии усеченного контекста. Мы используем этот подход для определения ветвящегося процесса и развертки консервативной вложенной сети Петри.

1.2 ***Структура работы***

В разделе 2 даны основные определения сетей Петри и вложенных сетей Петри. В разделе 3 представлен алгоритм для трансляции безопасных консервативных вложенных сетей Петри в классические. Раздел 4 посвящен сравнению метода построения разверток на основе трансляции вложенной сети Петри в классическую и метода непосредственного построения развертки вложенной сети. Последний раздел содержит выводы и обсуждения результатов работы.

2. Предварительные сведения

Пусть S – конечное множество. Мультимножеством m над множеством S называется функция $m: S \to Nat$, где Nat – множество неотрицательных целых чисел. Другими словами, мультимножество может содержать несколько копий одного и того же элемента.

Для двух мультимножеств m и m' полагаем $m \subseteq m'$, если $\forall s \in S : m(s) \leq m'(s)$ (отношение включения). Сумма и объединение двух мультимножеств m и m' также определяются стандартно: $\forall s \in S : (m + m')(s) = m(s) + m'(s), (m \cup m')(s) = max(m(s), m'(s))$.

2.1 Классические сети Петри

Пусть P и T – два конечных непересекающихся множества позиций и переходов и $F \subseteq (P \times T) \cup (T \times P)$ – функция инцидентности. Тогда $N = (P, T, F)$ является сетью Петри. Разметкой сети $N = (P, T, F)$ называется мультимножество над множеством позиций P. Через $\mathcal{M}(N)$ будем обозначать множество всех разметок сети N. Размеченная (маркированная) сеть Петри (N, M_0) – это сеть Петри вместе с её начальной разметкой M_0.

Графически сеть Петри представляется в виде ориентированного графа, в котором вершины-позиции изображаются кругами, а вершины-переходы – прямоугольниками. Позиции могут содержать фишки, представленные закрашенными кружками. Текущая разметка m определяется помещением $m(p)$ фишек в каждую позицию $p \in P$.

Для перехода $t \in T$ дуга (x, t) называется входящей дугой, а (t, x) – исходящей. Для каждой вершины $x \in P \cup T$ мы определяем пред-множество элементов для вершины x как $\bullet x = \{y \mid (y, x) \in F\}$.

Мы говорим, что переход t в сети Петри $N = (P, T, F)$ активен в разметке M если $\bullet t \subseteq M$. Активный переход может сработать и произвести новую разметку $M' = M - \bullet t + t \bullet$ (обозначается как $M \xrightarrow{t} M'$). Разметка M называется достижимой, если существует (возможно, пустая) последовательность срабатываний $M_0 \xrightarrow{t1} M_1 \xrightarrow{t2} M_2 \to \cdots \to M$ из начальной разметки в M. Через $\mathcal{RM}(N)$ обозначим множество всех достижимых разметок в N.

Разметка M называется безопасной, если для всех позиций $p \in P$ имеем $M(p) \leq 1$. Маркированная сеть Петри называется безопасной, если каждая достижимая разметка $M \in \mathcal{RM}(N)$ безопасна. Граф достижимости сети Петри (N, M_0) представляет детальную информацию о поведении сети. Это помеченный ориентированный граф, в котором вершины являются достижимыми разметками сети (N, M_0), а дуги соответствуют срабатываниям переходов. В графе достижимости дуга t между разметками M и M'существует тогда и только тогда, когда $M \xrightarrow{t} M'$.

2.2. Развертки классических сетей Петри

Развертки используются для представления семантики истинного параллелизма (true concurrency) сетей Петри. Для верификации используются конечные префиксы развёрток. Здесь мы приводим необходимые основные понятия и определения, связанные с развертками. Более детальное описание можно найти в [26,27].

Пусть, $N = (P, T, F)$ – сеть Петри. Следующие отношения определены на множестве $P \cup T$ вершин в N:

- Отношение (каузальной) зависимости (обозначается $<$) – это транзитивное замыкание F, соответственно, $\leq$ – рефлексивное замыкание $<$; мы говорим, что y зависит от x, если $x < y$.
- Отношение конфликта (обозначается #): для вершин $x, y \in P \cup T, x\#y \coloneqq \exists t, t' \in T. t \neq t' \wedge {}^\bullet t \cap {}^\bullet t' \neq \emptyset \wedge t \leq x \wedge t' \leq y$;
- Отношение параллельности (обозначается co): две вершины сети параллельны, если они не находятся в конфликте и ни одна из них не зависит от другой.

Для множества вершин B мы пишем $co(B)$ если все вершины в B являются попарно параллельными.

Сетью событий называется безопасная сеть Петри $ON = (B, E, G)$ такая, что:

- ON – ациклична;
- $\forall p \in B: |{}^\bullet p| \leq 1$;
- $\forall x \in B \cup E$ множество $\{y \mid y < x\}$ конечно, то есть каждая вершина в ON имеет конечное число предшествующих вершин;
- $\forall x \in B \cup E: \neg(x\#x)$, то есть, ни одна вершина не находится в конфликте с собой.

В сетях событий элементы из B обычно называются условиями, а элементы принадлежащие E – событиями.

Сети событий представляют поведение системы в семантике «истинного параллелизма» (true concurrency semantics). Семантика истинного параллелизма отличается от последовательной (интерливинговой) семантики тем, что в последовательной семантике в каждый конкретный момент времени может происходить не более одного события. В семантике истинного параллелизма это не так, и несколько событий могут происходить одновременно. Внешний наблюдатель не различает эти две семантики. Также, последовательная семантика проще и более удобна для анализа поведенческих свойств, поэтому она часто используется. Тем не менее, когда модель должна учитывать время, разница между этими семантиками становится заметной.

Перейдем к определению ветвящихся процессов и разверток. Конфигурацией $\mathcal{C}$ в сети событий $ON = (B, E, G)$ называют бесконфликтное подмножество вершин, которое замкнуто относительно отношения $<$, то есть $\forall x, y \in \mathcal{C}: \neg(x\#y)$ и $(x < y) \wedge y \in \mathcal{C}$, где $x \in \mathcal{C}$. Для каждого $x \in B \cup E$ мы определяем

локальную конфигурацию x такую что $[x] = \{y | y \in B \cup E, y < x\}$. Определение локальной конфигурации может быть обобщено на любое бесконфликтное множество вершин $X \subseteq B \cup E$, а именно $[X] = \{y | y \in B \cup E, x \in X, y < x\}$.

Определим множество ветвящихся процессов для данной маркированной сети Петри $N = (P, T, F, M_0)$, используя, так называемое, каноническое представление.

Множество C канонических имен N определено рекурсивно, как минимальное множество такое, что если $x \in P \cup T$ и A – конечное подмножество C, то $(A, x) \in C$.

Сеть событий (B, E, G) называется -сетью, если выполняются следующие условия:

- $B \cup E \subseteq C$;
- $\forall (A, x) \in B \cup E, \bullet (A, x) = A$.

Начальная разметка -сети Петри – это подмножество вершин $\{(\emptyset, x) | (\emptyset, x \in B)\}$. Для каждой C-сети CN определяется функция (морфизм) h, отображающая вершины CN на вершины сети N: $h((A, x)) = x$.

Пусть S – конечное или бесконечное множество -сетей. Тогда объединение сетей из S определяется покомпонентно, то есть:

$$\cup S = (\cup_{(P,T,F,M) \in S} P, \cup_{(P,T,F,M) \in S} T, \cup_{(P,T,F,M) \in S} F, \cup_{(P,T,F,M) \in S} M).$$

Множество ветвящихся процессов маркированной сети Петри $N = (P, T, F, M_0)$ определяется как наименьшее множество -сетей, удовлетворяющее следующим условиям:

- C-сеть $(I, \emptyset, \emptyset)$, где $I = \{(\emptyset, p) | p \in M_0\}$ (состоящая из условий I и не содержащая событий) – это ветвящийся процесс.
- Пусть $\mathcal{B}_1$ – ветвящийся процесс, M – достижимая разметка для $\mathcal{B}_1$, и $M' \subseteq M$, так что $h(M') = \bullet t$ для некоторого t в T. Пусть $\mathcal{B}_2$ – сеть, полученная с помощью добавления события (M', t) и условий $\{(\{(M', t)\}, p) | p \in t \bullet\}$ к $\mathcal{B}_1$. Тогда $\mathcal{B}_2$ – ветвящийся процесс.
- Пусть $\mathcal{BB}$ – конечное или бесконечное множество ветвящихся процессов. Объединение $\cup \mathcal{BB}$ также является ветвящимся процессом.

На Рис.2 показан пример ветвящегося процесса для сети Петри $PN1$ с начальной разметкой $\{p1\}$, изображенной на Рис.1.

Ветвящийся процесс $\mathcal{B}_1 = ((P_1, E_1, F_1), h_1)$ называется префиксом ветвящегося процесса $\mathcal{B}_2 = ((P_2, E_2, F_2), h_2)$, (обозначается $\mathcal{B}_1 \sqsubseteq \mathcal{B}_2$), если $P_1 \subseteq P_2$ и $E_1 \subseteq E_2$.

Максимальный относительно частичного порядка $\sqsubseteq$ ветвящийся процесс сети N называется разверткой сети N и обозначается как $U(N)$.

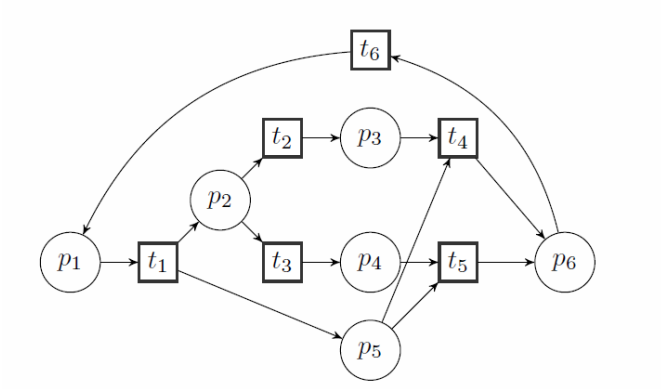


Рис. 1 Сеть Петри PN1
Fig. 1 Petri Net PN1

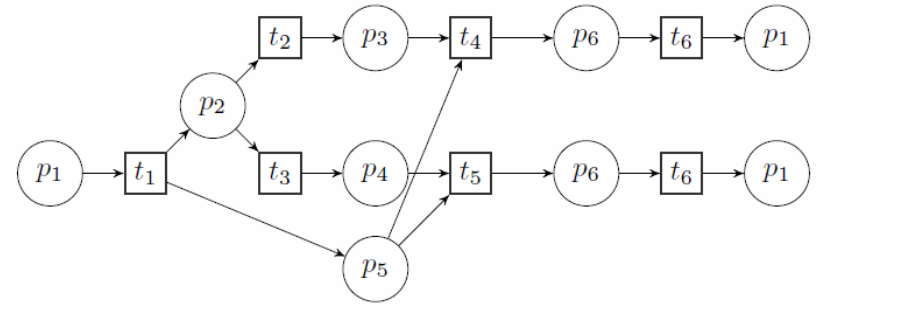


Рис. 2 Ветвящийся процесс для PN1
Fig 2 Branching process for PN1

2.3. Фундаментальное свойство разверток

Выполнение фундаментального свойства разверток означает, что для любой сети Петри поведение развертки эквивалентно поведению исходной сети. Фундаментальное свойство формулируется следующим образом.

Пусть M – достижимая маркировка сети Петри N, и пусть M_U является достижимой разметкой в $U(N)$ такой, что $h(M_U) = M$. Тогда:

- если существует шаг $M_U \xrightarrow{t_U} M_U{}'$ в $U(N)$, то существует шаг $M \xrightarrow{t} M'$ в N такой, что $h(t_U) = t \wedge h(M'_U) = M'$;
- если существует шаг $M \xrightarrow{t} M'$ в N, то существует шаг $M_U \xrightarrow{t_U} M_U{}'$ в $U(N)$ такой, что $h(t_U) = t \wedge h(M'_U) = M'$.

Другими словами, фундаментальное свойство разверток гласит, что граф достижимости развертки изоморфен графу достижимости исходной сети Петри. Это свойство очень важно для использования разверток при анализе семантических свойств и верификации. Развертки определены и изучены для различных классов сетей Петри: для сетей Петри высокого уровня [22], контекстных сетей [28], временных сетей Петри [29], гиперсетей [30]. Все эти конструкции имеют сходные свойства, которые служат обоснованием применимости определенных в этих работах разверток для верификации. Далее будет приведено определение развертки для вложенных сетей Петри, для которого также выполняется фундаментальное свойство разверток.

2.4 Вложенные сети Петри

В этой работе мы рассматриваем вложенные сети Петри, точнее, их специальный подкласс, называемый строго консервативными вложенными сетями Петри. Более подробную информацию о вложенных сетях Петри можно найти в [1,7]. Здесь мы приводим сокращенное определение, подходящее для рассматриваемого случая.

Во вложенных сетях Петри фишки сами могут быть сетями Петри. Вложенная сеть Петри состоит из системной и элементных сетей. Мы называем их компонентами вложенной сети Петри. Маркированные элементные сети называются сетевыми фишками. Сетевые фишки, также как и обычные черные фишки, могут находиться в позициях системной сети. Некоторые переходы во вложенных сетях Петри могут быть помечены метками синхронизации. Непомеченные переходы во вложенных сетях Петри могут срабатывать автономно, согласно правилам срабатывания переходов в классических сетях Петри. Помеченные переходы в системной сети должны синхронизироваться с переходами (имеющими такую же пометку) в сетевых фишках, задействованных в это срабатывание перехода.

В этой работе мы рассматриваем безопасные и типизированные вложенные сети Петри, то есть каждая позиция системной сети может содержать не более одной фишки: черной, либо сетевой фишки определенного типа.

Пусть $Type$ – множество типов, Var – множество типизированных (типами из $Type$) переменных, а Lab – множество меток. Типизированной вложенной сетью Петри NP называется кортеж $(SN, (EN_1, \dots EN_k), v, \lambda, W)$, где:

- $SN = (P_{SN}, T_{SN}, F_{SN})$ – сеть Петри, называемая системной сетью;
- Для каждого $i = \overline{1,k}$, $EN_i = P_{EN_i}, T_{EN_i}, F_{EN_i})$ есть сеть Петри, называемая элементной сетью; множества переходов и позиций в системной и элементных сетях попарно не пересекаются; каждой элементной сети приписан тип из *Type*;
- $v: P_{SN} \to Type \cup \{\bullet\}$ – функция типизации позиций системной сети;
- $\lambda: T_{NP} \to Lab$ – частичная функция пометки переходов, где $T_{NP} = T_{SN} \cup T_{EN_1} \cup \dots \cup T_{EN_k}$. Будем писать $\lambda(t) = \perp$, если λ не определена для t.

- W: $F_{SN} \to Var \cup \{\bullet\}$ – функция пометки дуг такая, что для дуги r тип выражения $W(r)$ совпадает с типом позиции, инцидентной r.

Маркированная элементная сеть называется сетевой фишкой. Далее для данной вложенной сети Петри через $A_{net} = \{(EN, m) | \exists i = 1, \dots k: EN = EN_i, m \in \mathcal{M}(EN_i)\}$ обозначим множество всех возможных фишек сети вместе с черными фишками (black dot token).

Элементно-автономный шаг. Пусть t – непомеченный переход в одной из сетевых фишек. Тогда срабатывание перехода t определяется стандартными правилами срабатывания перехода в сетях Петри. Сама фишка остается в той же самой позиции системной сети.

Системно-автономный шаг – это срабатывание непомеченного перехода $t \in T_{SN}$ в системной сети в соответствии с правилами срабатывания для высокоуровневых сетей Петри (например, раскрашенных сетей Петри [31]).

Синхронный шаг. Пусть t – переход, помеченный λ в системной сети SN, t активный в разметке M при означивании b переменных в выражениях на дугах. Пусть далее $\alpha_1, \dots \alpha_n \in A_{net}$ – сетевые фишки, задействованные в срабатывании t. Тогда t может сработать при условии, что в каждой сетевой фишке α_i $(1 \leq i \leq n)$, задействованной в срабатывании t, имеется активный переход, помеченный той же синхронизационной меткой λ. Синхронный шаг выполняется в два этапа: сначала срабатывает по одному переходу, помеченному λ, в каждой из сетевых фишек, задействованных в срабатывании t, и затем выполняется срабатывание t в системной сети.

Вложенная сеть NP называется безопасной, если в каждой ее достижимой разметке имеется не более одной фишки в каждой позиции в системной сети и в каждой позиции сетевых фишек.

Далее мы будем рассматривать только безопасные сети.

2.5 Консервативные вложенные сети Петри

Безопасная вложенная сеть Петри $N = (SN, (EN_1, \dots EN_k), v, \lambda, W)$ называется строго консервативной если

- Для каждого $t \in T_{SN}$ и для каждого $p \in \bullet t$, $\exists! p' \in t \bullet$. $W(p, t) = W(t, p')$ или $W(p, t) = \bullet$;
- Для каждого $t \in T_{SN}$ и для каждого $p \in t \bullet$, $\exists! p' \in \bullet t$. $W(p', t) = W(t, p)$ или $W(p, t) = \bullet$.

Строгая консервативность означает, что сетевые фишки не могут появляться или исчезать после срабатывания перехода в системной сети.

Заметим, что в [15] вложенные сети Петри называются консервативными, если фишки не могут исчезать после срабатывания перехода, но могут копироваться, таким образом, количество сетевых фишек в таких консервативных вложенных сетях Петри может быть не ограничено. Здесь мы рассматриваем более узкий подкласс вложенных сетей Петри с постоянным количеством сетевых фишек

(фишки не могут копироваться). Следует отметить, что хотя это ограничение довольно строгое, с помощью консервативных вложенных сетей Петри можно моделировать много важных и интересных мультиагентных систем.

3. Трансляция безопасных консервативных вложенных сетей Петри в классические сети Петри

Поскольку безопасные консервативные вложенные сети Петри ограничены, т.е. множество достижимых состояний любой такой сети конечно, для каждой такой сети существует эквивалентная по поведению классическая ограниченная сеть Петри. В этом разделе будет представлен алгоритм трансляции безопасной консервативной вложенной сети Петри в классическую сеть Петри с эквивалентным поведением и структурой, которая соответствует структуре исходной вложенной сети. Далее будет показано, что развертка полученной классической сети Петри изоморфна развертке исходной вложенной сети Петри. Таким образом, трансляция в классические сети Петри может быть использована для построения развертки и верификации ограниченных консервативных вложенных сетей Петри.

Алгоритм трансляции безопасных консервативных вложенных сетей Петри будет проиллюстрирован на примере вложенной сети Петри $NP2$, показанной на Рис. 3 (системная сеть) и Рис. 4 (элементная сеть). Эта сеть будет транслирована в безопасную классическую сеть Петри PN.

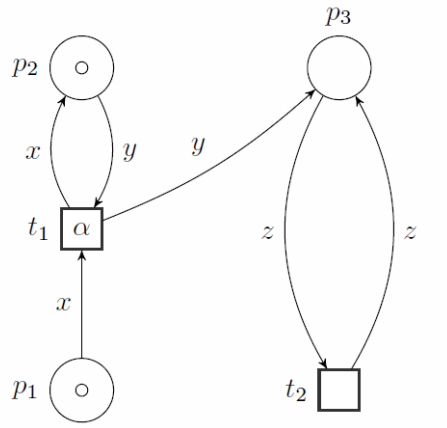


Рис. 3 Вложенная сеть Петри $NP2$
Fig. 3 Nested Perti Net NP2

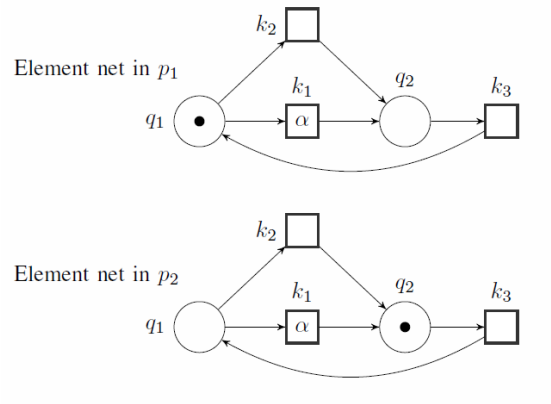


Рис. 4 Вложенная сеть Петри $NP2$

Fig. 4 Nested Perti Net NP2

3.1 Алгоритм трансляции

Пусть $NP = (SN, (EN_1, \dots EN_k), v, \lambda, W)$ – вложенная сеть Петри с множеством $NTok$ идентифицированных сетевых фишек в начальной разметке. Через I мы обозначим множество всех идентификаторов, использованных в $NTok$, а через $I_E \subseteq E$ обозначим подкласс идентификаторов сетевых фишек типа E. Сеть NP будет транслирована в сеть Петри $PN = (P_{PN}, T_PN, F_{PN})$ с начальной разметкой m_0.

- Сначала определяем множество P_{PN} позиций целевой сети PN. Для каждого типа E некоторой позиции в системной сети SN мы создаем множество $\mathfrak{S}_E$ позиций для P_{PN}. Множество $\mathfrak{S}_E$будет содержать копию каждой позиции типа E в системной сети для каждой сетевой фишки типа E (помеченную идентификатором сетевой фишки) и копию каждой позиции в PE для каждой сетевой фишки типа E, то есть, $\mathfrak{S}_E = \{(p, id) | p \in P_{SN}, v(p) = E, id \in I_E\} \cup \{(q, id) | q \in P_E, id \in I_E\}$. Для каждой позиции в SN типа черной фишкой создаем только одну копию p без идентификатора. Затем множество P_{PN} позиций для целевой сети PN определяется как объединение этих множеств. Результат выполнения первого шага алгоритма для вложенной сети $NP2$ изображен на Рис. 5.

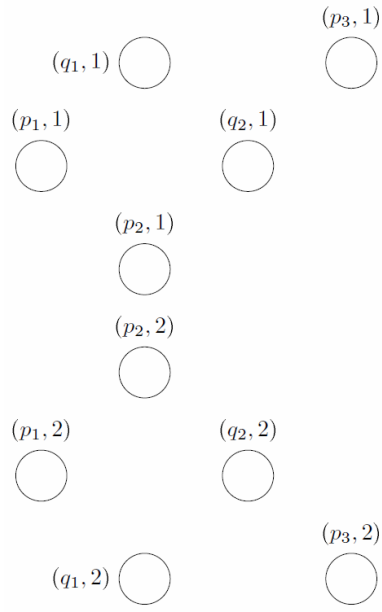


Рис. 5 Создание позиций PN

Fig. 5 Position creation for PN

- Чтобы определить начальную разметку для сети PN разметку позиций P_{NP} во вложенной сети Петри кодируем разметками на построенных позициях сети P_{PN}. Если сетевая фишка $\eta = (id, E, m)$ находится в позиции p в разметке M системной сети, то в целевой сети черные фишки помещаются в позицию (p, id) и во все позиции (q, id) для всех q таких, что $m(q) = 1$. Если позиция типа черной точки в системной сети SN содержит черную фишку, то единственная соответствующая позиция в сети PN также будет содержать черную фишку. Легко заметить, что эта кодировка определяет взаимно-однозначное соответствие между разметками консервативной безопасной вложенной сети Петри и безопасными разметками в PN. В нашем примере первая элементная сеть находится в позиции $p1$, вторая – в позиции $p2$. Таким образом, черные фишки помещаются в позиции $(p_1, 1)$ и $(p_2, 2)$; а также в $(q_1, 1)$ и $(q_1, 2)$.

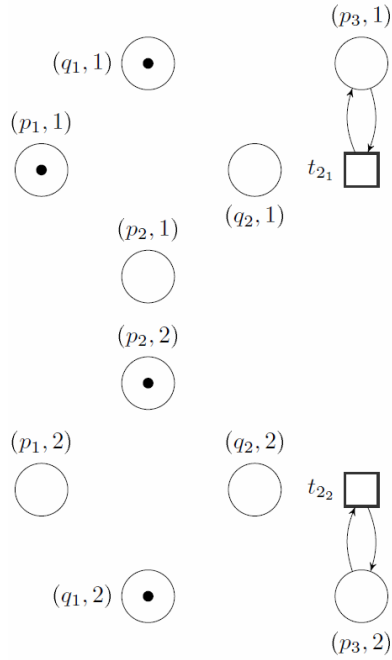


Рис. 6 Системно-автономный шаг

Fig. 6 System-autonomous step

- Для каждого автономного перехода t в системной сети SN мы строим множество T_t переходов следующим образом. Поскольку каждая переменная на входной дуге перехода t может быть означена, вообще говоря, любой сетевой фишкой подходящего типа, то для каждого такого означивания строится отдельный переход в PN с соответствующими входными и выходными дугами. В нашем примере для перехода t_2 мы строим два перехода t_{21}и t_{22}.
- Для каждого автономного перехода в сетевой фишке из $NTok$ строится соответствующий переход, инцидентный позициям, помеченным id. Так в нашем примере мы получим четыре перехода: k_{21}, k_{22}, k_{31} и k_{32}.

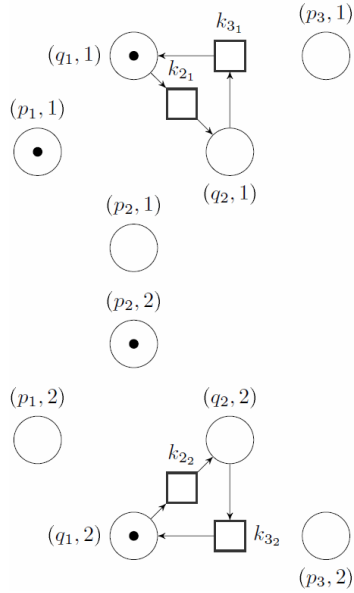


Рис. 7. Элементно-автономный шаг
Fig. 7. Element-autonomous step

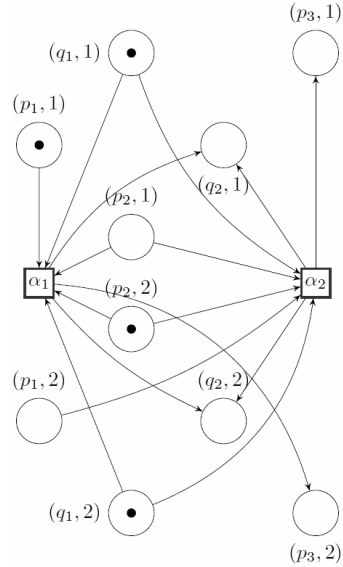


Рис. 8 Синхронный шаг
Fig 8. Synchronous step

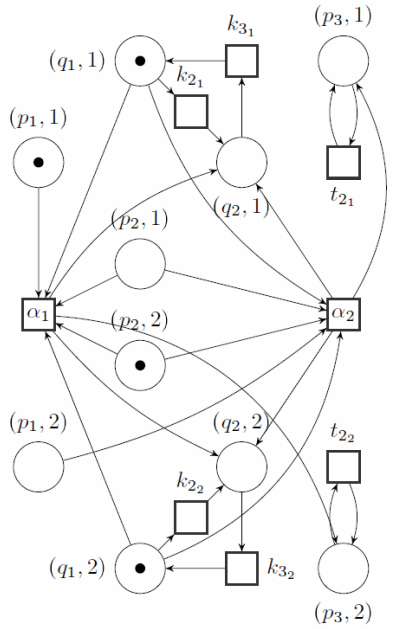


Рис. 9 Результат трансляции $NP2$ в классическую сеть Петри
Fig. 9. The result of NP2 translation into classical Petri Net

- Срабатывание синхронного перехода означает одновременное срабатывание перехода в системной сети и срабатывание переходов, помеченных такой же меткой в каждой задействованной в этом срабатывании сетевой фишке. Таким образом, синхронный шаг является комбинацией шага 3 и шага 4. В нашем примере есть две сетевые фишки, и переходы α_1 и α_2 строятся для каждой из них. Таким образом мы можем моделировать синхронный шаг для каждой из возможных начальных разметок системной сети.

Корректность определенной выше трансляции обеспечивает следующая

Теорема 1.
Пусть NP – вложенная сеть Петри и PN – сеть Петри, полученная из сети NP с помощью трансляции, описанной выше. Тогда графы достижимости сетей NP и PN изоморфны.

Доказательство. Шаг 2 алгоритма определяет взаимное соответствие между достижимыми разметками сетей NP и PN. Легко заметить, что в соответствии с определением трансляции соответствующие шаги срабатывания в обеих сетях не нарушают это соответствие.

4 Построение развертки для ограниченной консервативной вложенной сети Петри

После трансляции вложенной сети Петри в классическую для полученной классической сети Петри можно построить ветвящиеся процессы и развертки, пользуясь известными методами. Так на Рис. 10 показан один из ветвящихся процессов для сети $NP2$. Развертка тогда путем отсечения конечного префикса ветвящегося процесса по сечению, определяющему состояние, которое уже входит в этот префикс.

Сравним метод построения развертки путем трансляции вложенной сети Петри в классическую покомпонентным методом, предложенным в [4].

Теорема 2.

Развертка безопасной консервативной вложенной сети Петри *NP*, полученная описанным в [4] методом, изоморфна развертке классической безопасной сети Петри, полученной в результате описанной выше трансляции сети *NP*.

Доказательство. В [4] было доказано выполнение фундаментального свойства развертки для описанных там покомпонентных разверток, а именно, что граф достижимости вложенной сети Петри изоморфен графу достижимости его покомпонентной развертки. По *Теореме 1* граф достижимости классической сети, полученной в результате трансляции, изоморфен графу достижимости исходной вложенной сети. Для классических сетей Петри выполняется фундаментальное свойство разверток, т.е. граф достижимости развертки изоморфен графу достижимости сети Петри. Из всего этого следует, что развертки, построенные методом из [4] и путем трансляции в классические сети Петри, изоморфны.

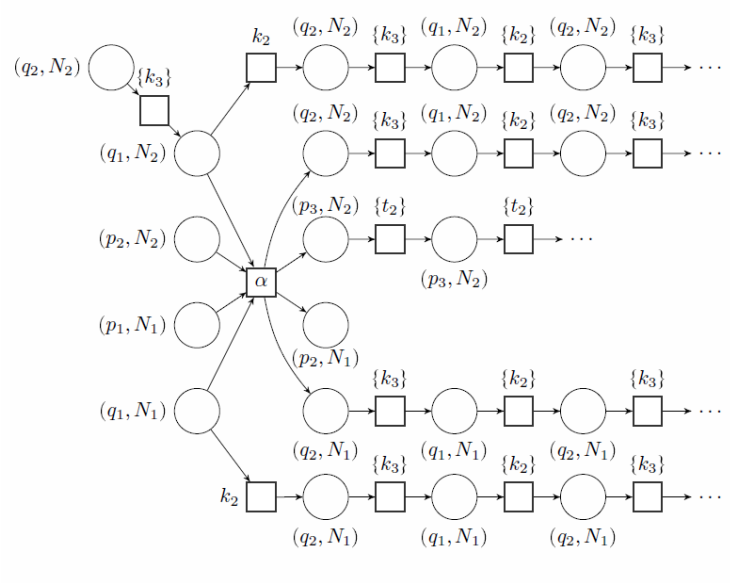


Рис. 10 Ветвящийся процесс для сети NP2
Fig. 10. Branching process for NP2 net

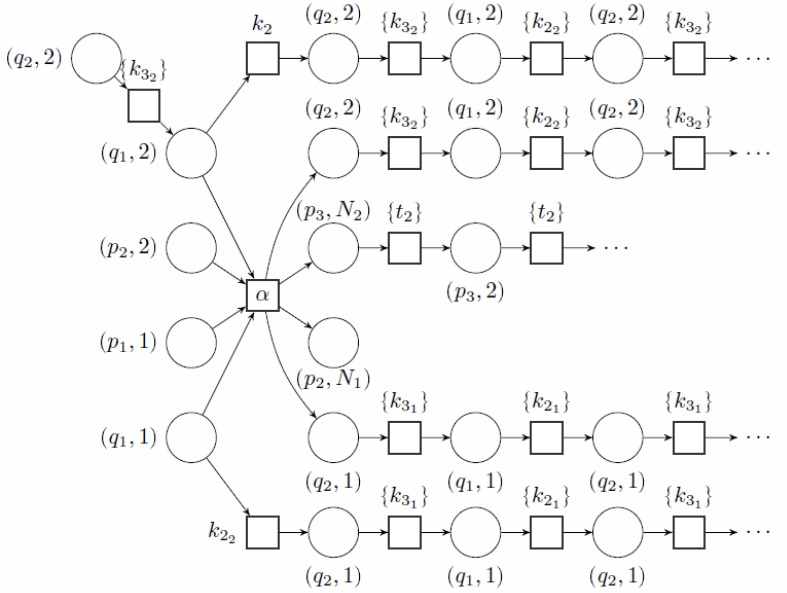


Рис. 11 Ветвящийся процесс, построенный путем трансляции сети NP2
Fig. 11. Branching process built after NP2 translation

5 Заключение

В этой работе мы показали, что каждая консервативная безопасная вложенная сеть Петри может быть транслирована в поведенчески эквивалентную классическую сеть Петри так, что их графы достижимости изоморфны. Таким образом, чтобы построить развертки для вложенной сети Петри достаточно транслировать её в классическую сеть Петри и потом применить метод развертки для классических сетей Петри.

Другой способ построения развертки безопасной консервативной вложенной сети Петри описан в [4]. В этой работе развертка сети строится непосредственно путем построения разверток отдельных компонентов вложенной сети Петри. Мы показали, что оба метода дают одинаковый с точностью до изоморфизма результат.

Каждый из этих методов имеет свои преимущества и недостатки. Построение разверток вложенных сетей Петри путем трансляции в классические сети Петри позволяет использовать готовые методы и инструментарий для построения разверток и верификации классических сетей Петри. Применение метода построения разверток непосредственно для вложенных сетей Петри, описанного в [4], требует разработки специального программного обеспечения. С другой стороны, построение промежуточной классической сети Петри может привести к значительному росту затрат времени и памяти при построении развертки вложенной сети.

Поэтому интересно сравнить сложность этих двух методов: метода, предложенного в [4], и подхода, основанного на трансляции вложенных сетей Петри в классические. Нетрудно заметить, что если в начальной разметке сети имеется несколько сетевых фишек одного и того же, то трансляция вложенной сети в классическую сеть Петри ведет к значительному росту сети. Так для перехода системной сети с n входными позициями одного типа и k фишек этого типа в начальной разметке строится k^n копий этого перехода в целевой сети Петри. Понятно, что этого нельзя избежать, так как необходимо различать разметки сетевых фишек, находящихся в разных позициях системной сети.

Чтобы сравнить временные затраты этих двух методов на практике, нами были реализованы компьютерные программы, позволяющие выполнять

- трансляцию консервативной безопасной сетей Петри в классическую сеть Петри и построение развертки для нее;
- построение развертки для вложенной сети Петри напрямую.

Наша гипотеза состояла в том, что увеличение количества сетевых фишек ведет к значительному росту классической сети во время трансляции. Чтобы получить репрезентативные результаты, мы провели эксперименты на сетях, имеющих сходную структуру, но разное количество элементных сетей разных типов.

Эксперименты с сетями небольшого размера подтвердили нашу гипотезу. Так, для сети $NP2$ из нашего примера время построения развертки составило 0.38

мс. при применении метода покомпонентной развертки и 0.54 мс. при построении развертки путем трансляции в классическую сеть Петри. Таким образом, даже в случае двух сетевых фишек, разница времени исполнения весьма заметна. В будущем мы планируем провести эксперименты по сравнительной оценке сложности двух методов построения разверток на больших примерах. Также планируется выполнить эксперименты по верификации конкретных свойств вложенных сетей Петри с помощью построения разверток.

Благодарность

Работа выполнена при поддержке Программы фундаментальных исследований НИУ ВШЭ и Российского фонда фундаментальных исследований (проект 16-01-00546).

Список литературы

[1]. Lomazova I.A. Nested Petri nets—a formalism for specification and verification of multi-agent distributed systems. Fundamenta Informaticae 43(1), 2000, pp. 195–214.
[2]. McMillan K.L. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. Computer Aided Verification, Springer, 1992, pp. 164–177.
[3]. Nielsen M., Plotkin G., Winskel G. Petri nets, event structures and domains, part I. Theoretical Computer Science 13(1), 1981, pp. 85–108.
[4]. Frumin D., Lomazova I.A. Branching processes of conservative nested Petri nets. VPT 2014. Second International Workshop on Verification and Program Transformation Vol. 28: EPiC Series. EasyChair, 2014. P. 19-35.
[5]. Lomazova I.A., van Hee K.M., Oanea O., Serebrenik A., Sidorova N., Voorhoeve M. Nested nets for adaptive systems. Application and Theory of Petri Nets and Other Models of Concurrency, LNCS, 2006, pp. 241–260.
[6]. Lomazova I.A. Modeling dynamic objects in distributed systems with nested petri nets. Fundamenta Informaticae 51(1-2), 2002, pp. 121–133.
[7]. Lomazova I.A. Nested petri nets for adaptive process modeling. Pillars of computer science. Springer, 2008, pp. 460–474
[8]. L´opez-Mellado E., Villanueva-Paredes N., Almeyda-Canepa H. Modelling of batch production systems using Petri nets with dynamic tokens. Mathematics and Computers in Simulation 67(6), 2005, pp. 541–558.
[9]. Kahloul L., Djouani K., Chaoui A. Formal study of reconfigurable manufacturing systems: A high level Petri nets based approach. Industrial Applications of Holonic and Multi-Agent Systems. Springer, 2013, pp. 106–117.
[10]. Zhang, L., Rodrigues, B. Nested coloured timed Petri nets for production configuration of product families. International journal of production research 48(6), 2010, pp. 1805–1833.
[11]. Venero M.L.F., da Silva F.S.C. Modeling and simulating interaction protocols using nested Petri nets. Software Engineering and Formal Methods. Springer, 2013, pp. 135–150.
[12]. Chang L., He X., Lian J., Shatz S. Applying a nested Petri net modeling paradigm to coordination of sensor networks with mobile agents. Proc. of Workshop on Petri Nets and Distributed Systems, Xian, China, 2008, pp. 132–145.
[13]. Cristini F., Tessier C. Nets-within-nets to model innovative space system architectures. Application and Theory of Petri Nets. Springer, 2012, pp. 348–367.

[14]. Mascheroni M., Farina F. Nets-within-nets paradigm and grid computing. Transactions on Petri Nets and Other Models of Concurrency V. Springer, 2012, pp. 201–220.
[15]. Dworza´nski L.W., Lomazova I.A. On compositionality of boundedness and liveness for nested Petri nets. Fundamenta Informaticae 120(3-4), 2012, pp. 275–293.
[16]. Dworza´nski L., Lomazova I.A. CPN tools-assisted simulation and verification of nested Petri nets. Automatic Control and Computer Sciences 47(7), 2013, pp. 393–402.
[17]. Venero M.L.F. Verifying cross-organizational workflows over multiagent based environments. Enterprise and Organizational Modeling and Simulation. Springer, 2014, pp. 38–58.
[18]. Winskel G. Event structures. Springer, 1986.
[19]. Bonet B., Haslum P., Hickmott S., Thi´ebaux S. Directed unfolding of petri nets. Transactions on Petri Nets and Other Models of Concurrency I. Springer, 2008, pp. 172–198.
[20]. McMillan K.L. A technique of state space search based on unfolding. Form. Methods Syst. Des. 6(1), 1995, pp. 45–65.
[21]. Heljanko K. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. Fundamenta Informaticae 37(3), 1999, pp. 247–268.
[22]. Khomenko V., Koutny M. Branching processes of high-level Petri nets. In Garavel, H., Hatcliff, J., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2619 of Lecture Notes in Computer Science. Springer, 2003, pp. 458–472.
[23]. Langerak R., Brinksma E. A complete finite prefix for process algebra. Computer Aided Verification, Springer, 1999, pp. 184–195.
[24]. Khomenko V., Koutny M., Vogler W. Canonical prefixes of Petri net unfoldings. Acta Informatica 40(2), 2003, pp. 95–118.
[25]. Khomenko V. Model Checking Based on Prefixes of Petri Net Unfoldings. Ph.D. Thesis, School of Computing Science, Newcastle University, 2003.
[26]. Esparza J., Heljanko K. Unfoldings: a partial-order approach to model checking. Springer, 2008.
[27]. Engelfriet J. Branching processes of Petri nets. Acta Informatica 28(6), 1991, pp.575–591.
[28]. Baldan P., Corradini A., Knig B., Schwoon S. Mcmillans complete prefix for contextual nets. Jensen, K., Aalst, W.M., Billington, J., eds.: Transactions on Petri Nets and Other Models of Concurrency I. Volume 5100 of Lecture Notes in Computer Science. Springer, 2008, pp. 199–220.
[29]. Fleischhack H., Stehno C. Computing a finite prefix of a time Petri net. Esparza J., Lakos C., eds.: Application and Theory of Petri Nets 2002. Volume 2360 of Lecture Notes in Computer Science. Springer, 2002, pp. 163–181.
[30]. Mascheroni, M. Hypernets: a Class of Hierarchical Petri Nets. Ph.D. Thesis, Facolt di Scienze Naturali Fisiche e Naturali, Dipartimento di Informatica Sistemistica e Comunicazione, Universit`a Degli Studi Di Milano Bicocca, 2010.
[31]. Jensen K., Kristensen L.M. Coloured Petri nets: modelling and validation of concurrent systems. Springer, 2009.

Translation of Nested Petri Nets into Classical Petri Nets for Unfoldings Verification

V.O. Ermakova <ermakovavo@gmail.com>
I.A. Lomazova <ilomazova@hse.ru>
National Research University Higher School of Economics,
20 Myasnitskaya St., Moscow, 101000, Russia

Аннотация. Nested Petri nets (NP-nets) have proved to be one of the convenient formalisms for distributed multi-agent systems modeling and analysis. It allows representing multi-agent systems structure in a natural way, since tokens in the system net are Petri nets themselves, and have their own behavior. Multi-agent systems are highly concurrent. Verification of such

systems with model checking method causes serious difficulties arising from the huge growth of the number of system intermediate states (state-space explosion problem). To solve this problem an approach based on unfolding system behavior was proposed in the literature. Earlier in [4] the applicability of unfolding for nested Petri nets verification was studied, and the method for constructing unfolding for safe conservative nested Petri nets was proposed. In this work we propose another method for constructing safe conservative nested Petri nets unfoldings, which is based on translation of such nets into classical Petri nets and applying standard method for unfolding construction to them. We discuss also the comparative merits of the two approaches.

Key words: multi-agent systems; verification; Petri nets; nested Petri nets; unfoldings.

DOI: 10.15514/ISPRAS-2016-28(4)-7

For citation: Ermakova V.O., Lomazova I.A. Translation of Nested Petri Nets into Classical Petri Nets for Unfoldings Verification. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 4, 2016, pp. 115-136 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-7

References

[1]. Lomazova I.A. Nested Petri nets—a formalism for specification and verification of multi-agent distributed systems. Fundamenta Informaticae 43(1), 2000, pp. 195–214.
[2]. McMillan K.L. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. Computer Aided Verification, Springer, 1992, pp. 164–177.
[3]. Nielsen M., Plotkin G., Winskel G. Petri nets, event structures and domains, part I. Theoretical Computer Science 13(1), 1981, pp. 85–108.
[4]. Frumin D., Lomazova I.A. Branching processes of conservative nested Petri nets. VPT 2014. Second International Workshop on Verification and Program Transformation Vol. 28: EPiC Series. EasyChair, 2014. P. 19-35.
[5]. Lomazova I.A., van Hee K.M., Oanea O., Serebrenik A., Sidorova N., Voorhoeve M. Nested nets for adaptive systems. Application and Theory of Petri Nets and Other Models of Concurrency, Lecture Notes in Computer Science, vol. 4024, 2006, pp. 241–260.
[6]. Lomazova I.A. Modeling dynamic objects in distributed systems with nested petri nets. Fundamenta Informaticae 51(1-2), 2002, pp. 121–133.
[7]. Lomazova I.A. Nested petri nets for adaptive process modeling. Pillars of computer science. Lecture Notes in Computer Science, vol. 4800, Springer, 2008, pp. 460–474
[8]. L´opez-Mellado E., Villanueva-Paredes N., Almeyda-Canepa H. Modelling of batch production systems using Petri nets with dynamic tokens. Mathematics and Computers in Simulation 67(6), 2005, pp. 541–558.
[9]. Kahloul L., Djouani K., Chaoui A. Formal study of reconfigurable manufacturing systems: A high level Petri nets based approach. Industrial Applications of Holonic and Multi-Agent Systems. Springer, 2013, pp. 106–117.
[10]. Zhang, L., Rodrigues, B. Nested coloured timed Petri nets for production configuration of product families. International journal of production research 48(6), 2010, pp. 1805–1833.
[11]. Venero M.L.F., da Silva F.S.C. Modeling and simulating interaction protocols using nested Petri nets. Software Engineering and Formal Methods. Springer, 2013, pp. 135–150.
[12]. Chang L., He X., Lian J., Shatz S. Applying a nested Petri net modeling paradigm to coordination of sensor networks with mobile agents. Proc. of Workshop on Petri Nets and Distributed Systems, Xian, China, 2008, pp. 132–145.

[13]. Cristini F., Tessier C. Nets-within-nets to model innovative space system architectures. Application and Theory of Petri Nets. Springer, 2012, pp. 348–367.
[14]. Mascheroni M., Farina F. Nets-within-nets paradigm and grid computing. Transactions on Petri Nets and Other Models of Concurrency V. Springer, 2012, pp. 201–220.
[15]. Dworza´nski L.W., Lomazova I.A. On compositionality of boundedness and liveness for nested Petri nets. Fundamenta Informaticae 120(3-4), 2012, pp. 275–293.
[16]. Dworza´nski L., Lomazova I.A. CPN tools-assisted simulation and verification of nested Petri nets. Automatic Control and Computer Sciences 47(7), 2013, pp. 393–402.
[17]. Venero M.L.F. Verifying cross-organizational workflows over multiagent based environments. Enterprise and Organizational Modeling and Simulation. Springer, 2014, pp. 38–58.
[18]. Winskel G. Event structures. Springer, 1986.
[19]. Bonet B., Haslum P., Hickmott S., Thi´ebaux S. Directed unfolding of petri nets. Transactions on Petri Nets and Other Models of Concurrency I. Springer, 2008, pp. 172–198.
[20]. McMillan K.L. A technique of state space search based on unfolding. Form. Methods Syst. Des. 6(1), 1995, pp. 45–65.
[21]. Heljanko K. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. Fundamenta Informaticae 37(3), 1999, pp. 247–268.
[22]. Khomenko V., Koutny M. Branching processes of high-level Petri nets. In Garavel, H., Hatcliff, J., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2619 of Lecture Notes in Computer Science. Springer, 2003, pp. 458–472.
[23]. Langerak R., Brinksma E. A complete finite prefix for process algebra. Computer Aided Verification, Springer, 1999, pp. 184–195.
[24]. Khomenko V., Koutny M., Vogler W. Canonical prefixes of Petri net unfoldings. Acta Informatica 40(2), 2003, pp. 95–118.
[25]. Khomenko V. Model Checking Based on Prefixes of Petri Net Unfoldings. Ph.D. Thesis, School of Computing Science, Newcastle University, 2003.
[26]. Esparza J., Heljanko K. Unfoldings: a partial-order approach to model checking. Springer, 2008.
[27]. Engelfriet J. Branching processes of Petri nets. Acta Informatica 28(6), 1991, pp.575–591.
[28]. Baldan P., Corradini A., Knig B., Schwoon S. Mcmillans complete prefix for contextual nets. Jensen, K., Aalst, W.M., Billington, J., eds.: Transactions on Petri Nets and Other Models of Concurrency I. Volume 5100 of Lecture Notes in Computer Science. Springer, 2008, pp. 199–220.
[29]. Fleischhack H., Stehno C. Computing a finite prefix of a time Petri net. Esparza J., Lakos C., eds.: Application and Theory of Petri Nets 2002. Volume 2360 of Lecture Notes in Computer Science. Springer, 2002, pp. 163–181.
[30]. Mascheroni, M. Hypernets: a Class of Hierarchical Petri Nets. Ph.D. Thesis, Facolt di Scienze Naturali Fisiche e Naturali, Dipartimento di Informatica Sistemistica e Comunicazione, Universit`a Degli Studi Di Milano Bicocca, 2010.
[31]. Jensen K., Kristensen L.M. Coloured Petri nets: modelling and validation of concurrent systems. Springer, 2009.

Федотов А.Н. Метод оценки эксплуатируемости программных дефектов. *Труды ИСП РАН*, 2016, том 28, вып. 4, с. 137-148.

Метод оценки эксплуатируемости программных дефектов

А.Н. Федотов <fedotoff@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье рассматривается метод оценки эксплуатируемости программных дефектов. Применение данного подхода позволяет осуществить приоритезацию найденных ошибок в программах. Это даёт возможность разработчику программного обеспечения исправлять ошибки, которые представляют наибольшую угрозу безопасности в первую очередь. Метод представляет собой совместное применение предварительной классификации аварийных завершений и автоматическую генерацию эксплойтов. Предварительная классификация используется для фильтрации неэксплуатируемых ошибок. Если аварийное завершение считается потенциально эксплуатируемым, то выбирается соответствующий алгоритм генерации эксплойта. В случае успешной генерации эксплойта происходит проверка работоспособности посредством эксплуатации анализируемой программы в эмуляторе. Для поиска программных дефектов можно использовать различные методы. В качестве таких методов можно выделить фаззинг и активно развивающийся в настоящее время подход к поиску ошибок на основе динамического символьного выполнения. Главным требованием для использования предлагаемого метода является возможность получения входных данных, на которых проявляется найденный дефект. Разработанный подход применяется к бинарным файлам программ и не требует дополнительной отладочной информации. Реализация метода представляет собой совокупность программных средств, которые связаны между собой управляющими скриптами. Метод предварительной классификации и метод автоматической генерации эксплойтов реализованы в виде отдельных программных средств, которые могут работать независимо друг от друга. Метод был апробирован на анализе 274 аварийных завершений, полученных в результате фаззинга. В результате анализа удалось обнаружить 13 эксплуатируемых дефектов, для которых в последствии успешно сгенерированы работоспособные эксплойты.

Ключевые слова: уязвимость; переполнение буфера; символьное выполнение; эксплойт; бинарный код.

DOI: 10.15514/ISPRAS-2016-28(4)-8

Для цитирования: А.Н. Федотов. Метод оценки эксплуатирумости программных дефектов. Труды ИСП РАН, том28, вып. 4, 2016 г., стр. 137-148. DOI: 10.15514/ISPRAS-2016-28(4)-8

1. Введение

В настоящее время проблема безопасности программного обеспечения становиться всё более актуальной. Растёт количество программных продуктов, которые применяются в различных отраслях промышленности. Кроме того, повседневная жизнь человека тесно связана с использованием программ. Современные бытовые приборы, телефоны, автомобили используют в своей работе программное обеспечение, которое может иметь выход в интернет. Сбои в работе могут приводить к серьёзным последствиям, а эксплуатация уязвимостей может нанести непоправимый ущерб. Крупные корпорации, такие как Google, Apple, Microsoft, Cisco и т.д. уделяют большое внимание разработке средств, которые обеспечивают поиск дефектов, а также внедряют их в цикл разработки ПО.

Существуют различные подходы к поиску ошибок, которые применяются в цикле разработке программного обеспечения. Одним из таких подходов является фаззинг. Главным преимуществом фаззинга – является возможность получения входных данных для воспроизведения ошибки. Современные фаззеры находят достаточно большое количество аварийных завершений во время своей работы [1,2,3,4]. Кроме того, в настоящее время перспективным направлением поиска ошибок является динамическое символьное выполнение, которое также как и фаззинг способно обнаруживать ошибки и снабжать входными данными, на которых эти ошибки проявляются. Важно отметить, что эти подходы применяются к бинарному коду, что позволяет искать ошибки в программах, для которых отсутствуют исходные тексты. Для эффективного исправления ошибок разработчиками необходимо обеспечить приоритизацию найденных дефектов. Наиболее высокий приоритет следует назначить ошибкам, которые могут привести к возникновению уязвимостей, приводящих к выполнению произвольного кода. Исправление таких ошибок в первую очередь, позволит повысить безопасность исполняемого кода.

Наличие эксплойта, который активирует уязвимость – является однозначным подтверждением её присутствия в программе. Существующие подходы к автоматической генерации экслойтов, описанные в работах [5-7] позволяют без особого вмешательства аналитика получить входные данные для активации уязвимости. Кроме того, работы [5,6] помимо эксплуатации обеспечивают и поиск уязвимости. В основе этих подходов лежит динамическое символьное выполнение. При анализе большого количества дефектов, использование высокотехнологичных видов анализа, таких как динамическое символьное выполнение, влечёт за собой рост времени анализа и потребления ресурсов. Далеко не каждый дефект является уязвимостью, приводящей к выполнению произвольного кода. Поэтому, прежде чем использовать ресурсоёмкий анализ, стоит произвести легковесную предварительную фильтрацию ошибок, с целью исключить заведомо неэксплуатируемые ошибки. В качестве таких неэксплуатируемых ошибок можно выделить деление на ноль, разыменование нулевого указателя и т.д. В статье предлагается подход, который совмещает в

себе предварительную классификацию ошибок и автоматическую генерацию эксплойта. За основу предварительной классификации дефектов используется подход представленный в инструменте с открытым исходным кодом !exploitable [8]. В качестве метода генерации эксплойтов был выбран метод, описанный в работе [9]. Классифицируемый дефект будет иметь максимальный приоритет, если ошибка считается потенциально эксплуатируемой после предварительной стадии, и затем впоследствии удалось сгенерировать работоспособный эксплойт.

Статья организована следующим образом. Во втором разделе рассмотрены существующие инструменты, которые применяются для оценки эксплуатируемости ошибок. В третьем разделе представлен метод оценки эксплуатируемости программных дефектов. В четвёртом разделе приведены основные результаты применения метода. В последнем, шестом разделе рассматриваются полученные результаты и обсуждаются перспективные направления исследований.

2. Обзор существующих подходов к оценке эксплуатируемости программых дефектов

В качестве основных подходов к оценке эксплуатируемости программных дефектов можно выделить два направления: анализ аварийных завершений и автоматическая генерация эксплойта с помощью динамического символьного выполнения.

2.1 Анализ аварийных завершений

Во время анализа аварийных завершений изучается состояние программы (значения регистров, памяти, стек вызовов) на момент срабатывания исключения. К этому состоянию применяется набор правил, каждое правило которого описывает класс аварийного завершения. В качестве примера такого класса можно привести исключение, вырабатываемое процессором при попытке выполнить инструкцию по адресу, доступ к которому запрещён, а также срабатывание защиты от переполнения буфера, вовремя вызова безопасных функций копирования. Каждый класс имеет ранг, чем ниже ранг тем, тем выше вероятность, что нарушителю удастся проэксплуатировать лежащую в основе уязвимость. Кроме того, каждый класс принадлежит одной из четырёх групп: эксплуатируемые, возможно эксплуатируемые, возможно неэксплуатируемые и недостаточно изученные. В последний класс попадают аварийные завершения, при анализе которых было слишком мало информации, для того чтобы сделать выводы об эксплуатируемости данного аварийного завершения. Например, если единственное, что удалось узнать при возникновении исключения, что это исключение возникло при нарушении доступа к памяти, то основываясь только на этой информации невозможно сделать вывод об эксплуатируемости потенциальной уязвимости. Одному

аварийному завершению может соответствовать несколько классов. Вывод об эксплуатируемости производится на основе информации о классе, ранг которого минимальный.

Реализация описанного подхода представлена в инструментах [8, 10]. Инструменты являются схожими и имеют один и тот же набор правил. Инструмент [8] анализирует программы, работающие под управлением операционных систем семейства Windows, а инструмент [10] работает с Linux-программами. Одним из достоинств данного подхода стоит отметить скорость анализа. Высокая скорость работы достигается благодаря тому, что проводится анализ только состояния программы в момент аварийного завершения, а не исследуется всё выполнение программы. Как следствие, от этого могут возникнуть ложные срабатывания. Существует вероятность, что неэксплуатируемые ошибки оцениваются как эксплуатрируемые, но ошибки, распознанные как неэксплуатируемые, как правило, таковыми и являются. Поэтому данный метод хорошо подходит для фильтрации неэксплуатируемых дефектов.

2.2 Автоматическая генерация эксплойтов

Метод автоматической генерации эксплойтов позволяет получить набор входных данных, который активирует существующую уязвимость в программе. Таким образом, можно утверждать, что уязвимость является эксплутируемой. Данный метод основывается на использовании динамического символьного выполнения. Во время динамического символьного выполнения конкретные значения переменных, которые участвуют в обработке входных данных, заменяются символьными значениями. В качестве входных данных используются различные источники: аргументы командной строки, переменные окружения, файлы и сеть. Операции над символьными значениями представляются в виде формул для SMT-решателя. Каждое ветвление в программе, результат которого зависит от символьных данных, добавляет ограничение в общую систему. Это ограничение отражает прохождение управления по конкретной ветви программы. Набор таких ограничений называется предикатом пути. Предикат пути описывает одно выполнение программы. Если данный набор ограничений предоставить на вход SMT-решателю, то результатом его работы будет набор входных данных, для активации того сценария работы программы, на котором был получен набор ограничений.

Для успешной эксплуатации уязвимости, необходимо описать дополнительные ограничения, которые позволят её активировать. Формализация эксплуатации уязвимости является сама по себе отдельной сложной задачей. Для некоторых видов уязвимостей эта задача уже решена и описана в работах [5,7,9,11,12]. В качестве таких уязвимостей выступают переполнение буфера на стеке и уязвимость форматной строки.

Существующие инструменты автоматической генерации эксплойтов имеют несколько различий между собой. Инструменты [7,9,11] используют полносистемную эмуляцию, что позволяет анализировать не только пользовательские приложения, но и программы уровня ядра операционной системы, тогда как инструменты [5,6] анализируют только пользовательские программы. Инструменты [5,6] кроме эксплуатации осуществляют ещё и поиск уязвимости, но, к сожалению, эти инструменты недоступны. Важным достоинством инструментов является, то, что участие аналитика в их работе является минимальным. В качестве недостатков можно отметить значительный рост потребления ресурсов, при анализе больших программ. Таким образом, важно обеспечить правильную предварительную фильтрацию и классификацию дефектов, для обеспечения производительности.

3. Разработанный метод

Разработанный метод представляет собой совместное применение анализа аварийных завершений и автоматической генерации эксплойта. Метод разделён на три этапа (рис.1).

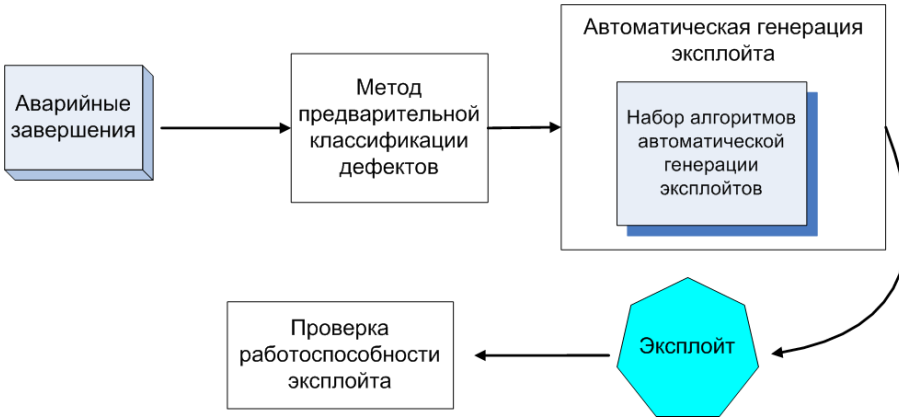


Рис. 1. Декомпозиция метода на три этапа

Fig. 1. The method decomposition into three stages

Анализ аварийных завершений применяется для осуществления предварительной классификации ошибок, с целью отфильтровать заранее неэксплуатируемые дефекты, такие как деление на ноль, разыменование нулевого указателя, срабатывание защиты во время копирования с использованием безопасных функций. Кроме того, предварительная классификация позволяет выбрать алгоритм для дальнейшей генерации эксплойта. Например, если после анализа аварийных завершений стало известно, что произошло исключение при попытке выполнить инструкцию по адресу, доступ к которому запрещён, а также существует повреждение стека, то для генерации эксплойта следует запустить алгоритм генерации эксплойта для

уязвимости переполнения буфера на стеке. После успешной генерации следует этап проверки работоспособности эксплойта. В случае успешной работы экспойта, можно однозначно утверждать о наличии эксплуатируемого дефекта в программе.

За основу метода предварительной классификации был выбран подход, реализованный в инструментах [8,10]. Для успешной интеграции метода потребовалось переработать ранги и группы для некоторых классов аварийных завершений. Так как в реализации инструмента некоторые классы аварийных завершений не подлежат эксплуатации в рамках текущей реализации автоматической эксплуатации. В качестве примера можно привести классы аварийных завершений, связанных переполнением буфера на куче. В настоящий момент автоматическая эксплуатация уязвимости переполнения буфера на куче отсутствует, поэтому эти классы перенесены в группу возможно не эксплуатируемых аварийных завершений. Также в эту группу перенесён класс, который отвечает за срабатывание защитных механизмов: таких как "канарейка" и безопасные функции работы со строками. Классы, отвечающие за исключения, возникающие при чтении из памяти или записи в память, перенесены в группу возможно эксплуатируемых, т.к. реализован подход к исправлению перезаписанных указателей во время переполнения буфера.

Проблема перезаписанных указателей возникает, если после перезаписи указателя осуществляется к нему доступ, в результате которого возникает исключение, которое, в свою очередь приводит к досрочному аварийному завершению. Технология исправления перезаписанных указателей базируется на методе, который описан в работе [9]. Этот метод использует динамическое символьное выполнение по бинарным трассам программ. В предлагаемом подходе происходит построение предиката пути от точки получения входных данных до точки срабатывания исключения доступа к памяти. Кроме этого добавляется ограничение на то, что указатель равен адресу, доступ по которому не вызывает исключения. Примером такого адреса может быть адрес из диапазона адресов секции данных анализируемого исполняемого файла. Результатом решения получается набор входных данных, который проходит по тому же пути и не вызывает срабатывания исключения. Затем этот набор входных данных вновь подвергается предварительной классификации, в результате которой возможны три случая. Первый случай, когда аварийное завершение классифицируется как эксплуатируемое, например, переход по неправильному адресу после выполнения инструкции передачи управления. В таком случае, будет происходить генерация эксплойта с учётом предыдущих ограничений на указатели. Второй случай возникает, если ещё срабатывает исключение при доступе к другому указателю. В этом случае процесс с исправлением указателя повторяется. Третий случай возникает, тогда, когда после исправления указателя аварийное завершение классифицируется как возможно неэксплуатируемое. Таким образом, обеспечивается исправление перезаписанных указателей. Также стоит отметить, что существуют класс

аварийных завершений, при котором перезаписан указатель стека в момент вызова инструкции возврата из функции. Исправление этой ситуации идентично с подходом к исправлению перезаписанных указателей. В качестве значения указателя стека используется значение, которое было на момент вызова текущей функции.

Применение вышеописанных методов позволяет произвести оценку эксплуатируемости программных дефектов.

4. Реализация метода и результаты практического применения

Предложенный метод был реализован в виде совместного использования нескольких программных средств. Метод предварительной классификации был реализован на основе утилиты gdb_exploitable [10]. Для автоматической генерации эксплойта был использован модуль-расширение из среды анализа бинарного кода[9]. Для проверки работоспособности использовался полносистемный эмулятор Qemu[13]. Механизм обеспечения совместной работы всех программных средств представлен в виде скриптов на языке Python.

Для получения набора аварийных завершений был использован подход, описанный в работе [5]. На основе этого подхода был реализован фаззер утилит, которые используют интерфейс командной строки. В качестве объекта для фаззинга были выбраны приложения из дистрибутива Debian6.0.10. В результате фаззинга были обнаружены 274 приложения, для которых есть входные данные, приводящие к аварийному завершению. Для этих приложений проводилась оценка найденных дефектов. Результаты предварительной классификации представлены в табл.1.

Табл. 1. Результаты предварительной классификации

Table 1. Results of the preliminary classification

Группа аварийных завершений	Класс аварийных завершений	Количество аварийных завершений
эксплуатируемые	Исключение при доступе к памяти, адрес которой совпадает со счётчиком команд	13
Возможно эксплуатируемые	Переполнение буфера на куче	23
Не достаточно изученные	Нарушение доступа	238

Далее для группы эксплуатируемых аварийных завершений применялась автоматическая генерация эксплойта. Так как класс аварийных завершений

соответствует ситуации, когда происходит передача управления по перезаписанному адресу во время выхода из функции, то применялся алгоритм автоматической генерации эксплойта для переполнения буфера на стеке. Для данных аварийных завершений были сгенерированы эксплойты. К сожалению, для успешной эксплуатации уязвимостей пришлось отключить рандомизацию адресного пространства, защита выполнения данных была отключена по умолчанию.

Так как нет алгоритма генерации эксплойтов для переполнения буфера на куче, данный класс аварийных завершений относится к группе возможно неэксплуатируемых дефектов. Для большинства аварийных завершений, единственной информацией, которую удалось извлечь, оказалась информация о том, что произошло только нарушение доступа. Основываясь только на этой информации невозможно сделать вывод о эксплуатируемости данного дефекта.

На тестирование всего набора примеров было потрачено примерно 10 часов. Среднее время проведения предварительной эксплуатации составило 1 минуту. Среднее время генерации эксплойта с последующей проверкой составило 21 минуту. Таким образом, использование предварительной классификации дефектов позволяет существенно сократить время, потраченное на оценку эксплуатируемости дефектов.

5. Заключение

В статье представлен метод оценки эксплуатируемости программных дефектов. Метод основан на совместном использовании предварительной классификации аварийных завершений и автоматической генерации эксплойта. Реализация метода представляет собой набор программных средств связанных между собой управляющими скриптами. Данный метод применяется непосредственно к исполняемым файлам. Применение этого метода позволяет разработчикам программного обеспечения понять, какие ошибки представляют наибольшую угрозу для безопасности ПО. Предварительная классификация позволяет отсеять неэксплуатируемые дефекты, а для потенциально эксплуатируемых запустить нужный алгоритм автоматической генерации эксплойтов. Представленные в данной работе результаты предлагают законченный метод, позволяющий оценивать эксплуатируемость найденных ошибок.

В качестве дальнейших работ стоит выделить автоматическую генерацию для других типов программных дефектов, например переполнение буфера на куче. В современных аллокаторах из библиотеки glibc, присутствует большое количество разного рода проверок, которые значительно усложняют эксплуатацию уязвимостей переполнения буфера на куче. Также отдельный интерес представляют ситуации, когда нарушитель может контролировать адрес, по которому происходит запись и данные, которые будут записаны по контролируемому адресу. Такие ситуации открывают массу возможностей для нарушителя: перезапись адреса возврата из функции или указателя на функцию

адресом в памяти, где располагается код полезной нагрузки. Эксплуатация такого рода уязвимостей позволяет обойти защитный механизм "канарейка", который призван защищать адрес возврата из функции от перезаписи во время переполнения буфера на стеке. Важным направлением исследования, является эксплуатация уязвимостей в рамках работы современных защитных механизмов уровня операционной системы. В качестве таких защит можно выделить DEP и ASLR. Для эксплуатации уязвимостей при включённых защитах следует использовать ROP- компиляцию, которая описана в работе [14].

Список литературы

[1]. Miller C. et al. Crash analysis with BitBlaze. At BlackHat USA, 2010.
[2]. American fuzzy lop fuzer. URL: http://lcamtuf.coredump.cx/afl//.
[3]. Peach fuzzer. URL: http://www.peachfuzzer.com/
[4]. Codenomicon fuzzer. URL: http://www.codenomicon.com/
[5]. T. Avgerinos, S. K. Cha, Alexandre Rebert, Edard J. Schwartz, Maverick Woo, and D.Brumley. AEG: Automatic exploit generation. *Commun. ACM*, 2014, №2.
[6]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code. IEEE Symposium on Security and Privacy, 2012
[7]. Huang S. K. et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations.Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on. IEEE, 2012, pp. 78-87.
[8]. Инструмент !exploitable. URL: https://msecdbg.codeplex.com/.
[9]. Падарян В. А., Каушан В. В., Федотов А. Н. Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке. Труды ИСП РАН, т. 26, вып. 3, стр. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7.
[10]. Плагин exploitable для gdb. URL: https://github.com/jfoote/exploitable.
[11]. Вахрушев И. А. и др. Метод поиска уязвимости форматной строки //Труды Института системного программирования РАН, т. 27, вып. 4, 2015, стр. 23-38. DOI: 10.15514/ISPRAS-2015-27(4)-2.
[12]. Heelan S. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2009.
[13]. Эмулятор Qemu. URL: http://wiki.qemu.org/Main_Page.
[14]. Schwartz E. J., Avgerinos T., Brumley D. Q: Exploit Hardening Made Easy //USENIX Security Symposium, 2011, pp. 25-41.

Method for exploitability estimation of program bugs

A.N. Fedotov <fedotoff@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn str., Moscow, Russia, 109004

Abstract. The method for exploitability estimation of program bugs is presented. Using this technique allows to prioritize software bugs that were found. Thus, it gives an opportunity for a developer to fix bugs, which are most security critical at first. The method is based on combining preliminary classification of program bugs and automatic exploit generation. Preliminary classification is used to filter non-exploitable software defects. For potentially exploitable bugs corresponding exploit generation algorithm is chosen. In case of a successful exploit generation the operability of exploit is checked in program emulator. There are various ways that used for finding software bugs. Fuzzing and dynamic symbolic execution are often used for this purpose. The main requirement for the use of the proposed method is an opportunity to get input data, which cause program to crash. The technique could be applied to program binaries and does not require debug information. Implementation of the method is a set of software tools, which are interconnected with control scripts. The preliminary classification method and automatic exploit generation method are implemented as stand-alone tools, and could be used separately. The technique was used to analyze 274 program crashes, which were obtained by fuzzing. The analysis managed to detect 13 exploitable bugs, for which successfully workable exploits were generated.

Keywords: vulnerability; buffer overflow; symbolic execution; exploit; binary code.

DOI: 10.15514/ISPRAS-2016-28(4)-8

For citation: A.N. Fedotov. Method for exploitability estimation of program bugs. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016. pp. 137-148 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-8

References

[1]. Miller C. et al. Crash analysis with BitBlaze. At BlackHat USA, 2010.
[2]. American fuzzy lop fuzer. URL: http://lcamtuf.coredump.cx/afl//.
[3]. Peach fuzzer. URL: http://www.peachfuzzer.com/
[4]. Codenomicon fuzzer. URL: http://www.codenomicon.com/
[5]. T. Avgerinos, S. K. Cha, Alexandre Rebert, Edard J. Schwartz, Maverick Woo, and D.Brumley. AEG: Automatic exploit generation. *Commun. ACM,* 2014, №2.
[6]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code. IEEE Symposium on Security and Privacy, 2012
[7]. Huang S. K. et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations.Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on. IEEE, 2012, pp. 78-87.

[8]. !exploitable. URL: https://msecdbg.codeplex.com/.

[9]. Padaryan V.A., Kaushan V.V., Fedotov A.N.[Automated exploit generaton method for stack buffer overflow vulnerabilities]. *Trudy ISP RAN/Proc. ISP RAS*, vol. 26, issue 3, 2014, pp. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7.

[10]. Exploitable plugin for gdb. URL: https://github.com/jfoote/exploitable.

[11]. Vakhrushev I. A. et al. [Search method for format string vulnerabilities]. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 4, pp. 23-38. DOI: 10.15514/ISPRAS-2015-27(4)-2.

[12]. Heelan S. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2009.

[13]. Qemu. URL: http://wiki.qemu.org/Main_Page.

[14]. Schwartz E. J., Avgerinos T., Brumley D. Q: Exploit Hardening Made Easy //USENIX Security Symposium, pp. 25-41, 2011.

Поиск ошибок доступа к буферу в программах на языке C/C++

[1,2] *И.А. Дудина <eupharina@ispras.ru >*
[1] *В.К. Кошелев <vedun@ispras.ru >*
[1] *А.Е. Бородин <alexey.borodin@ispras.ru >*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. В статье рассматривается алгоритм статического анализа для поиска в исходном коде программы ошибок доступа к буферу. Алгоритм использует символьное исполнение с объединением состояний и является чувствительным к путям. Рассматриваются только обращения к буферам, имеющим известный в момент компиляции размер и размещённым в статической памяти либо на стеке. В работе приведено формальное определение ошибки доступа к буферу, возникающей при прохождении некоторой последовательности рёбер графа потока управления программы. Приведён алгоритм, позволяющий для переменных программы суммировать информацию о возможных значениях по всем путям с учётом совместности условий переходов, взаимосвязи переменных через арифметические операции, инструкции преобразования типов, бинарные отношения в условиях переходов. Для инструкций доступа к буферу с помощью вычисленной для переменной индекса информации о возможных значениях вычисляются достаточные условия выхода за границы. Выполнимость достаточных условий проверяется SMT-решателем и, в случае нахождения модели, с её помощью обнаруживается ошибочный путь и выдаётся предупреждение. На основе данного подхода в инструменте статического анализа Svace был реализован межпроцедурный чувствительный к путям детектор ошибок доступа к буферу, способный обнаруживать новые, не покрытые предыдущими реализациями детекторов типы ошибок.

Ключевые слова: статический анализ; поиск дефектов; переполнение буфера; чувствительность к путям; символьное исполнение.

DOI: 10.15514/ISPRAS-2016-28(4)-9

Для цитирования: Дудина И.А., Кошелев В.К., Бородин А.Е. Поиск ошибок доступа к буферу в программах на языке C/C++. Труды ИСП РАН, том 28, вып. 4, 2016, стр. 149-168. DOI: 10.15514/ISPRAS-2016-28(4)-9

1. Введение

В последнее время автоматический поиск дефектов в программном коде всё чаще становится неотъемлемой частью процесса разработки современного программного обеспечения. Одним из подходов к решению этой задачи является статический анализ исходного кода, не предполагающий запуск анализируемой программы и позволяющий найти дефекты даже на не покрытых при тестировании путях.

Одним из наиболее распространенных типов дефектов в программах на языках C и C++ являются ошибки доступа к буферу [1]. Они возникают в том случае, когда обращение к буферу происходит по индексу, выходящему за его границы, т.е. происходит доступ (чтение или запись) к памяти вне данного буфера. Такое поведение может привести к падению программы, неправильной работе, в некоторых случаях к появлению эксплуатируемой уязвимости [2].

Задача поиска дефектов такого рода в общем случае является алгоритмически неразрешимой (т.к. к ней сводится задача останова [3]), т.е. идеальный (выдающий предупреждения обо всех реальных дефектах и только о них) анализатор построить нельзя. Поэтому сценарий использования анализатора определяет конкретные требования к нему: уровень полноты анализа, степень масштабируемости, ограничение на количество потребляемых ресурсов и время анализа, приемлемую долю ложных срабатываний. Совокупность этих требований в свою очередь определяет подходящие для данного сценария методы анализа.

Задачей данной работы является разработка алгоритма поиска ошибок доступа к буферу, удовлетворяющего следующим требованиям. Во-первых, он должен хорошо масштабироваться, т.к. необходимо анализировать большие программные системы (объем исходного кода исчисляется миллионами строк) за время ночной сборки (4-6 часов). Во-вторых, необходимо обеспечить высокий уровень истинных предупреждений (не менее 50-70%, в противном случае затраты на разбор ложных предупреждений нивелируют пользу автоматического поиска ошибок), при этом каждое выданное предупреждение должно сопровождаться достаточной аргументацией возникших подозрений о наличии ошибки для пользователя.

Разрабатываемый алгоритм предлагается реализовать в рамках инструмента статического анализа Svace [4], разрабатываемого в ИСП РАН. Уже существующий в Svace детектор ошибок доступа к буферу выдаёт предупреждения, если будет найдено некоторое ребро графа потока управления такое, что все проходящие через него пути содержат ошибку. В примере на рис. 1 этому свойству удовлетворяет пунктирное ребро – на любом проходящем через него пути произойдет доступ к буферу размера 7 по индексу 7.

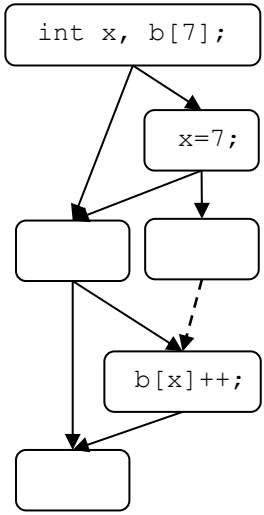


Рис. 1 Пример ошибки

Fig. 1. An example of error

К сожалению, существуют ошибочные ситуации, которые не удовлетворяют этому критерию. В качестве иллюстрации этого тезиса рассмотрим пример, изображенный на рис. 2.

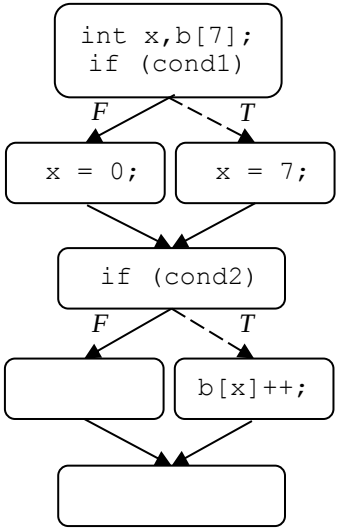


Рис. 2 Пример ошибки

Fig 2. An example of error

Если `cond1` и `cond2` могут одновременно принимать значение “истина”, то такая программа содержит ошибку доступа к буферу. В данном графе не существует единственного ребра, прохождение через которое гарантирует возникновение ошибки (для каждого ребра существует безошибочный путь через него). Тем не менее, можно предъявить такую последовательность рёбер (на рисунке выделены пунктиром), что любой путь, включающий эту последовательность, будет содержать ошибку. Ещё раз подчеркнём, что для выдачи предупреждения в такой ситуации необходимо прежде проанализировать совместность условий `cond1` и `cond2`.

Чтобы обнаруживать такие дефекты, при этом не превышая допустимое количество ложных срабатываний, необходимо реализовать чувствительный к путям анализ. Одним из методов решения этой задачи является символьное исполнение с объединением состояний [5]. Такой подход позволяет сводить задачу перебора путей к задаче определения выполнимости булевых формул, что позволяет сохранять масштабируемость анализа.

Дальнейшее изложение организовано следующим образом. Во второй части приведены априорные предположения об анализируемых программах, в рамках которых производятся построения алгоритма; дано формальное определение ошибки доступа к буферу; описана необходимая базовая инфраструктура ядра анализатора. В третьей части описан разработанный алгоритм, основанный на вычислении достаточных условий наличия ошибки, приведён алгоритм построения таких условий. Четвертая часть содержит описание реализации данного алгоритма в рамках анализатора Svace.

2. Постановка задачи

Необходимо разработать критерий ошибочной ситуации; разработать и реализовать алгоритм обнаружения таких ситуаций, удовлетворяющий ограничениям на количество ложных срабатываний.

2.1. Определение ошибки доступа к буферу

В данной работе рассматриваются только обращения к буферам, имеющим константный (т.е. известный в момент компиляции) размер и размещённым в статической памяти либо на стеке. Такой подход был выбран в качестве первого приближения к решению задачи т.к., с одной стороны, позволяет найти значительную часть ошибок доступа к буферу, а с другой – предложенные методы могут быть впоследствии доработаны для организации поиска ошибок более общего вида, в т.ч. анализа доступа к динамически выделенной памяти. Кроме этого, ошибки доступа к статическим массивам в некоторых ситуациях могут являться источником уязвимости [2].

При реализации конкретной функции разработчик как правило стремится к тому, чтобы её поведение было корректным во всех потенциальных контекстах вызова (а не только в реально существующих в проекте на данный момент). Это

требование в первую очередь важно для библиотечных функций, функций, которые ещё будут использоваться при разработке нового кода. Поэтому, чтобы обнаруживать дефекты, возникающие в потенциальных контекстах вызова, при проведении анализа каждая функция считается точкой входа в программу. Далее будем считать, что при анализе рассматривается некоторая функция вместе со всеми вызываемыми ею, вызываемыми вызываемыми ею и т.д.

Сценарий использования разрабатываемого анализатора предполагает анализ проектов при отсутствии части исходного кода (например, реализаций библиотечных функций, кода пользовательских расширений). Кроме этого, анализатор должен быть полностью автоматическим, а значит не предполагается предоставление никакой дополнительной информации от разработчиков. В данном случае при анализе функции подразумеваемое программистом предусловие неизвестно (с учетом того, что в рассмотрение включены в том числе отсутствующие в программе потенциальные контексты вызова). Вытекающая из этого сложность анализа заключается в том, что на значения, получаемые из неизвестных функций или передаваемые в функцию в качестве аргументов, могут быть наложены неизвестные анализу ограничения, например, программисты могут подразумевать существование некоторой взаимосвязи значений аргументов функции. Полное игнорирование возможности наличия таких взаимосвязей приведет к выдаче чрезмерного количества ложных срабатываний, т.к. анализатор будет сообщать об ошибках, возникающих в случае их нарушения, что нежелательно.

Для некоторой функции будем называть *неизвестными переменными* такие приходящие из других функций (вызывающих данную и вызываемых данной) значения, параметризующие выполнение анализируемой функции. Набор значений неизвестных переменных однозначно определяет *конкретный путь* исполнения функции, т.е. путь на графе потока управления (ГПУ) и значения всех переменных, состояние памяти на каждом его ребре. Такими неизвестными переменными будут являться входные параметры, начальное на входе в функцию состояние памяти, результаты вызова неизвестной функции. Совокупность ограничений на множество значений набора таких параметров будем называть *контрактом*. Таким образом, при анализе необходимо иметь в виду всё множество возможных контрактов и выдавать предупреждения только в тех случаях, когда ошибка происходит при каждом контракте из этого множества. Какие именно контракты считать возможными решается при построении конкретной стратегии анализа. Выше уже было отмечено что выбор пустого множества в качестве множества возможных контрактов приводит к чрезмерному количеству ложных срабатываний.

С другой стороны, если считать, что возможны абсолютно любые контракты, то придется пропустить слишком много реальных дефектов. Как правило, для ошибочной функции можно подобрать искусственное предусловие, исключающее возникновение ошибки, при этом оно будет куда более строгим, чем реальное, задуманное программистом. Т.е. существование такого

искусственного, полностью “безопасного” предусловия (как правило, лишающего функцию смысла) не является препятствием для выдачи предупреждения об ошибке.

В качестве компромисса предлагается ввести априорное предположение о свойствах контрактов функций, которое определит множество возможных контрактов, и, как следствие, позволит отделить потенциально возможные ошибочные сценарии исполнения функции от предположительно запрещенных контрактом. В рамках данной работы будем считать подозрительными такие ситуации, в которых наличие ошибки доступа к буферу следует из свойств ГПУ программы, но не зависит напрямую от множества допустимых значений неизвестных переменных.

Проиллюстрируем это различие на примере на рис. 3. В функции `foo` может произойти переполнение буфера `buf`, если будет выполнено `idx` ≥ `S`. Исходя из вышесказанного, данную ситуацию не будем считать ошибочной, т.к. считаем, что такие значения `idx` запрещены контрактом функции. Заметим, что, наличие ошибки напрямую зависит от множества возможных значений `idx`. Теперь рассмотрим функцию `bar`, здесь переполнение произойдет, если будет выполнено:

$$(a \geq S - 1) \wedge (b \neq 0) \qquad (1)$$

```
1  #define S 10
2
3  int buf[S];
4
5  void foo(int idx) {
6       buf[idx]++;
7  }
8
```

```
9  int bar(int a, int b) {
10      if (a >= S-1) {
11          // …
12      }
13      if (b)
14          a++;
15      return buf[a];
16 }
```

Рис. 3 Функции с неизвестными контрактами
Рис. 3. Functions with unknown contracts

Мы будем считать такую ситуацию дефектом, т.к. ошибка следует из свойств ГПУ, а именно из наличия возможно выполнимого пути, на котором гарантированно произойдёт переполнение. При этом также контракт функции может запрещать (1), – тогда ошибки нет, т.е. наличие дефекта зависит не напрямую от множества значений неизвестных переменных, а косвенно, т.к. контракт влияет на выполнимость некоторых путей на ГПУ. Чтобы строго различить две рассмотренные в функциях `foo` и `bar` ситуации, будем считать, что контракты функций не могут влиять на выполнимость путей, т.е. контракта, запрещающего (1) не существует, тогда очевидно, что функция `bar` содержит

ошибку. Заметим, что рассмотренный для функции `foo` контракт удовлетворяет этому предположению, значит предупреждение об ошибке не будет выдано. Сформулируем описанное предположение о контрактах строго.

Пусть G – подграф межпроцедурного потока управления программы, содержащий только анализируемую функцию и всех её потомков в графе вызовов вплоть до листьев. Пусть G_k – граф G после развёртки каждого его цикла на k итераций [6]. Рассмотрим множество P всех путей графа G_k. Будем говорить, что некоторый путь $p \in P$ выполним, если существует хотя бы один соответствующий ему конкретный путь. Пусть $P' \subseteq P$ — подмножество путей графа G_k, состоящее только из тех путей, которые выполнимы при условии, что набор неизвестных переменных может принимать абсолютно любые сочетания значений. Обозначим как P'' — подмножество путей графа потока управления программы, состоящее только из тех путей, которые выполнимы, если набор неизвестных переменных удовлетворяет подразумеваемому программистом контракту. Очевидно, что $P'' \subseteq P' \subseteq P$. Наше предположение о контрактах будет заключаться в том, что эти контракты не сужают множество выполнимых путей, т.е. $P'' = P'$.

Введение такого предположения приводит нас к следующему определению ошибки:

Будем говорить, что функция содержит ошибку доступа к буферу, если в графе G_k существует путь, удовлетворяющий следующим условиям:

1. *он содержит инструкцию обращения к буферу размера S по индексу i;*
2. *на любом соответствующем конкретном пути значение переменной i перед этой инструкцией не принадлежит интервалу* $[0, S-1]$;
3. *данному пути соответствует хотя бы один конкретный путь исполнения (в предположении, что набор неизвестных переменных может принимать любые комбинации значений).*

Покажем, что если программа удовлетворяет описанному определению, то существует выполнимый путь, содержащий некорректный доступ к буферу. Необходимо убедиться, что если найденный путь выполним в случае, когда набор неизвестных переменных принимает некоторое произвольное значение, то он выполним и при условии выполнения контракта. Это напрямую следует из предположения о контрактах.

Покажем, что если в программе при любых удовлетворяющих предположению контрактах существует выполнимый путь, проходящий не более k^n раз по каждому обратному ребру цикла вложенности n и содержащий некорректный доступ к буферу, то она соответствует определению. Допустим в анализируемой программе происходит ошибка доступа к буферу на некотором конкретном пути c, не проходящем более k^n раз по каждому обратному ребру цикла вложенности n. Тогда в графе G_k существует путь $p : c \in concrete(p)$, и он, очевидно, удовлетворяет первому и третьему пункту определения.

Предположим, что для этого пути существует другой конкретный путь c': $c' \in concrete(p)$, $c' \neq c$, на котором не происходит ошибки. Значит, условие возникновения ошибки зависит от значения неизвестных переменных. Следовательно, существует контракт, который запрещает значения неизвестных переменных, задающих конкретный путь c, (запрещает только c). Такой контракт удовлетворяет предположению, т.к. p по-прежнему выполним, раз существует c'. Существование такого контракта противоречит свойствам рассматриваемой ошибки, поэтому такого пути c' не может существовать, а значит, верен и второй пункт.

Наличие ошибки, удовлетворяющей этому определению, следует из свойств графа потока управления и не зависит от множества допустимых значений неизвестных параметров.

Рассмотрение графа G_k вместо графа G приводит к тому, что игнорируются ошибки, происходящие на итерации цикла большей чем k-ая. В реализации алгоритма используется эвристика, позволяющая частично обойти эти ограничения. Она заключается в изменении семантики арифметических операций, позволяющем моделировать некоторую обобщенную итерацию цикла.

2.2. Инфраструктура анализатора

В данной работе предполагается, что рассматриваемый подход будет реализован в качестве модуля-детектора в рамках общей инфраструктуры статического анализа Svace. На уровне ядра анализатора в Svace решаются задачи построения ГПУ, поиска недостижимого кода, анализа алиасов, анализа функций, завершающих выполнение программы. Всем детекторам доступна информация о результатах этих анализов [7].

Ядром производится нумерация значений, т.е. вычисляются классы эквивалентности значений переменных, называемые идентификаторами значений [8]. Детекторы ассоциируют с идентификаторами значений вычисленные свойства программы.

Ядро проводит символьное исполнение программы с объединением состояний. При этом вычисляются необходимые условия достижимости каждой точки программы в виде формул алгебры логики, где роль переменных играют идентификаторы значений. Детекторы оповещаются о всех событиях, происходящих внутри функции. Реализация детектора заключается в описании обработчиков для этих событий. Описание интересных с точки зрения данной работы событий приводится в разделе 3.2.

Далее множество точек программы будем обозначать как $Instr$, множество идентификаторов значений как VId, необходимые условия достижимости точки $q \in Instr$ как $ReachCond(q) = c,\ c \in Cond$.

3. Поиск внутрипроцедурных срабатываний

В рамках данной работы остановимся на рассмотрении методов поиска внутрипроцедурных ошибок доступа к буферу, при этом буфер расположен на стеке анализируемой функции, либо в доступной ей статической памяти. Для обнаружения таких ошибок для каждой подходящей инструкции доступа к буферу необходимо проверить, существует ли путь на ГПУ, проходящий через данную инструкцию и удовлетворяющий пунктам 2-3 определения ошибки.

Предположим, что для любых идентификаторов значения $v, x \in VId$ в каждой точке программы $q \in Instr$ известно условие в виде формулы алгебры логики $NotLess(q, v, x)$, из которой следует, что управление пришло в точку q по некоторому пути графа потока управления, при этом на каждом соответствующем ему конкретном пути в точке q выполнено $v \geq x$ (идентификаторы значений играют роль переменных в логической формуле). Аналогичная формула $NotGreater(q, v, x)$ известна для условия $v \leq x$. Тогда для инструкции $ac \in Instr$ доступа к буферу размером $s \in VId$ по индексу $i \in VId$ достаточным условием наличия ошибки в точке ac будет являться выполнимость формулы

$$ReachCond(ac) \wedge \big(NotLess(ac, i, s) \vee NotGreater(ac, i, -1)\big) \qquad (2)$$

Если выполнено $NotLess(ac, i, s) \vee NotGreater(ac, i, -1)$, то существует путь на ГПУ, удовлетворяющий второму пункту определения. Выбор инструкции ac обеспечивает выполнение первого пункта, а т.к. одновременно выполнено $ReachCond(ac)$, то найденный путь выполним, и, следовательно, верен третий пункт определения.

Таким образом, задача поиска ошибок описанного типа сводится к вычислению как можно более слабых условий $NotLess(q, v, x)$ и $NotGreater(q, v, x)$. Для её решения для идентификатора значения v в точке q определяется значение $s \in Summary$, суммирующее информацию о значениях v по всем путям, заканчивающихся в q. Искомые условия $NotLess(q, v, x)$ и $NotGreater(q, v, x)$ будут вычисляться с помощью s.

Предлагается организовать поиск ошибок доступа к буферу в три этапа:

1. В ходе символьного исполнения для идентификаторов значений $v \in VId$ в каждой точке программы $q \in Instr$ построить частичное отображение
$$VS\colon\ Instr \times VId \longrightarrow Summary.$$
2. При обработке инструкции ac доступа к буферу `b` по индексу `i` на основе значения $VS(ac, i)$ составляется формула (2) и проверяется на выполнимость.
3. В случае, если формула выполнима, т.е. подобраны значения переменных, приводящие к переполнению, из $VS(ac, i)$ путём подстановки конкретных значений переменных извлекается конкретный путь, приводящий к ошибке, и выдается предупреждение, указывающее на этот путь.

3.1. Отображение ValueSummary

Рассмотрим подробнее что представляет из себя отображение VS и как вычислять условия $NotLess$ и $NotGreater$ с его помощью.

$$VS\colon\ Instr \times VId \longrightarrow Summary.$$

Каждое значение $s \in Summary$ содержит в себе собственный идентификатор значения, информацию о котором оно суммирует по разным путям ГПУ, т.е. если $VS(q, v) = s$, то v является собственным идентификатором s.

Для вычисления из s условий $NotLess$ и $NotGreater$ определены функции:

$$HB, LB\colon Summary \times VId\ \rightarrow Cond.$$

Для любых $x \in VId, q \in Instr$, если $VS(q, v) = s$, то $HB(s, x)$ является достаточным условием того, что существует путь на ГПУ, заканчивающийся в q, такой что для каждого соответствующего конкретного пути выполнено $v \geq x$ (соответственно $v \leq x$ для формулы $LB(s, x)$). С помощью этих формул будем вычислять условия $NotLess$ и $NotGreater$:

$$NotLess(q, v, x) = HB(VS(q, v), x),$$
$$NotGreater(q, v, x) = LB(VS(q, v), x).$$

Таким образом, задача свелась к построению отображения VS и вычислению условий $HB(s, x)$ и $LB(s, x)$ (опять, чем слабее будут эти условия, тем лучше). Рассмотрим подробнее что представляют из себя элементы множества $Summary$ и как для них строить искомые условия. Значения множества принадлежат одному из следующих типов:

$$Summary = Const \cup Assume \cup Arithm \cup Cast \cup Join.$$

1. $Const = \{\langle v, n\rangle \mid v \in VId,\ \ n \in \mathbb{Z}\}$ – определение константы.

Значения констант всегда одни и те же на всех путях, поэтому если

$$s_v = \langle v, n\rangle \in Const,$$

то из этого следует:

$$HB(s_v, x) = (v = n) \wedge (n \geq x),$$
$$LB(s_v, x) = (v = n) \wedge (n \leq x).$$

2. $Relation = \{\langle v, s_{comparand}, \square\rangle \mid id \in VId,$ $s_{comparand} \in Summary,\ \square \in \{>, \geq, <, \leq, =\}\}$ – факт истинности отношения $\square$ для пары идентификаторов значений v и $comparand$, вытекающий из перехода по условию $v \square comparand$. Элемент данного типа является результатом отображения VS идентификатора значения v в точке q в том случае, когда $VS(q, comparand) = s_{comparand}$, и у предшествующего условного оператора ветка с условием $v \square comparand$ доминирует над текущей точкой. Очевидно, что условие перехода по данной ветке гарантированно выполнено в текущей точке. Отсюда можно вывести искомые достаточные условия, рассмотрим их на примере отношения строгого сравнения. Пусть:

$$s_v = \langle v, s_{comp}, >\rangle \in Relation \mid\ s_{comparand} \in Summary$$

Тогда:

$HB(s_v, x) = (v > comp) \wedge HB(s_{comp,} x - 1)$,

$LB(s_v, x) = false$ (ничего не известно о верхней границе v).

3. $Arithm = \{\langle v, s_a, s_b, \diamondsuit\rangle \mid v \in VId,\ s_a, s_b \in Summary, \diamondsuit \in \{+, -, \times, /\}\}$ – результат арифметической операции $v = a \diamondsuit b$, для каждого из операндов которой в данной точке определено значение отображения:

$$VS(q, a) = s_a, \qquad VS(q, b) = s_b.$$

Рассмотрим вычисление достаточных условий $HB(s_v, x)$ на примере вычитания. Пусть

$$s_v = \langle v, s_a, s_b, -\rangle \in Arithm \mid s_a, s_b \in Summary.$$

Предположим, что необходимо для некоторого x доказать, что поток управления достиг данной точки по некоторому пути ГПУ, такому что на любом соответствующем ему конкретном пути выполнено $v \geq x$, т.е. $a - b \geq x$. Заметим, что для любых $a, \tilde{a}, b, \tilde{b} \in \mathbb{Z}$ верно:

$$a \geq \tilde{a} \ \wedge\ b \leq \tilde{b} \ \wedge\ \tilde{a} - \tilde{b} \geq x \quad \Rightarrow \quad a - b \geq x.$$

Следовательно, достаточно предъявить для пути ГПУ, проходящего через данную точку, два целых числа $\tilde{a}$ и $\tilde{b}$, таких что на любом соответствующем ему конкретном пути выполнена посылка импликации: $a \geq \tilde{a} \ \wedge\ b \leq \tilde{b} \ \wedge\ \tilde{a} - \tilde{b} \geq x$. Отсюда получаем формулу

$$HB(s_v, x) = (v = a - b) \wedge \left(\exists \tilde{a} \exists \tilde{b}\ LB(s_a, \tilde{a}) \wedge HB(s_b, \tilde{b}) \wedge (\tilde{a} - \tilde{b} \geq x)\right).$$

Аналогично выводится формула $LB(s_v, x)$ и такие же формулы для сложения.

Введение дополнительных переменных $\tilde{a}$ и $\tilde{b}$ здесь необходимо, т.к. заранее (до анализа совместности условий переходов) определить нижние и верхние границы значений переменных a и b невозможно, поэтому значения $\tilde{a}$ и $\tilde{b}$ определит решатель.

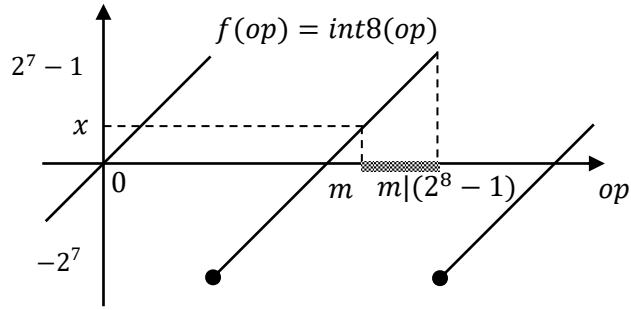


Рис. 4. Вычисление достаточного условия для нижней границы для результата инструкции приведения типа

Fig. 4. Calculating the sufficient condition for the lower BOUND for the result of the CAST instruction

4. $Cast = Trunc \cup ZExt$ – результат инструкции преобразования типов, для аргумента которой в данной точке имеет определено значение отображения $VS(q, op) = s_{op}$.

a. $ZExt = \{\langle v, s_{op}\rangle \mid v \in VId,\ s_{op} \in Summary\}$ – беззнаковое расширение значения op. Пусть

$$s_v = \langle v, s_{op}\rangle \in ZExt \mid s_{op} \in Summary.$$

Тогда:

$$HB(s_v, x) = HB(s_{op}, x),$$
$$LB(s_v, x) = LB(s_{op}, x) \wedge (x \geq 0).$$

b. $Trunc = \{\langle v, s_{op}, w\rangle \mid v \in VId,\ s_{op} \in Summary,\ w \in \mathbb{N}\}$ – приведение op к типу меньшего размера, равного w. Тогда если:

$$s_v = \langle v, s_{op}, w\rangle \in Trunc \mid s_{op} \in Summary, \qquad w \in \mathbb{N},$$

то

$$HB(s_v, x) = \exists m\ (m\&(2^b - 1) = x) \wedge HB(s_{op}, m) \wedge LB\left(s_{op}, m \middle| (2^b - 1)\right).$$

Вывод этой формулы для случая $w = 8$ поясняется на рис. 4, где показана зависимость результата инструкции приведения целочисленного значения op к однобайтному целому типу. У числа m старшие (обрезаемые) биты совпадают с соответствующими в числе op, а младшие 8 бит совпадают с младшими 8-ю битами числа x. Таким образом, $op \geq x$ тогда и только тогда, когда op принадлежит интервалу $[m, m|(2^8 - 1)]$ (на рисунке этот интервал выделен штриховкой).

Условие $\mathrm{LB}(s_v, x)$ вычисляется аналогично.

5. $Join = Single \cup Double$,

$$Single = \{\langle joinedId, \langle s_{br}, c\rangle\rangle \mid joinedId \in VId, s_{br} \in Summary, c \in Cond\},$$
$$Double = \{\langle joinedId, \langle s_l, c_l\rangle, \langle s_r, c_r\rangle\rangle \mid joinedId \in VId,\ s_l, s_r \in Summary,\ c_l, c_r \in Cond\}$$

– значение в точке слияния двух веток с условиями c_l и c_r, таких что, значение отображения определено на обоих ветках: $VS(q, l) = s_l,\ VS(q, r) = s_r$, либо только на одной $VS(q, br) = s_{br}$ (в этом случае условие этой ветки обозначается просто c). Для случая двух веток, если:

$$s_{jId} = \langle jId, \langle s_l, c_l\rangle, \langle s_r, c_r\rangle\rangle \in Join \mid s_l, s_r \in Summary,\ c_l, c_r \in Cond,$$

то

$$HB(s_{jId}, x) = \bigvee \begin{matrix}(joinedId = l) \wedge HB(s_l, x) \wedge c_l \\ (joinedId = r) \wedge HB(s_r, x) \wedge c_r\end{matrix}.$$

Достаточные условия $LB(s_{jId}, x)$ и оба вида достаточных условий для случая, когда отображение определено только на одной ветке, записываются аналогично.

В случае, если для некоторого идентификатора значения v в данной точке q значение отображения не определено $VS(q, v) = \emptyset$, то информации о возможных его значениях нет, поэтому функции HB и LB можно доопределить:

$$HB(\emptyset, x) = false, \qquad LB(\emptyset, x) = false.$$

3.2. Построение отображения ValueSummary

Перед началом символьного исполнения $VS = \emptyset$. Обновление отображения производится для следующих событий:

- $newConst(v, n)$, $v \in VId$, $n \in \mathbb{Z}$ – объявление константы.
- $binaryOp(r, a, b, \diamondsuit)$, $r, a, b \in VId$, $\diamondsuit \in \{+, -, \times, /\}$ – арифметическая операция $r = a \diamondsuit b$
- $assume(v, cmp, \square)$, $v, cmp \in VId, \square \in \{>, \geq, <, \leq, =\}$ – условие на ребре инструкции ветвления.
- $castZext(v, op)$, $v, op \in VIds$ – беззнаковое расширение.
- $castTrunc(v, op, w)$, $v, op \in VIds$, $w \in \mathbb{N}$ – приведение к типу меньшего размера, равного w.
- $join(jId, l, c_l, r, c_r)$, $jId, l, r \in VId$, $c_l, c_r \in Cond$ – слияние двух идентификаторов l и r в jId по веткам с условиями c_l и c_r.

Обновление отображения для этих событий происходит в соответствии с правилами вывода (см. рис. 5). Для прочих инструкций значения отображения для всех идентификаторов копируются с предыдущего состояния.

$$\frac{q = newConst(v, n),\quad s_v = \langle v, n\rangle \in Const}{VS \uplus \{\langle q, v\rangle \rightarrow s_v\}}$$

$$\frac{q = binaryOp(r, a, b, \diamondsuit),\quad VS(q, a) = s_a, VS(q, b) = s_b,\quad s_r = \langle r, s_a, s_b, \diamondsuit\rangle \in Arithm}{VS \uplus \{\langle q, r\rangle \rightarrow s_r\}}$$

$$\frac{q = assume(v, cmp, \square),\quad VS(q, cmp) = s_{cmp},\quad s_v = \langle v, s_{cmp}, \square\rangle \in Relation}{VS \uplus \{\langle q, v\rangle \rightarrow s_v\}}$$

$$\frac{q = castZext(v, op),\quad VS(q, op) = s_{op},\quad s_v = \langle v, s_{op}\rangle \in ZExt}{VS \uplus \{\langle q, v\rangle \rightarrow s_v}$$

$$\frac{q = castTrunc(v, op, w),\quad VS(q, v) = s_{op},\quad s_v = \langle v, s_{op}, b\rangle \in Trunc}{VS \uplus \{\langle q, v\rangle \rightarrow s_v}$$

$$\frac{q = join(jId, l, c_l, r, c_r),\quad VS(q, l) = \emptyset,\ VS(q, r) = s_r,\quad s_{jId} = \langle jId, \langle s_r, c_r\rangle\rangle \in Join}{VS \uplus \{\langle q, jId\rangle \rightarrow s_{jId}\}}$$

$$\frac{q = join(jId, l, c_l, r, c_r),\quad VS(q, l) = s_l,\quad VS(q, r) = \emptyset,\quad s_{jId} = \langle jId, \langle s_l, c_l\rangle\rangle \in Join}{VS \uplus \{\langle q, jId\rangle \rightarrow s_{jId}\}}$$

$$\frac{q = join(jId, l, c_l, r, c_r),\quad VS(q, l) = s_l, VS(q, r) = s_r,\quad s_{jId} = \langle jId, \langle s_l, c_l\rangle, \langle s_r, c_r\rangle\rangle \in Join}{VS \uplus \{\langle q, jId\rangle \rightarrow s_{jId}\}}$$

Рис. 5 Правила вывода
Fig. 5. Inference rules

По построению отображения и достаточных условий наличия ошибки, в случае если программа удовлетворяет введенным ранее предположениям и результаты проведенных ядром анализов корректны, то выполнимость формулы (2) всегда будет означать наличие дефекта в анализируемой программе.

3.3. Пример обнаружения ошибки доступа к буферу

```
1  int bar(int a, int b){
2    if (a1 >= c9){
3        // …
4    }
5    if (b)
6        a2 = a1 + c1;
7    a3 = phi(a1, a2);
8    return buf[a3];
9  }
```

Рис. 6. Пример ошибки
Fig. 6. An example of error

Рассмотрим предложенный подход на примере поиска ошибки в функции `bar` из разд. 2 (см. рис. 6). Здесь для удобства вместо исходных переменных приведены идентификаторы значений.

В ходе символьного исполнения обрабатываются следующие события, перечисленные в табл. 1.

Табл. 1. События
Table 1. Events

Стр.	Событие
2	$const(c_9, 9)$
2	$assume(a_1, c_9, \geq)$
4	$join(a_1, \begin{matrix} a_1, (a_1 \geq 9), \\ a_1, (a_1 < 9) \end{matrix})$
6	$const(c_1, 1)$
6	$binaryOp(a_2, a_1, c_1, +)$
7	$join(a_3, \begin{matrix} a_1, (b = 0), \\ a_2, (b \neq 0) \end{matrix})$
$\boldsymbol{a_1, a_2, a_3, c_1, c_9, b \in VId}$	

В результате для идентификатора a_3 перед инструкцией доступа ac: `buf`$[a_3]$ значение $s_6 = VS(ac, a_3)$ можно представить в виде графа на рис. 7.

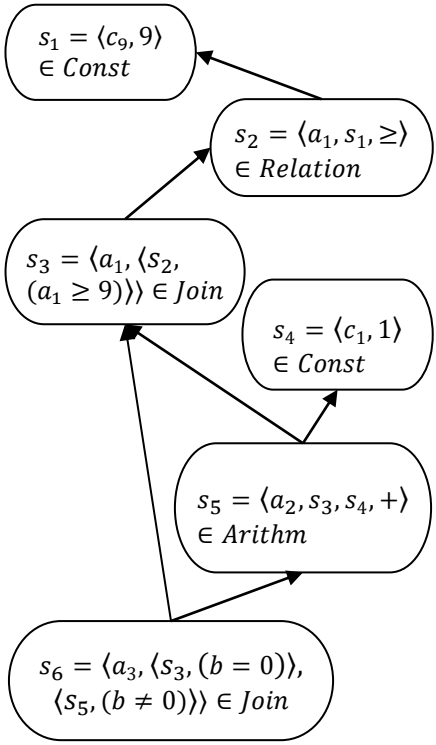


Рис.7. Граф значения $s_6 = VS(ac, a_3)$
Fig. 7. Graph of value $s_6 = VS(ac, a_3)$

Обрабатывается инструкция ac доступа к буферу на строке 8. Т.к. размер буфера равен 10, то необходимо проверить условие $a_3 \geq 10$.

$$NotLess(ac, a_3, 10) = HB(VS(ac, a_3), 10) = HB(s_6, 10).$$

Значение a_3 получилось из слияния двух значений ($s_6 = VS(ac, a_3))\rangle\rangle \in Join$), поэтому нужно проверить каждое из них с учётом условий слияния:

$$HB(s_6, 10) = \bigvee \begin{matrix}(a_3 = a_1) \wedge HB(s_3, 10) \wedge (b = 0)\\ (a_3 = a_2) \wedge HB(s_5, 10) \wedge (b \neq 0)\end{matrix}$$

Рассмотрим вторую ветвь. Значение в этой ветви является суммой двух значений ($s_5 = \langle a_2, s_3, s_4, +\rangle$). В общем случае для произвольной суммы нижние границы каждого из слагаемых неизвестны (т.к. в графе их значений могут быть узлы $Join$, а совместность условий путей еще не анализировалась), поэтому обозначим эти границы некоторыми вспомогательными переменными. Тогда искомое условие будет иметь вид:

$$HB(s_5, 10) = (a_2 = a_1 + \mathrm{c}_1) \wedge \left(\exists\widetilde{a_1}\exists\widetilde{\mathrm{c}_1}\, HB(s_3, \widetilde{a_1}) \wedge HB(s_4, \widetilde{\mathrm{c}_1}) \wedge (\widetilde{a_1} + \widetilde{\mathrm{c}_1} \geq 10)\right).$$

Т.к. c_1 это просто константа 1, то условие для неё выглядит тривиально:

$$s_4 = \langle c_1, 1\rangle \in Const \Rightarrow HB(s_4, \widetilde{\mathrm{c}_1}) = (c_1 = 1) \wedge (1 \geq \widetilde{\mathrm{c}_1}).$$

Для значения a_1 информация о значении есть только на одном из путей, сливающихся перед инструкцией на строке 13, поэтому:

$$HB(s_3, \widetilde{a_1}) = HB(ac, s_2, \widetilde{a_1}) \wedge (a_1 \geq 9).$$

Аналогично для первой ветви $HB(s_6, 10)$ можно записать:

$$HB(s_3, 10) = HB(s_2, 10) \wedge (a_1 \geq 9).$$

Про значение a_1, пришедшее с ветки истинности условия точно известно, что для него это условие выполнено, т.е. $(a_1 \geq \mathrm{c}_9)$. Очевидно, что, если $\mathrm{c}_9 \geq \widetilde{a_1}$, то $a_1 \geq \widetilde{a_1}$. Отсюда получаем:

$$HB(s_2, l') = (a_1 \geq \mathrm{c}_9) \wedge HB(ac, \mathrm{c}_9, \widetilde{a_1}) = (a_1 \geq \mathrm{c}_9) \wedge (\mathrm{c}_9 = 9) \wedge (9 \geq \widetilde{a_1}).$$

Аналогично можно развернуть оставшиеся выражения и вычислить итоговое условие. Получившаяся формула (2) будет выполнима, т.к. она верна при следующих значениях переменных:

Табл. 2. Модель для условия переполнения
Table 2. A model for condition of overflow

Переменная	c_1	c_9	$\widetilde{a_1}$	$\widetilde{\mathrm{c}_1}$	a_3	a_2	a_1	b
Значение	1	9	9	1	10	10	3	1

Чтобы получить ошибочный путь необходимо подставить полученные значения в условия в узлы типа $Join$. В результате подстановки получим, что любой путь, для которого выполнено $(a_1 \geq 9)$ и $(b \neq 0)$ будет ошибочным. В данном случае этому условию удовлетворяет единственный путь (2)-(3)-(4)-(5)-(6)-(7)-(8), и он действительно содержит ошибку доступа к буферу. В соответствующем подграфе значения s_6 на рис. 7 ребра выделены жирным.

4. Реализация детектора

На основе рассмотренного подхода в инструменте статического анализа Svace был реализован межпроцедурный путе- и контекстно-чувствительный детектор ошибок доступа к буферу. В качестве инструкций доступа к буферу рассматривались обычные инструкции индексации и вызовы библиотечных функций, осуществляющих доступ к переданному в качестве аргумента буферу (например, `memcpy`). Исходя из этого детектор выдает предупреждения двух типов: BUFFER_OVERFLOW.EX и BUFFER_OVERFLOW.LIB.EX.

В Svace и ранее имелось несколько межпроцедурных, но не чувствительных к путям детекторов выхода за границы массива. Преимуществами новой реализации, благодаря которым удается обнаружить новые типы ошибок, являются:

- чувствительность к путям, позволяющая обнаруживать ошибки, характеризующиеся последовательностью рёбер (более, чем одного);
- отслеживание взаимосвязей переменных, (включая арифметические операции, бинарные отношения между переменными в условиях перехода, значения с условиями в точках слияний), позволяющие строить цепочки из таких взаимосвязей для доказательства переполнения;
- поддержка инструкций преобразования типов (практика показала, что округление информации о результате преобразования в любую из сторон вместо тщательной обработки приводит к появлению существенного количества либо ложных срабатываний, либо пропущенных ошибок).

Кроме того, разработан эвристический алгоритм, который, используя информацию об индуктивных переменных и граничных условиях цикла, строит значения $Summary$ для переменных цикла и ищет ошибочные ситуации на основе этих значений. Детектор, разработанный на его основе, выдает предупреждения типа OVERFLOW_AFTER_CHECK.EX.

Для достижения хороших показателей кроме описанных в данной статье подходов необходима также поддержка межпроцедурного анализа, рассмотрение механизмов которого не вошло в данную работу. Поэтому, подробные результаты работы детекторов, а также сравнение с другими работами в этой области будет приведено в следующей статье. Здесь рассмотрим пример внутрипроцедурной ошибки, обнаруженной при анализе проекта Android-5.0.2. Слева на рис.8 приведён фрагмент исходного кода, а справа – трасса срабатывания детектора (последовательность событий записывается снизу-вверх). Здесь анализатор сообщает пользователю, что на некоторой итерации `ci` может равняться 2 исходя из сравнения на строке 8, и тогда, если цикл не завершится, на следующей итерации произойдет переполнение буфера `indices`.

1	`for (ci = 0; ci < folder->NumCoders;`	Array 'indices' of size 3 is accessed by 3 at line 6. This may lead to buffer overflow. • Buffer overflow at line 6. • Add: ci + 1 >= 3 at line 1. • Variable ci may be equal to 2 at line 8.
2	`      ci++) {`	
3	`  // …`	
4	`  if (folder->NumCoders == 4) {`	
5	`     UInt32 indices[] = { 3, 2, 0 };`	
6	`     si = indices[ci];`	
7	`     //…`	
8	`     if (ci == 2) {`	
9	`        //…`	
10	`     }`	
11	`  }`	
12	`}`	

Рис.8. Пример реального срабатывания
Fig. 8. An example of real operation

Список литературы

[1]. CVE and CCE Statistics Query Page. https://web.nvd.nist.gov/view/vuln/statistics
[2]. A. One, "Smashing the Stack for Fun and Profit", Phrack Magazine, Volume 7, Issue 49, November 1996.
[3]. D. Larochelle, D. Evans. Statically detecting likely buffer overflow vulnerabilities. 10th USENIX Security Symposium, Washington, D.C., August 2001.
[4]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ Труды ИСП РАН, том 26, 2014 г. стр 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7.
[5]. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. 2012. Efficient state merging in symbolic execution. SIGPLAN Not. 47, 6 (June 2012), 193-204. DOI=http://dx.doi.org/10.1145/2345156.2254088
[6]. В.К. Кошелев, И.А. Дудина, В.И. Игнатьев, А.И. Борзилов. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 59-86. DOI: 10.15514/ISPRAS-2015-27(5)-5.
[7]. А.Е. Бородин, А.А. Белеванцев. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6. 2015 г. стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.
[8]. А.Е. Бородин, Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++: дис. канд. ф.-м. наук. Москва, 2016г.

Statically detecting buffer overflows in C/C++

[1,2]I. Dudina <eupharina@ispras.ru>
[1]V. Koshelev <vedun@ispras.ru>
[1]A. Borodin < alexey.borodin@ispras.ru >
[1]ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation;
[2]CMC MSU, CMC faculty, 2 educational building,
MSU, Leninskie gory str., Moscow 119991, Russian Federation

Abstract. The paper describes a static analysis approach for buffer overflow detection in C/C++ source code. This algorithm is designed to be path-sensitive as it is based on symbolic execution with state merging. For now, it works only with buffers on stack or on static memory with compile-time known size. We propose a formal definition for buffer overflow errors that are caused by executing a particular sequence of program control-flow edges. To detect such errors, we present an algorithm for computing a summary for each program value at any program point along multiple paths. This summary includes all joined values at join points with path conditions. It also tracks value relations such as arithmetic operations, cast instructions, binary relations from constraints. For any buffer access we compute a sufficient condition for overflow using this summary for index variable and the reachability condition for the current function point. If this condition is proved to be satisfiable by an SMT-solver, we use its model given by the solver to detect error path and report the warning with this path. This approach

was implemented for Svace static analyzer as the new buffer overflow detector, and it has found a significant amount of unique true warnings that are not covered by the old buffer overflow detector implementations.

Keywords: static analysis, software error detection, buffer overflow, path-sensitivity, symbolic execution.

DOI: 10.15514/ISPRAS-2016-28(4)-9

For citation: Dudina I., Koshelev V., Borodin A. Statically detecting buffer overflows in C/C++. Trudy ISP RAN /Proc. ISP RAS, vol. 28, issue 4, 2016, pp. 149-168 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-9

References

[1]. CVE and CCE Statistics Query Page. https://web.nvd.nist.gov/view/vuln/statistics
[2]. A. One, "Smashing the Stack for Fun and Profit", Phrack Magazine, Volume 7, Issue 49, November 1996.
[3]. D. Larochelle, D. Evans. Statically detecting likely buffer overflow vulnerabilities. 10th USENIX Security Symposium, Washington, D.C., August 2001.
[4]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. Static analyzer Svace for finding of defects in program source code. Trudy ISP RAN /Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7
[5]. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. 2012. Efficient state merging in symbolic execution. SIGPLAN Not. 47, 6 (June 2012), 193-204. DOI=http://dx.doi.org/10.1145/2345156.2254088
[6]. V. Koshelev, I. Dudina, V. Ignatyev, A. Borzilov. Path-Sensitive Bug Detection Analysis of C# Program Illustrated by Null Pointer Dereference. Trudy ISP RAN /Proc. ISP RAS, 2015, vol. 27, issue 5, pp. 59-86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)- 5.
[7]. A. Borodin, A. Belevancev. A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels. Trudy ISP RAN /Proc. ISP RAS, 2015, vol. 27, issue 6, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8.
[8]. A. Borodin. PhD thesis. Interprocedural contex-sensitive static analysis for error detection in C/C++ source code. ISP RAN, Moscow, 2016

Поддержка стандарта OpenMP 4.0 для архитектуры NVIDIA PTX в компиляторе GCC[1]

А.В. Монаков <amonakov@ispras.ru>
В.А. Иванишин <vlad@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье описывается реализация стандарта OpenMP версии 4.0 для акселераторов NVIDIA PTX в компиляторе GCC. Особое внимание уделяется вопросам генерации корректного и эффективного кода для прагм OpenMP с учетом ограничений архитектуры PTX. Поскольку реализация опирается на существующую в GCC трансляцию OpenMP и интеграцию с библиотекой libgomp, для PTX реализованы вторичные программные стеки, позволяющие организовать общий для синхронной группы стек в глобальной памяти и передавать адреса на данные в таких стеках между нитями. Описывается схема организации выполнения одной OpenMP-нити в 32 синхронных потоках выполнения в PTX вне OpenMP SIMD-регионов за счет легковесной инструментации некоторых инструкций. Представлены результаты тестирования на микротестах и сравнение с реализацией стандарта OpenACC.

Ключевые слова: компиляторы, GCC, OpenMP, CUDA, PTX.

DOI: 10.15514/ISPRAS-2016-28(4)-10

Для цитирования: А.В. Монаков, В.А. Иванишин. Поддержка стандарта OpenMP 4.0 для архитектуры NVIDIA PTX в компиляторе GCC. Труды ИСП РАН, том 28, вып. 4, 2016, стр. 169-182. DOI: 10.15514/ISPRAS-2016-28(4)-10

1. Введение

Интерфейс программирования OpenMP [1] предоставляет набор расширений для языков C, C++, Fortran в виде аннотаций-прагм, с помощью которых можно описывать параллельные алгоритмы на общей памяти. Важно отметить, что результатом удаления всех прагм из корректного OpenMP-кода будет корректный последовательный код с эквивалентной семантикой. Таким образом обеспечивается возможность инкрементального переноса последовательного кода для многоядерных архитектур и поддержки как

[1] Работа поддержана грантом РФФИ 13-07-12102 офи_м.

последовательной, так и параллельной версий программы в рамках единого исходного кода.

В ревизии 4.0 стандарта OpenMP был добавлен ряд новых расширений, предназначенных для выполнения параллельных вычислений на специализированных программируемых акселераторах: x86-совместимых многоядерных процессорах Xeon Phi (Intel MIC) или графических акселераторах (для которых обычно используется интерфейс CUDA или OpenCL).

В настоящее время акселераторы, как правило, подключаются через интерфейс PCI-Express, и поэтому доступ к оперативной памяти основного процессора имеет большую задержку и меньшую пропускную способность; с другой стороны, сами акселераторы имеют специализированную оперативную память с очень высокой пропускной способностью. Чтобы дать программисту возможность эффективно ей распоряжаться, в интерфейсах CUDA и OpenCL есть ряд функций для выделения блоков памяти на акселераторе и копирования между акселератором и хост-процессором. В OpenMP порядок обмена данными между хост-системой и акселератором задается с помощью map-прагм.

Поскольку графические акселераторы не совместимы с хост-процессором по набору команд, компиляторы должны транслировать все участки кода, которые могут выполняться на акселераторе или хост-процессоре (это target-регионы, а также все функции, помеченные прагмой declare target) больше одного раза: не только для архитектуры команд хост-процессора, но и для всех поддерживаемых акселераторных архитектур.

2. Поддержка OpenMP 4.0 в компиляторе GCC

В компиляторе GCC поддержка OpenMP 4.0 была добавлена в версии 4.9 в 2014 году [2], но поддержка акселераторных платформ появилась только в версии 5 с добавлением запуска вычислений на Intel Xeon Phi через библиотеку liboffloadmic. Также в версии 5 была добавлена поддержка интерфейса OpenACC 2.0 для графических акселераторов NVIDIA; генерация кода для GPU выполняется самим GCC: для этого была добавлена новая целевая архитектура nvptx.

В качестве стабильного набора команд, который мог бы использоваться компиляторами для GPU, NVIDIA предложила абстрактный ассемблеро-подобный интерфейс PTX [3]. Таким образом достигается компромисс между необходимостью формата для хранения кода, для которого требуется совместимость с будущими акселераторами, и, с другой стороны, возможностью изменения аппаратного набора инструкций и их кодирования. В отличие от PTX, аппаратный набор команд документирован производителем лишь минимально, в виде, достаточном, например, для чтения дизассемблированного кода (однако в проектах, напрямую выдающих машинный код для акселераторов, таких как Nouveau и maxas [4], за счет обратной разработки стали известны детали кодирования инструкций).

Аналогичный подход используется в таких проектах как PNaCl, где для распространения кода используется представление LLVM IR.

Как и LLVM IR, PTX структурирован: для каждой функции задан ее прототип. Машинные регистры общего назначения не доступны непосредственно: каждая функция декларирует имена и типы регистров, используемых в ней. Таким образом, регистры PTX аналогичны скалярным неадресуемым локальным переменным в Си; соответственно, регистры не теряют свои значения при вызовах (на архитектурах без регистровых окон бинарные соглашения о вызовах для каждого регистра определяют, какая сторона ответственна за его сохранение и восстановление, вызываемая или вызывающая). Явного указателя стека нет, и в связи с этим динамическое выделение стековой памяти (для alloca в Си) не возможно. Статическое выделение стековой памяти возможно за счет объявления локальных объектов в пространстве памяти .local.

При трансляции PTX в машинный код происходит распределение регистров и множество машинно-зависимых оптимизаций: планирование команд, разворачивание циклов и другие.

В наборе команд PTX присутствуют как специализированные команды, специфичные для графических акселераторов (например, инструкция барьерной синхронизации bar.sync), так и часто используемые команды общего назначения, которые не имеют соответствующих аппаратных инструкций и раскрываются в нетривиальную последовательность при трансляции PTX (например, инструкция целочисленного деления div). Почти все инструкции указывают тип обрабатываемых данных с помощью суффикса и поддерживают условное выполнение.

Архитектура графических акселераторов оптимизирована для максимизации производительности вычислений в множестве параллельно работающих нитей; производительность однопоточного кода, запущенного на GPU, была бы невысока. Параллельные контексты выполнения на GPU образуют иерархию. В первую очередь можно выделить синхронные группы (warps), составленные из 32 нитей. Все нити группы имеют общий счетчик команд, так что на каждом шаге все нити выполняют одну и ту же инструкцию; в случаях, когда предикат условного перехода имеет разные значения в нитях группы, выполнение противоположных веток сериализовано: сначала часть нитей с одинаковым значением предиката полностью выполняет свою ветку, затем нити с противоположным значением предиката — свою; это достигается за счет сокращения маски активных нитей. После выполнения обеих веток (в непосредственном постдоминаторе ветвления) маска активных веток восстанавливается и продолжается выполнение синхронной группы. Аналогично обрабатываются вызовы по указателю.

Поскольку у программиста нет возможности вручную определять точки слияния синхронных групп (соответствующие инструкции не присутствуют в PTX и вставляются неявно при трансляции PTX-кода), это приводит к тому, что некоторые программы будут иметь взаимоблокировки. В частности, выполнить

взаимное исключение за счет цикла, выполняющего активное ожидание, невозможно.

Отметим, что на разных уровнях иерархии параллельные нити на GPU имеют разные ограничения и возможности коммуникации. Так, в пределах синхронной группы нити могут выполнять быстрый обмен содержимым регистров с помощью инструкции shfl. На следующем уровне иерархии параллельных нитей находятся блоки. В пределах блока нити имеют доступ к общему блоку быстрой памяти в пространстве .shared и могут выполнять барьерную синхронизацию с помощью инструкции bar.sync.

PTX-код определяет работу одной нити. Для определения положения текущей нити в общей иерархии доступны специальные регистры, например %laneid (номер в синхронной группе).

3. Трансляция OpenMP в GCC

Поддержка OpenMP в GCC разделена между собственно компилятором и библиотекой времени выполнения libgomp, поставляемой вместе с другими компонентами компилятора. Языковые фронт-энды C, C++, Fortran осуществляют парсинг прагм и сохранение их в составе AST-представления; далее одни из самых ранних фаз машинно-независимой трансляции производят анализ прагм при переходе к представлению GIMPLE и сведение их к универсальным конструкциям (например, вызовам функций, условным и циклическим блокам), которые могут обрабатываться последующими фазами компилятора.

В случаях, когда трансляция прагм приводит к добавлению специальных функций, либо когда на уровне исходного кода есть явные вызовы функций из OpenMP API, GCC не встраивает реализации этих функций в объектный код. Все подобные функции реализованы в рамках библиотеки libgomp. Таким образом, публичные функции библиотеки libgomp составляют два пространства имен: функции с префиксом omp_ реализуют функциональность OpenMP API, а функции с префиксом GOMP_ реализуют элементы внутреннего интерфейса между компилятором и libgomp. Например, вход в параллельный регион производится через функцию GOMP_parallel. Отдельно можно отметить функции в пространствах имен GOMP_PLUGIN_ и GOMP_OFFLOAD_, реализующие интерфейсы между основным кодом libgomp и ее модулями-плагинами для различных поддерживаемых акселераторных платформ.

Есть ровно три OpenMP-прагмы, для которых код полного выражения под прагмой переносится компилятором в отдельную функцию: это прагмы parallel, task и target, так как эти прагмы предписывают, что соответствующий код может выполняться не в рамках текущего контекста, а либо в нескольких параллельных нитях (прагма parallel), в любой нити (прагма task), или на акселераторном устройстве (прагма target). Эти функции вызываются через функции GOMP_parallel, GOMP_task и GOMP_target_ext; эти функции libgomp реализуют запуск новых нитей, распределение задач между нитями, запуск

вычислений на акселераторе. Новые функции, выделенные из пользовательского кода, принимают аргумент типа (struct omp_data_s*N* *) — указатель на структуру, содержащую указатели на переменные, которые объявлены вне блока, но используются внутри него (для переменных небольшого размера можно передавать непосредственно их значения вместо указателей, при условии, что внешняя переменная не модифицируется внутри блока: это заведомо так, например, для firstprivate-переменных).

#pragma omp parallel
 	
v = 0;
	omp_data_o.v = &v;
	GOMP_parallel(omp_fn1, &omp_data_o);
	
	void omp_fn1(omp_data_s *omp_data_i)
	{
	  *(omp_data_i->v) = 0;
	}

Рис. 1. Выделение OpenMP-региона в отдельную функцию с передачей адреса разделяемой переменной.

Fig. 1. Outlining an OpenMP region into a separate function and passing the address of a shared variable.

4. Вторичные стеки для автоматических переменных в PTX

При добавлении поддержки PTX как акселераторной архитектуры в OpenMP нами было принято решение использовать, где это оправдано, имеющиеся в GCC подходы к трансляции и функциональность libgomp. Это позволяет переиспользовать существующий код, снизить сложность компилятора и поддержки в будущей разработке. Возможны и другие подходы, когда трансляция регионов кода для акселератора выполняется компилятором иначе, чем для хост-процессора: но в этом случае поддержка всего многообразия OpenMP-прагм в target-регионах затруднена.

Как было отмечено, параллельные регионы вырезаются компилятором в отдельные функции, которые получают указатели на общие данные в структуре omp_data_i, передающейся по ссылке. Эта структура располагается на стеке нити, вызвавшей GOMP_parallel. Поскольку в PTX стековые данные располагаются в пространстве памяти .local, это приводит к тому, что другие нити не могут обратиться к этим данным.

Таким образом, реализация стеков в .local-памяти не совместима с предположением GCC, что указатели на стековые данные валидны во всех нитях (стандарт C11 оставляет это как implementation-defined поведение).

Для решения этой проблемы в GCC была реализована поддержка управляемых компилятором стеков, которые могут размещаться в пространстве памяти .global и, таким образом, допускать обмен данными между нитями. Для поддержки указателя на вершину альтернативного стека возможны два подхода:

- хранение указателя на PTX-регистрах и передача как дополнительного аргумента при вызовах;
- хранение указателя в памяти.

Первое решение добавляет небольшие накладные расходы во все нелистовые функции, даже те, которые не обращаются к альтернативному стеку. Второе решение избегает дополнительных расходов в большинстве функций за счет повышения стоимости кода в тех, которые имеют стековый фрейм.

Для хранения указателей на стек используется .shared память. Это существенно упрощает ее резервирование и освобождает от необходимости учитывать номер блока нитей в вычислении адреса указателя текущей нити. Более того, адрес зависит только от номера синхронной группы в блоке: все нити в пределах одной синхронной группы используют один массив в глобальной памяти как стек, и, соответственно, имеют один и тот же указатель на его вершину.

При входе в simd-регион указатель на вершину стека переключается на небольшой регион, выделенный в .local-памяти. За счет этого вызываемые внутри simd-региона функции могут продолжать работать с альтернативным стеком таким образом, что разные нити в одной синхронной группе могут хранить на нем разные данные.

5. Выполнение кода вне SIMD-регионов в синхронных группах

Как отмечалось ранее, OpenMP-нити выполняются синхронными группами, а отдельные PTX-нити в составе синхронной группы могут обрабатывать различные данные только внутри simd-регионов. Соответственно, код вне simd-регионов необходимо выполнять, как если бы в каждой синхронной группе была активна только одна нить, но при этом обеспечить, чтобы при входе в simd-регион все нити синхронной группы могли быть активны и имели одинаковые значения живых регистров. PTX не предоставляет средств для управления маской активных нитей, так что явно активировать нити на входе в simd-регион нельзя: необходимо обеспечить, чтобы они выполнили все переходы от начала выполнения GPU-ядра до входа в регион.

Рассмотрим, что происходит в случае, если код GPU-ядра выполняется нитями синхронной группы, имеющими изначально одинаковое состояние. Инструкции, выполняющие вычисления на регистрах, очевидно, сохраняют этот инвариант: если до выполнения инструкции I нити имели одинаковые значения регистров, то и после выполнения очередной инструкции они будут

совпадать. Инструкции загрузки из памяти также сохраняют этот инвариант: поскольку инструкция выдается синхронно для всех нитей и значение регистра, задающего адрес, совпадает, то и результат загрузки будет одинаковый. Далее, для записи в память PTX гарантирует, что ровно одна из синхронных записей будет успешна, и поскольку все нити записывают одно и то же значение, изменение глобальной памяти будет таким, как если бы запись выполняла одна нить.

Среди инструкций PTX, выдаваемых компилятором, можно выделить два класса инструкций, которые не гарантируют нужной нам инвариантности выполнения: это все атомарные инструкции и команда вызова функции (call). Теоретически, пользовательский код может содержать ассемблерные вставки, которые также могли бы содержать нарушающие инвариантность инструкции, но в настоящее время они не могут возникать в OpenMP target-регионах: для этого синтаксис ассемблерной строки должен быть допустимым как для PTX-ассемблера, так и для хост-архитектуры (обычно x86-64), что невозможно.

Отметим, что сами по себе call-инструкции не нарушают инвариантность: проблема в том, что вызванный код в целом может иметь наблюдаемые эффекты, различные в зависимости от того, активны ли все нити синхронной группы или только одна. Так, вызов vprintf приведет к появлению 31 копии форматированных данных, а вызов malloc выделит 32 блока памяти (и вернет уникальный указатель в каждой нити синхронной группы).

Для вызовов мы можем потребовать, чтобы все вызванные функции сохраняли инвариантность (это можно требовать для всех функций, компилируемых GCC; исключение составляют функции malloc, free, vprintf, на которые полагается реализация libc (newlib) для nvptx).

Для атомарных операций и перечисленных трех функций необходимо обеспечить, чтобы операция была выполнена ровно одной нитью в синхронной группе и после этого вычисленный регистр был распространен в другие нити этой группы (кроме функции free, которая не имеет возвращаемого значения). Для этого достаточно выполнить оригинальную команду под предикатом, истинным только в нулевой нити, и воспользоваться инструкцией shfl для распространения регистра с результатом. Например, если исходный код выполняет инструкцию атомарного обмена (atom.exch.b32 dst, [addr], src), то инструментированный код выглядит так:

```
[pred]  atom.exch.b32   dst, [addr], src
        shfl.idx.b32    dst, dst, 0
```

Инструкции shfl поддерживаются в PTX только для GPU архитектурного уровня sm_30 или выше (архитектура NVIDIA Kepler или более новая). Для акселераторов без поддержки shfl-инструкций можно воспользоваться .shared-

памятью (но это не реализовано: в настоящее время кодогенерация GCC рассчитана на уровень не ниже sm_30).

Предикатный регистр pred можно вычислять перед каждым использованием или в прологе функции, сравнивая индекс нити %laneid с 0. Для call-инструкций инструментация выполняется аналогично.

Отметим, что предложенная схема неявно предполагает, что инструментируемые инструкции сами не имеют контролирующего предиката. В настоящее время GCC не может генерировать инструкции с условным выполнением для nvptx (это может происходить только после распределения регистров, но для nvptx вся последовательность машинно-зависимых преобразований выполняется на псевдорегистрах). Хотя преобразование можно обобщить на случаи, когда исходная команда сама выполняется под предикатом, эффективнее будет запретить выдачу атомарных инструкций и вызовов под предикатом.

Заметим, что хотя предложенный выше подход решает проблему для кода вне simd-регионов, для кода внутри simd-регионов такое преобразование делать нельзя, так как в них побочные эффекты от атомарных инструкций должны происходить независимо во всех активных нитях (причем не все нити синхронной группы могут быть активны). Просто запретить инструментацию для инструкций в simd-регионах было бы недостаточно: если в них есть вызовы других функций, для вызываемой функции также нужно запретить инструментацию, и так далее. Решить это за счет клонирования функций нельзя, так как в общем случае оно невозможно.

Предлагается использовать следующее решение. Изменим инструкцию shfl так, чтобы индекс нити с распространяемым регистром мог быть динамическим:

```
[pred]  atom.op.b32     dst, ...
        shfl.idx.b32    dst, dst, master
```

Как отмечено ранее, вне simd-регионов pred = (%laneid == 0) и master = 0. Внутри simd-регионов необходимо обеспечить, чтобы исходная операция выполнилась во всех нитях, так что pred = 1. Далее, последующая инструкция shfl не должна изменить значение регистра: для этого достаточно master = %laneid.

Чтобы эффективно вычислять pred и master, будем для каждой синхронной группы хранить флаг, указывающий, находится ли выполнение внутри simd-региона. Положим, что вне simd-регионов значение этого флага равно 0, а внутри -1 (т. е. все биты в слове установлены). Тогда можно сначала вычислить master как %laneid & mask (побитовое «и» индекса нити со значением флага), и затем pred как %laneid == master.

Как и для указателя на вершину стека, флаги предлагается хранить в массиве в .shared памяти.

6. Подходы, использованные в реализации OpenACC

Реализация OpenACC в GCC не использует libgomp для организации выполнения параллельных регионов на акселераторе: вместо этого компилятор непосредственно генерирует код, выполняющийся на границах параллельных и векторных регионов. Тем не менее, похожие задачи также решаются и для OpenACC. Описанная в предыдущей секции проблема решается следующим образом. Код вне векторных регионов выполняется только нитью 0 в каждой синхронной группе. Все остальные нити выполняют только переходы на конец очередного базового блока; при этом, если ветвление в конце базового блока является условным, с помощью инструкции shfl выполняется распространение предикатного регистра из нити 0 в остальные. Таким образом достигается, что все нити каждой синхронной группы выполнят переходы между базовыми блоками до входа в векторный регион в той же последовательности, что и нить 0. При достижении векторного региона состояние побочных нитей не синхронизировано с основной нитью, так как они пропускали все базовые блоки. Для синхронизации состояния выполняется копирование содержимого регистров и стекового фрейма. При этом на входе в векторный регион можно скопировать только текущий стековый фрейм, хотя содержимое фреймов вызывающих функций также может использоваться внутри векторного региона.

7. Тестирование реализации на модельных примерах

Для оценки производительности генерируемого с помощью GCC кода для пользовательских программ на OpenMP для акселераторов NVIDIA PTX был создан набор тестов, содержащий модельные примеры, пригодные для предварительной оценки производительности реализации OpenMP. Он включает в себя умножение матрицы на вектор (mul-matr-vec), извлечение квадратного корня методом Ньютона (newton-sqrt), вычисление скалярного произведения двух векторов (scalar-prod) и сложение двух векторов (vector-add). Каждый из этих тестов был реализован с использованием интерфейсов программирования OpenMP, OpenACC и CUDA (являясь более низкоуровневым интерфейсом, чем два других, он представляет для них ориентировочную верхнюю границу производительности).

Ниже приведены графики зависимости времени выполнения от различных параметров запуска (число нитей, блоков, наличие прагмы simd, размер входных данных). В случаях, когда графики для всех четырех тестов имеют типичный вид, приведен один график. Измерения времени, проводимого управлением внутри GPU-ядер, выполнялись с помощью утилиты nvprof, поставляемой в составе CUDA toolkit.

Как видно из графика на рис. 2, накладные расходы на старт параллельного региона составляют около 100 мкс. Для небольших входных данных (размеры вектора до 2^{11} элементов) выгоднее запускать меньшее количество нитей. С

увеличением размеров матрицы прирост производительности от дополнительных нитей начинает перевешивать накладные расходы на их запуск.

Распараллеливание выполнения на 32 контекста (прагма simd) дает ожидаемый прирост производительности, близкий к 32-кратному, для теста newton-sqrt. Это ожидаемо, так как этот тест имеет высокое соотношение числа арифметических опер аций к числу операций с памятью. График справа на рис. 3 типичен для остальных тестов из набора. Сверхлинейное ускорение, вероятно, объясняется увеличением эффективности использования кеша на акселераторе. С другой стороны, на тестах с большой интенсивностью обращений к памяти масштабирование может быть ограничено пропускной способностью памяти, что объясняет снижение достигнутого ускорения до величин, существенно меньших 32, при росте количества блоков и нитей.

На рис. 4 каждая из линий на графике слева отображает зависимость времени выполнения от числа блоков нитей (прагма teams) при фиксированном числе нитей в блоке. Каждая из линий на графике справа показывает зависимость времени выполнения от числа нитей в каждом блоке при фиксированном числе блоков.

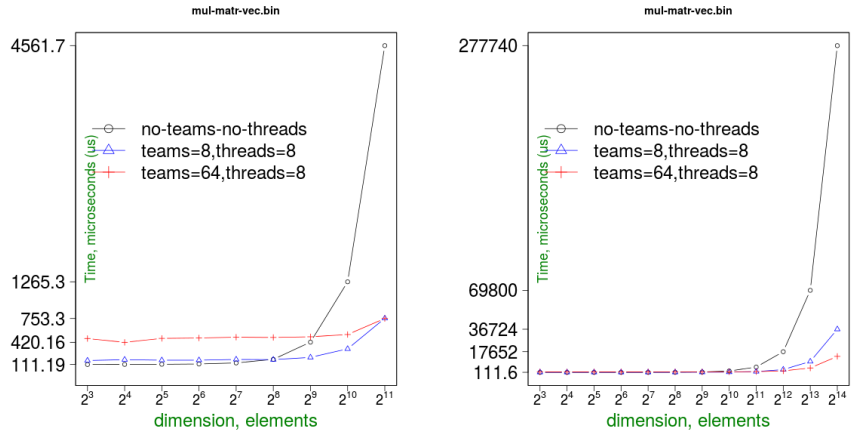


Рис. 2. Зависимости времени выполнения параллельного региона от размерности матрицы для различного числа нитей и блоков.

Fig. 2. Execution time of a parallel region againO9st matrix size, for different thread and block counts.

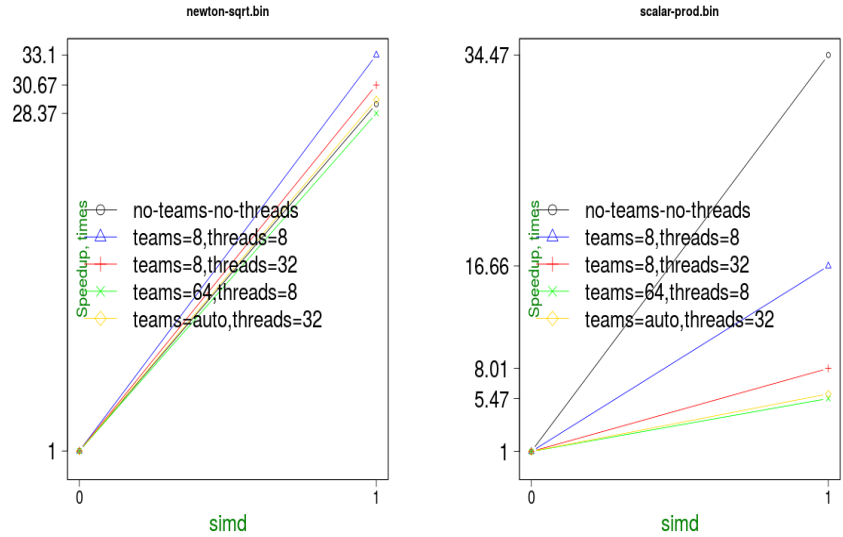


Рис. 3. Ускорение, полученное при включении прагмы simd для различных значений чисел нитей и блоков на тесте newton-sqrt (слева) и scalar-prod (справа).

Fig. 3. Speedup from enabling simd pragma for various numbers of teams and threads per team on newton-sqrt test (left) and the scalar-prod test (right).

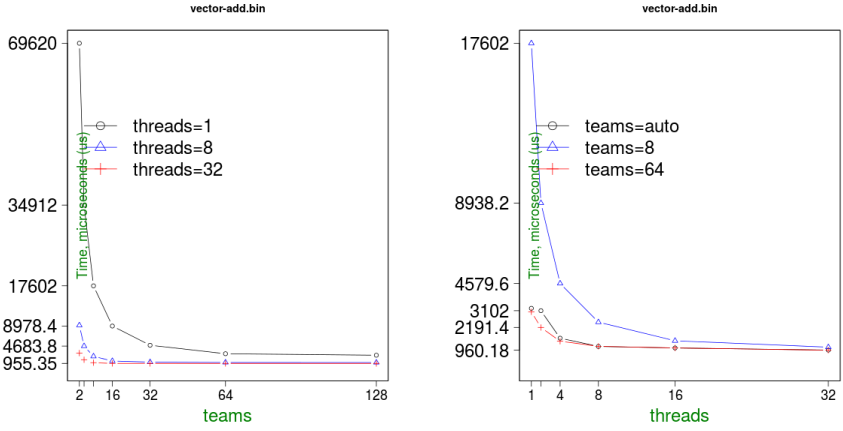


Рис. 4. Зависимости времени выполнения параллельного региона от числа блоков при различных значениях числа нитей и от числа нитей в блоке для различных значений числа блоков.

Fig. 4. Execution time of a parallel region against number of teams, for different threads per team counts (left), and against threads per team, for different team counts (right)

Список литературы

[1]. OpenMP Application Program Interface, 2013. (http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)
[2]. GCC 4.9 Release Series, 2014. (https://gcc.gnu.org/gcc-4.9/changes.html)
[3]. Parallel Thread Execution ISA, 2016. (http://docs.nvidia.com/cuda/parallel-thread-execution/)
[4]. S. Gray. Assembler for NVIDIA Maxwell architecture, 2016 (https://github.com/NervanaSystems/maxas)

Implementing OpenMP 4.0 for the NVIDIA PTX architecture in GCC compiler

A.V. Monakov <amonakov@ispras.ru>
V.A. Ivanishin <vlad@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. The paper describes the approach used in implementing OpenMP offloading to NVIDIA accelerators in GCC. Offloading refers to a new capability in OpenMP 4.0 specification update that allows the programmer to specify regions of code that should be executed on an accelerator device that potentially has its own memory space and has an architecture tuned towards highly parallel execution. NVIDIA provides a specification of the abstract PTX architecture for the purpose of low-level, and yet portable, programming of their GPU accelerators. PTX code usually does not use explicit vector (SIMD) computation; instead, vector parallelism is expressed via SIMT (single instruction – multiple threads) execution, where groups of 32 threads are executed in lockstep fashion, with support on hardware level for divergent branching. However, some control flow constructs such as spinlock acquisition can lead to deadlocks, since reconvergence points after branches are inserted implicitly. Thus, our implementation maps logical OpenMP threads to PTX warps (synchronous groups of 32 threads). Individual PTX execution contexts are therefore mapped to logical OpenMP SIMD lanes (this is similar to the mapping used in OpenACC). To implement execution of one logical OpenMP thread by a group of PTX threads we developed a new code generation model that allows to keep all PTX threads active, have their local state (register contents) mirrored, and have side effects from atomic instructions and system calls such as malloc happen only once per warp. This is achieved by executing the original atomic or call instruction under a predicate, and then propagating the register holding the result using the shuffle exchange (shfl) instruction. Furthermore, it is possible to setup the predicate and the source lane index in the shuffle instruction in a way that this sequence has the same effect as just the original instruction inside of SIMD regions. We also describe our implementation of compiler-defined per-warp stacks, which is required to have per-warp automatic storage outside of SIMD regions that allows cross-warp references (normally automatic storage in PTX is implemented via .local memory space which is visible only in the PTX thread that owns it). This is motivated by our use of unmodified OpenMP lowering in GCC where possible, and thus using libgomp routines

for entering parallel regions, distribution of loop iterations, etc. We tested our implementation on a set of micro-benchmarks, and observed that there is a fixed overhead of about 100 microseconds when entering a target region, mostly due to startup procedures in libgomp (and notably due to calls to malloc), but for long-running regions where that overhead is small we achieve performance similar to analogous OpenACC and CUDA code.

Keywords: compilers; GCC; OpenMP; CUDA; PTX.

DOI: 10.15514/ISPRAS-2016-28(4)-10

For citation: A.V. Monakov, V.A Ivanishin. Implementing OpenMP 4.0 for the NVIDIA PTX architecture in GCC compiler. *Trudy ISP RAN/Proc. ISP RAS]*, vol. 28, issue 4, 2016. pp. 169-182 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-10

References

[1]. OpenMP Application Program Interface, 2013. (http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)
[2]. GCC 4.9 Release Series, 2014. (https://gcc.gnu.org/gcc-4.9/changes.html)
[3]. Parallel Thread Execution ISA, 2016. (http://docs.nvidia.com/cuda/parallel-thread-execution/)
[4]. S. Gray. Assembler for NVIDIA Maxwell architecture, 2016 (https://github.com/NervanaSystems/maxas)

Модель поведения объектов, подверженных спонтанному изменению, в прецедентном подходе к управлению[1]

[1,2] *В.Н. Юдин <yudin@ispras.ru>*
[1,3] *Л.Е. Карпов <mak@ispras.ru>*
[1] *Институт системного программирования РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*
[2]*Московский областной научно-исследовательский клинический институт им. М.Ф. Владимирского, Россия, 129110, г. Москва, ул. Щепкина, д. 61/2*
[3]*Московский государственный университет имени М.В. Ломоносова, Россия, 119991, г. Москва, Ленинские горы, д. 1.*

Аннотация. В рамках прецедентного подхода к управлению сложными объектами, не поддающимися формализации в виде математической модели, предложена модель поведения объектов, вовлеченных в вялотекущие процессы и подверженных, казалось бы, внезапному спонтанному переходу в лавинообразный процесс. Для описания таких изменений вводится понятие сопутствующего класса, влияние которого можно трактовать как дополнительное управляющее воздействие по отношению к исходному классу, к которому принадлежал объект. Модель ориентирована прежде всего на медицину, где можно найти множество аналогов такого поведения.

Ключевые слова: сложные объекты, прецедентный подход к управлению, классы состояний, управляющее воздействие, вялотекущие процессы, лавинообразный процесс.

DOI: 10.15514/ISPRAS-2016-28(4)-11

Для цитирования: В.Н. Юдин, Л.Е. Карпов. Модель поведения объектов, подверженных спонтанному изменению в прецедентном подходе к управлению. Труды ИСП РАН, том 28, вып. 4, 2016, стр. 183-192. DOI: 10.15514/ISPRAS-2016-28(4)-11

В настоящее время разработано множество различных программных систем поддержки принятия решений. Наиболее трудными для анализа и принятия решений являются ситуации, чьи характеристики не поддаются формализации,

[1] Работа поддержана грантами Российского фонда фундаментальных исследований № 15-01-02362-а и № 15-07-02355-а.

то есть выявлению основных действующих факторов и связей между ними. В силу недостаточности знаний об объекте и о среде, в которой он функционирует, получить точную модель поведения такого объекта не представляется возможным. Однако управление этими объектами представляет не меньший интерес и является не менее важным, чем управление хорошо формализуемыми объектами.

Вывод, основанный на прецедентах, – это метод принятия решений, в котором используются знания о предыдущих ситуациях или случаях (прецедентах). При таком выводе прецедент, если он признан схожим, является обоснованием решения. Подобная методика принятия решений моделирует человеческие рассуждения, на практике она применяется во многих областях человеческой деятельности, где взята на вооружение широким спектром всевозможных приложений, в том числе прикладными системами управления объектами, не поддающимися формализации. Рассматривая человеческий организм как объект управления, приходится констатировать, что современные методы формализации не позволяют описать его с помощью простой математической модели. Именно поэтому методы вывода по прецедентам для систем поддержки врачебных решений могут рассматриваться как весьма перспективные.

Исследовательская система, разрабатываемая в Институте системного программирования Российской академии наук с поддержкой со стороны Российского фонда фундаментальных исследований, предназначена для отработки методов прецедентного подхода к принятию решений. Система "Спутник Врача" – инсталляция, созданная основе на базе Московского областного клинического института им. Владимирского – предназначена для информационной поддержки врачебных решений в диагностике и выборе лечения.

Вывод по прецедентам – это поиск решения подобных проблемных ситуаций на основе прошлого опыта решения задач. Вместо того, чтобы искать решение каждый раз сначала, можно пытаться использовать решение, ранее принятое в сходной ситуации, возможно, адаптировав его к изменившейся ситуации текущего случая. После того, как текущий случай будет обработан, его можно внести в базу прецедентов вместе с принятым решением для возможного последующего использования.

Согласно [1]), прецедент включает:

1. Описание проблемы,
2. Решение проблемы,
3. Результат применения решения.

Подход, основанный на прецедентах, в наиболее общем виде состоит из следующих составляющих [2]:

1. Извлечение наиболее релевантных прецедентов для текущего случая из базы прецедентов
2. Адаптация выбранного решения для текущего случая, если это необходимо
3. Применение решения
4. Оценка применения (проверка корректности) решения
5. Сохранение случая (добавление текущего случая в базу прецедентов)

На основании проведенных исследований разрабатываемого математического аппарата и накопленной врачебной практики (см. [3-4]) был предложен подход к формализации понятия «лечение», рассматривая процесс лечения как продолжительный процесс управления таким сложным объектом, как человеческий организм, и трактуют процесс лечения как последовательность управляющих воздействий на организм больного.

В дальнейшем этот подход был распространен на другие виды процессов. «Классические» подходы к управлению объектами (см., например, [5]) строятся на предположении, что можно получить точную, аналитически заданную форму функциональной зависимости входных и выходных параметров системы управления. Современные требования к качеству управления сложными объектами в технике, экономике, экологии, медицине и других областях человеческой деятельности привели к тому, что во многих случаях методы классической теории управления, в частности, принятия решений, оказываются непригодными. В силу недостаточности знаний об объекте и среде, в которой он функционирует, получить точную модель поведения такого объекта затруднительно. Из-за сложности реальных объектов и неопределенности их функционирования, особенно заметные при попытках поддерживать врачебные решения, задача построения математической модели их поведения откладывается на долгое время. Однако управление такими объектами, относящимися ко второй их категории, не менее интересно и не менее важно, чем управление хорошо формализуемыми объектами.

Для управления такого рода объектами был предложен метод принятия решений, в котором вместо математической модели объекта доступна априорная информация о состояниях объекта управления, управляющих воздействиях на него и результатах воздействий, что соответствует прецедентному подходу. Предложенный подход является оригинальным и не используется в настоящее время, хотя его полезность стала ясна с самого начала творческого содружества с практикующими врачами Московского областного научно-исследовательского клинического института (МОНИКИ). Несмотря на то, что идея прецедентного управления во многих сферах человеческой деятельности лежит на поверхности, вместо нее используется вывод по правилам. Так обстоит дело и в медицинской литературе. Медицина -

прецедентная наука, тем не менее, все примеры врачебных решений строятся на правилах «если-то», в частности, на деревьях решений [6].

Предложенная организация управления существенно отличается от управления, выполняемого на основе прямых математических расчётов. Система управления должна обеспечить управление в условиях неполноты знаний, как самих объектов, так и результатов управляющих воздействий на эти объекты. Это требование приводит к тому, что система управления обязательно должна обладать способностью к внутренним изменениям, вызванным постепенным накоплением знаний об объектах в процессе управления.

Схематически любой шаг управления объектом можно представить в виде тройки составляющих: состояние объекта до воздействия, управляющее воздействие и состояние объекта после воздействия. В терминах вывода по прецедентам – это, соответственно, описание проблемы, решение и исход. Эту тройку будем называть «случаем». Совокупность таких случаев образует так называемую «базу прецедентов». Случаи, отражающие хронологию воздействий на отдельный объект, связываются в так называемую «цепь управляющих воздействий», вершины которой – состояния объекта, а дуги – управляющие воздействия.

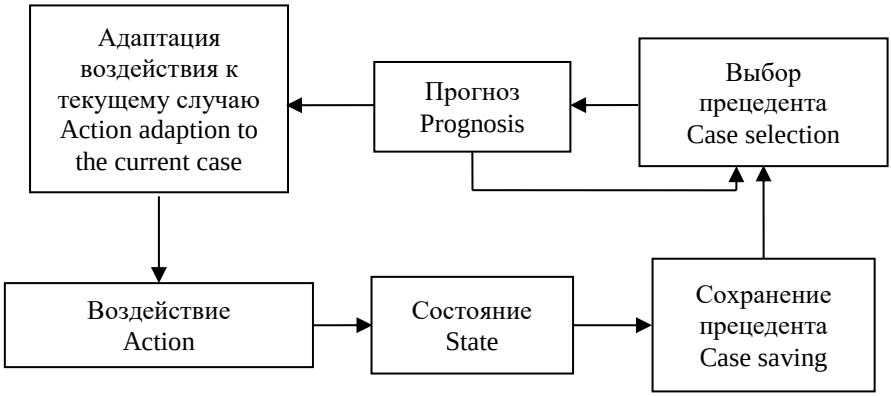


Рис. 1. Структурная схема управления по прецедентам.

Fig. 1. Case control scheme.

Если на основе априорной информации о предметной области удается сформировать обобщенные образы – классы состояний, то управляющее воздействие можно рассматривать как отображение объекта из класса в класс (в частности, возможно отображение на тот же класс, то есть удержание объекта в том же классе). Понятие цели управления не всегда отождествляется с достижением конкретного состояния. Целью может быть управляемое поведение, учитывающее переходы объекта из одного класса состояний в

другой. Так, при лечении хронических заболеваний, задача восстановления больного органа – невыполнима. Тогда целью управления может стать замедление процесса дегенерации рабочей ткани. Поэтому, говоря о цели, имеется в виду не состояние, а оптимальное поведение объекта. Необходимо найти алгоритм управления, обеспечивающий достижение цели за конечное число управляющих воздействий.

Структура управления, где для прогноза вместо математической модели объекта используется накопленная база прецедентов, приведена на рис. 1.

При выборе воздействия нужно решить несколько задач:

- оценка состояния объекта до воздействия (отнесение его состояния к тому или иному классу) по его наблюдаемым признакам.
- отбор прецедентов со схожими состояниями,
- прогнозирование поведения объекта при воздействиях, которые заимствованы у этих прецедентов.
- окончательный выбор воздействия для перевода объекта в нужный класс.
- оценка состояния объекта в классе назначения.

При выборе воздействия из базы прецедентов выбираются воздействия, которые применялись к сходным состояниям. Соответствующий прецедент содержит воздействие, при помощи которого достигается нужный класс состояния.

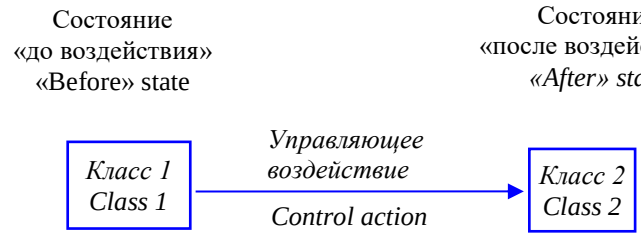


Рис. 2. Структура прецедента при управлении.
Fig. 2. Case structure in case control.

Оценка состояния объекта «до воздействия» – это сравнение его признаков с описаниями классов в базе прецедентов. Если некоторое подмножество признаков попадает в границы соответствующего класса, это говорит о возможной принадлежности состояния объекта к классу. В общем случае, можно выделить ряд таких подмножеств (не обязательно раздельных). Каждое из них может соответствовать либо одному классу, либо сразу нескольким

Отбор похожих состояний производится с помощью введенного ранее метода для оценки близости подобных объектов [3]. Оценка состояния «до

воздействия» производится в признаковом пространстве текущего состояния объекта. Соответствующая объекту точка сравнивается с расположением сформированных классов в проекции на пространство его признаков. Прецеденты, если таковые найдены, ранжируют по степени близости к текущему состоянию объекта.

При выборе воздействия из базы прецедентов выбираются те из них, что применялись к сходным состояниям. Соответствующий прецедент содержит воздействие, при помощи которого достигается нужный класс (рис. 3).

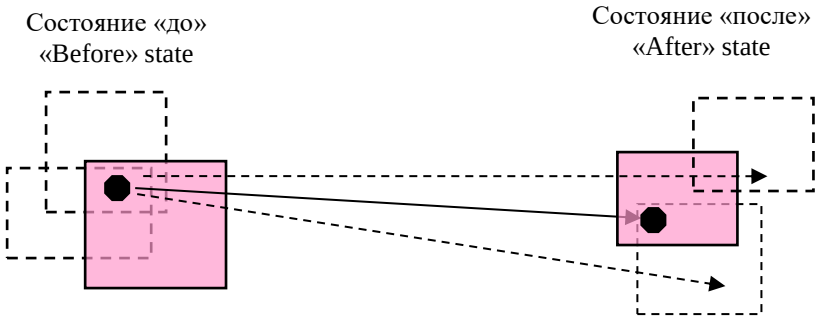


Рис. 3. Отбор прецедентов
Fig. 3. Case selection

В случае если найденный прецедент не полностью совпадает с текущим, должна выполняться адаптация решения – модификация воздействия, имеющегося в выбранном прецеденте. Невозможно выработать единый вариант для такой адаптации, так как это в большой степени зависит и от прикладной области, и также связано с большим разнообразием классов и типов воздействий в такой области, как медицина. Если существуют алгоритмы адаптации, они обычно предполагают наличие зависимости между начальными состояниями прецедентов и содержащимися в них воздействиями. Такие зависимости могут задаваться человеком при построении базы прецедентов или обнаруживаться в базе автоматически методами добычи данных.

Идентификация состояния объекта в классе назначения производится аналогично, в пространстве признаков текущего случая. Задача идентификации объекта в классе до воздействия не является типичной для каждого шага управления. Еще до первого воздействия нужно определиться с принадлежностью объекта, выявив все необходимые дополнительные признаки, возможно, потратив на это дополнительные ресурсы, иначе управление теряет смысл. Принадлежность объекта тому или иному классу после воздействия уже предсказуема, и, как правило, отслеживается по одному или нескольким признакам, которые являются информативными для данного класса.

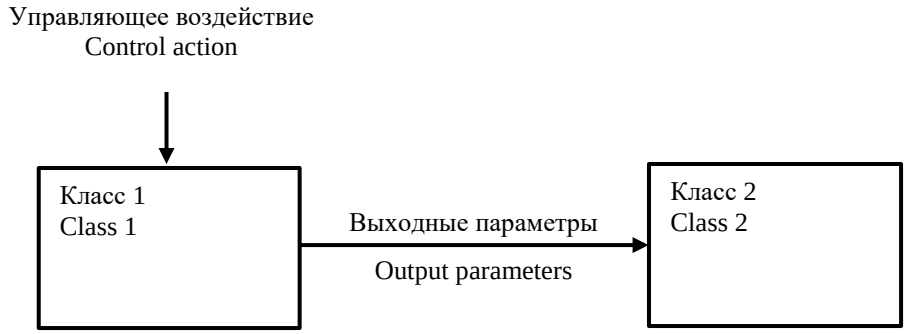


Рис. 4. Управляющее воздействие, переводящее объект в новое состояние
Fig. 4. Control action, and changing object state.

Необходимость адекватно описывать изменения в поведении объекта, желание формализовать это поведение приводит к модификации классической схемы работы с прецедентами. Конечно, в главном общая схема управления объектом сохраняется, однако, для стандартного процесса обработки прецедента, в котором управляющее воздействие (как составляющая прецедента) переводит его из одного класса состояний в другой (в частности, оставляет в том же классе) (рис. 4), введено понятие сопутствующего класса, подобно сопутствующему заболеванию в медицине.

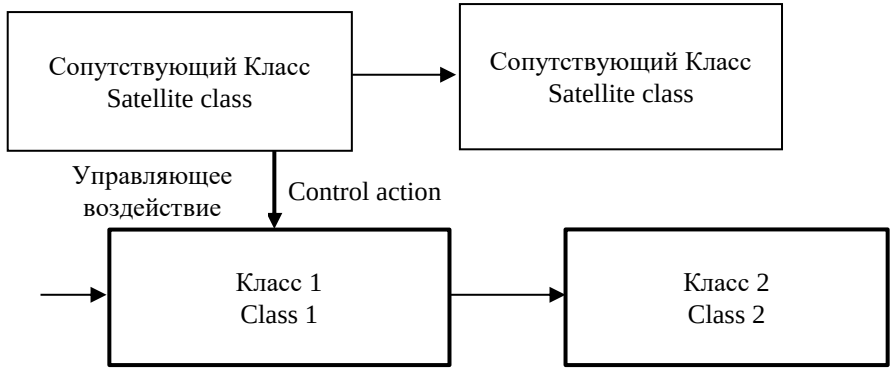


Рис. 5. Влияние сопутствующего класса состояний на исходный.
Fig. 5. Satellite state class influence onto the initial class.

В процесс, характеризующийся нахождением объекта в исходном классе состояний, добавляется еще один появившийся класс (рис. 5). Наличие сопутствующего класса является своего рода управляющим воздействием на

объект в исходном классе. Это приводит к резкому, теперь уже не спонтанному, изменению поведения объекта. При этом причинами появления сопутствующего класса могут самые разные явления: как подспудно возникающие внутри исходного класса, так и возникшие вследствие каких-то внешних воздействий. Эти причины должны стать предметом дальнейших исследований.

Предложенная схема позволяет адекватно описать существенное, изначально представляемое как спонтанное, изменение в поведении объекта, которое до этого можно было описать как вялотекущее, и формализовать это поведение.

Исходящие стрелки на рис. 5 указывают направление передачи параметров описываемого состояния от предыдущего воздействия. Значения этих параметров, даже при отсутствии воздействия извне, также являются своего рода управляющим воздействием. Можно проиллюстрировать сказанное следующим примером из медицины: вялотекущий процесс угасания почечного трансплантата вдруг переходит в лавинообразный процесс, за несколько месяцев приводящий к полной потере его функции. Причиной служит появление сопутствующего заболевания – инфаркта миокарда, следствием которого оказывается резкое снижение кровотока в организме. Сопутствующее заболевание и его следствие оказываются при этом управляющим воздействием на трансплантат: снижение кровотока приводит к снижению функции органа, что, в свою очередь, влияет на работу других органов, и через положительную обратную связь подталкивает процесс угасания. Можно привести множество других примеров из медицины, когда элементарная простуда вызывает обострение соматических заболеваний. Однако эти примеры легко найти и в других прикладных областях.

Структура прецедента при такой схеме не претерпевает больших изменений. В базе прецедентов, заполненной как реальными, так и смоделированными случаями, должен храниться случай, когда на исходный класс оказывается воздействие, аналогичное тому, что оказывает сопутствующий класс. Тогда поведение, ранее считавшееся спонтанным, уже можно прогнозировать.

Сопутствующий класс может развиваться по своим правилам. Как результат – его воздействие на состояние в этом же классе (иными словами, развитие сопутствующего заболевания).

Список литературы

[1]. Klaus-Dieter Althof, Eric Auriol, Ralph Barlette, and Michel Manago. A Review of Industrial Case-Based Reasoning Tools. AI Intelligence, 1995.

[2]. Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. AI Communications, 7(1):39-59, 1994.

[3]. Valery Yudin, Leonid Karpov. "The Case-Based Software System for Physician's Decision Support". Sami Khari, Lenka Lhotska, Nadia Pisanti (eds.), "Information Technology in Bio- and Medical Informatics, ITBAM 2010", Proc. of the First

International Conference, Bilbao, Spain. Lecture Notes in Computer Science Sublibrary: SL 3, Springer Verlag, Berlin, Heidelberg, 2010, pp. 78-85. ISSN 0302-9743.

[4]. А. В. Ватазин, Л. Е. Карпов, В. Н. Юдин. Процесс лечения как адаптивное управление человеческим организмом в программной системе "Спутник врача". Альманах клинической медицины, т. 17, ч. 1, 2008, стр. 262-265.

[5]. Я. З. Цыпкин. Адаптация и обучение в автоматических системах. М.: Наука, 1968.

[6]. Hillary Don. Decision making in critical care. University of California School of Medicine San Francisco, California, B. C. Decker Inc., The C. V. Mosby company, 1985, есть русский перевод: Х. Дон. Принятие решения в интенсивной терапии, М., Медицина, 1995, 224 с., ISBN 5-225-00489-X, ISBN 0-941158-35-7.

Model of spontaneously changing object behavior in case control approach

[1,2] *V.N. Yudin <yudin@ispras.ru>*
[1,3] *L.E. Karpov <mak@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn str., Moscow, 109004, Russia.*
[2] *M.F. Vladimirsky Moscow Regional Science Research Clinical Institute, 61/2, Shchepkina str., Moscow, 129110, Russia*
[3] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. As a further development of case-based approach to complex object control with objects that cannot be formalized by mathematical model, authors offer new object behavior model. This model deals with objects that are involved in low-intensity processes, and then suddenly (as it seems) and spontaneously are changing their states in avalanche-like manner. To describe such changes a new conception of satellite class is introduced. The influence of satellite classes may be treated as an additional control action that effects on the class to which the object belonged. This model is firstly oriented on medicine applications where one can find many examples of the similar behavior.

Keywords: complex objects, case-based object control, classes of states, control action, low-intensity process, avalanche-like process.

DOI: 10.15514/ISPRAS-2016-28(4)-11

For citation: Yudin V.N., Karpov L. E. Model of spontaneously changing object behavior in case control approach. *Trudy ISP RAN /Proc. ISP RAS*, 2016, vol. 28, issue 4, pp. 183-192 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-11

References

[1]. Klaus-Dieter Althof, Eric Auriol, Ralph Barlette, and Michel Manago. A Review of Industrial Case-Based Reasoning Tools. AI Intelligence, 1995.

[2]. Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. AI Communications, 7(1):39-59, 1994.

[3]. Valery. Yudin, Leonid Karpov. "The Case-Based Software System for Physician's Decision Support". Sami Khari, Lenka Lhotska, Nadia Pisanti (eds.), "Information Technology in Bio- and Medical Informatics, ITBAM 2010", Proc. of the First International Conference, Bilbao, Spain. Lecture Notes in Computer Science Sublibrary: SL 3, Springer Verlag, Berlin, Heidelberg, 2010, pp. 78-85. ISSN 0302-9743.

[4]. Yudin V. N., Karpov L. E., Vatazin A. V. Protsess lecheniya kak adaptivnoe upravlenie chelovecheskim organizmom v programmnoy sisteme «Sputnik vracha» [Cure process as an adaptive control of human organism in software system "Physician's Partner"]. Almanah klinicheskoy meditsinyi [Almanac of Clinical Medicine], Moscow, 2008, v. 17, part 1, pp. 262-265, ISSN 2072-0505 (in Russian).

[5]. J. Z. Tsypkin. Adaptatsya i obutchenie v avtomatchieskykh sistemakh [Adaptation and learning in automate systems]. Moscow, Nauka, 1968 (in Russian).

[6]. Hillary Don. Decision making in critical care. University of California School of Medicine San Francisco, California, B. C. Decker Inc., The C. V. Mosby company, 1985, ISBN 5-225-00489-X, ISBN 0-941158-35-7.

Некоторые задачи на графовых базах данных

Р. И. Гуральник <guralnikr@gmail.com>
Санкт-Петербургский государственный университет,
199034, Санкт-Петербург, Университетская набережная, д. 7/9

Аннотация. Одним из наиболее популярных и актуальных подвидов нереляционных баз данных являются графовые базы данных. В данной работе рассмотрены задачи на таких базах данных, которые наиболее часто встречаются в современной литературе. Изучены задачи максимизации влияния, motif mining (MM), задача оценки схожести узлов графа, сопоставление образца в графе. Рассмотрены первичные алгоритмы каждого направления и некоторые промежуточные работы. Проанализированы алгоритмы, соответствующие текущему положению дел.

Ключевые слова: графовые базы данных; сетевые мотивы; сопоставление с образцом; максимизация влияния; simrank.

DOI: 10.15514/ISPRAS-2016-28(4)-12

Для цитирования: Гуральник Р.И. Некоторые задачи на графовых базах данных. Труды ИСП РАН, том 28, вып. 4, 2016, стр. 193-216. DOI: 10.15514/ISPRAS-2016-28(4)-12

1. Введение

Большие данные (англ. big data) – совокупность подходов, инструментов и методов обработки структурированных и неструктурированных данных огромных объемов и значительного многообразия для получения воспринимаемых человеком результатов [1]. Другими словами, большие данные – это проблема хранения и обработки гигантских объемов данных. С другой стороны, обработка больших объемов информации – это только часть «айсберга». Как правило, когда говорят о «больших данных», то используют наиболее популярное определение трех «V», что означает Volume – объем данных, Velocity – необходимость обрабатывать информацию с большой скоростью и Variety – многообразие и часто недостаточную структурированность данных. [2]

Графовые базы данных стали одним из наиболее актуальных представлений больших данных. Их популярность обусловлена их удобством применения в задачах, в которых данные имеют большое количество связей, например в пищевых цепочках, белок-белковых взаимодействиях и социальных сетях.

Кроме того, ребра графа являются хранимыми данными, а значит обход графа не требует дополнительных вычислений. Такая система оказалась естественной и востребованной в современном мире сети Интернет и социальных сетей [3].

Самым крупным разделом задач на графовых базах данных является глубинный анализ данных (data mining). Сюда входят задачи по обучению ассоциативным правилам, классификации и категоризации данных, кластерный анализ, регрессионный анализ и др. Среди менее крупных разделов задач можно отметить пространственный и статистический анализ данных, визуализацию аналитических данных [1, 4]. Описать все задачи в данном небольшом обзоре не представляется возможным.

Данная статья представляет собой обзор наиболее популярных задач на графовых базах данных из раздела data mining: оценка схожести объектов, распространение влияния узлов внутри графа, поиск часто встречающихся подграфов и сопоставление с образцом. Их популярность отражается большим набором публикаций по этим задачам на крупных конференциях последних годов. В работе будут представлены традиционные постановки задач и базовые алгоритмы их решения, предложенные 10-15 лет назад, а также рассмотрены некоторые работы, соответствующие текущему положению дел по рассматриваемому направлению.

Структурно данный обзор состоит из 4 разделов. Первый раздел посвящен задаче оценки схожести объектов и алгоритму SimRank. Второй раздел описывает задачу максимизации влияния узлов графа. В третьем разделе представлен анализ задачи сопоставления графа с образцом подграфа. В четвертом раздел содержит алгоритмы поиска сетевых мотивов (network motifs).

2. Simrank

В данном разделе мы рассмотрим такую задачу как измерение "похожести" объектов. Во множестве основных задач на графах, например, в прогнозировании связей (*link prediction*), кластеризации, обнаружении спама, поиске часто встречающегося подграфа, рекомендательных системах и др. требуется тщательный анализ схожести сущностей, а значит возникает необходимость платформы для эффективного вычисления похожести объектов.

В 2002 г. Глен Йех и Дженифер Видом представили миру модель под названием SimRank [5] - меру оценки подобия объектов, основанную на анализе взаимоотношения этих объектов друг с другом. Основной идеей SimRank считается фраза "два объекта похожи если на них ссылаются похожие объекты". Поскольку формулировка похожести задается через саму себя, то базой этой рекурсии служит утверждение "каждый объект максимально похож на самого себя".

Формально модель описывается следующим образом. Обозначим за $I(v)$ и $O(v)$ множества входящих и выходящих соседей узла v, а за $s(a,b)$ меру схожести двух узлов. Следую рекурсивному определению $s(a,b) = 1$, если $a = b$. Иначе

$$s(a,b) = \frac{C}{|I(a)||I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} s_i(I_i(a), I_j(b)) \quad (1)$$

где C - константа со значением из $(0, 1)$. Это константа называется *коэффициентом затухания* и является необходимой для реалистичной оценки. В своих вычислениях авторы используют C = 0,8. Опять же, из-за рекурсивности меры похожести на практике используется итеративная модель вычисления, определяя для первой итерации единицей меру похожести одинаковых объектов и нулем меру похожести разный объектов. На каждой итерации похожесть как бы "распространяется" по графу следуя формуле

$$R_{k+1}(a,b) = \frac{C}{|I(a)||I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} R_k(I_i(a), I_j(b)) \quad (2)$$

Йех и Видом[5] доказывают сходимость процесса, при котором $R_k(a,b) \to s(a,b)$, при $k \to \infty$, а так же утверждают, что при k = 5 итерациям, полученные значения можно считать достаточно точными для оценки значения похожести.

В их работе так же предлагается иной способ вычисления оценки схожести с помощью *пар случайных блужданий* (*random surfer-pairs model*). В этом случае, для графа G строится соответствующий ему граф G^2, в котором узлы представляют собой пары, составленные из узлов G, а дуга из (a,b) в (c,d) присутствует в G^2 только если в G есть дуга из a в c и из b в d. Предположим, что 2 пользователя случайно блуждают по графу, начиная из узлов a и b соответственно. Тогда утверждается, что оценка $s(a,b)$ схожести узлов a и b полученная SimRank совпадает с ожидаемым числом шагов, необходимым пользователям для того, чтобы встретиться в каком-либо узле. В этом случае формула оценки выглядит следующим образом:

$$s'(a,b) = \sum_{t:(a,b)\rightsquigarrow(x,x)} P[t]c^{l(t)} \quad (3)$$

Суммирование ведется по всем t - путь в G^2 из какого либо узла в одноточечный узел (в этом случае пользователи находятся в одном и том же узле начального графа G). $l(t)$ - длина пути, а константа c играет ту же роль, что и C в (1). Вероятность выбора пути t P[t] вычисляется формулой

$$P[t] = \prod_{i=1}^{k-1} \frac{1}{|O(w_i)|} \quad (4)$$

где w_i - промежуточный узел пути t.

Описывая SimRank, полезно упомянуть родственный ему алгоритм PageRank, который является первым и наиболее популярным алгоритмом для упорядочивания результатов поисковой системы Google. PageRank подсчитывает количество и качество ссылок на страницу и грубо оценивает,

насколько эта страница важна. В основе PageRank лежит предположение, что чем важнее вебсайт, тем больше на него ссылаются другие сайты.

На рис. 1 изображен пример применения PageRank для простой сети, результаты выражены в процентном соотношении. Страница C имеет показатель PageRank выше, чем у страницы E, несмотря на то, что к странице E ведут больше ссылок. На страницу C ссылается одна важная страница и, следовательно, эта ссылка ценится выше. Если блуждания, начавшиеся со случайной страницы имеют вероятность 85% выбора исходящей ссылки со страницы, на которой они находятся, и вероятность 15% перейти на любую случайную страницу из всей сети, то в 8.1% случаев они окажутся на странице E (без фактора случайного «прыжка» все блуждания заканчивались бы на страницах A, B или C).

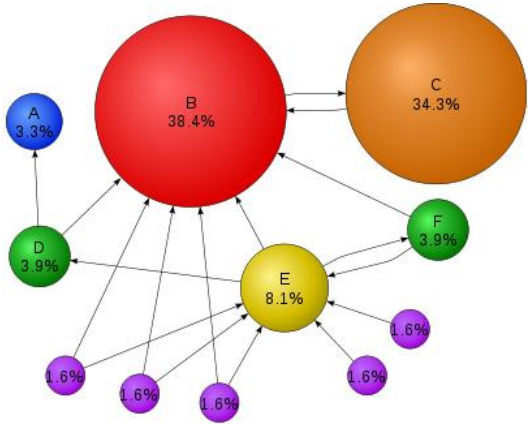


Рис. 1. Пример представления PageRank.

Fig. 1. PageRank representation.

Предложенный метод случайных блужданий для SimRank имеет вычислительную сложность в общем случае равную $O(n^4)$. Для оптимизации Йех и Видом предлагают оценивать схожесть пар, основываясь только на влиянии ближайших соседей (например на расстоянии 2-3 дуг), тогда сложность всего алгоритма составит $O(kn^2d^2)$, где d - средняя степень узла (не должна зависеть от n), а k - количество итераций. По утверждениям авторов, после всех оптимизаций, модель способна проводить вычисления для графов в пределах $n = 278{,}626$ объектов, что является практически неприемлемым в наше время ввиду того факта, что современные графовые базы данных насчитывают миллионы узлов и миллиарды связей. Так же стоит отметить такой ограничивающий фактор, как объем машинной памяти, требуемый для хранения результатов вычислений. В среднем потребуется порядка n^2 единиц памяти, что может составить терабайты для обработки баз данных с миллионом или более объектов.

За последние 15 лет для уменьшения вычислительных стоимостей был предложен целый ряд алгоритмов [6-13]. Для относительного сравнения этих работ удобно использовать табл. 1, представленную в работе Касумото *et al* [14], но для начала разберем основные типы задач для SimRank. 1) Одиночная пара: сосчитать значение похожести двух данных узлов *u* и *v*. 2) Одиночный источник: для данного узла *u* сосчитать ее значение похожести со всеми остальными узлами. 3) Все пары: сосчитать значение похожести для каждых двух узлов.

Табл. 1. Сложности алгоритмов для задач Simrank.

Table 1. Algorithmic complexities for Simrank problem types.

Алгоритм	Тип задачи	Временная сложность	Память	Метод
Kasumoto *et al* [14] (state-of-the-art)	Top-*k* поиск	$<<O(n)$	$O(m)$	Линейно-рекурсивная формулировка и Монте-Карло
Kasumoto *et al* [14] (state-of-the-art)	Top-*k* все пары	$<<O(n^2)$	$O(m)$	Линейно-рекурсивная формулировка и Монте-Карло
Li *et al* [9]	Одиноч. пара	$O(Td^2n^2)$	$O(n^2)$	Пары случайных проходов (итеративный)
Fogaras and Racz [6]	Одиноч. пара	$O(TR)$	$O(m+nR)$	Пары случайных проходов (Монте-Карло)
Jeh and Widom [5]	Все пары	$O(Td^2n^2)$	$O(n^2)$	Наивный
Lizorkin *et al* [11]	Все пары	$O(Tmin\{nm,n^3/\log n\})$	$O(n^2)$	Частичные суммы
Yu *et al* [13]	Все пары	$O(Tmin\{nm,n^w\})$	$O(n^2)$	Быстрое перемножение матриц
Li *et al* [8]	Все пары	$O(r^4n^2)$	$O(n^2)$	Разложение на сингулярные числа
Fujiwara *et al* [15]	Все пары	$O(r^4n)$	$O(r^2n^2)$	Разложение на сингулярные числа
Yu *et al* [10]	Все пары	$O(n^3+Tn^2)$	$O(n^2)$	Разложение на собственные числа

Касумото *et al* [14] используют матричную формулировку SimRank: пусть P - матрица перехода для графа G, т. е.,

$$P_{i,j} = \begin{cases} \dfrac{1}{|I(j)|}, & (i,j) \in E, \\ 0, & (i,j) \notin E \end{cases} \qquad (5)$$

Пусть S - матрица, содержащие значения похожести узлов, то есть $S_{i,j} = s(i,j)$. Тогда (1) в матричном виде перепишется как

$$S = \max\{c(P^{\mathsf{T}}SP), I\} \qquad (6)$$

Для сокращения вычислительных ресурсов Касумото *et al* предлагают алгоритм вычисления top-*k* узлов, основывающийся, на четырех пунктах.

1. Линейно-рекурсивная формулировка. Касумото *et al* [14] вводят диагональную матрицу *D:*

$$S = c(P^{\mathsf{T}} SP) + D \qquad (7)$$

и доказывают, что она существует и корректно определена. Эта матрица позволяет преобразовать нелинейные рекурсивные вычисления в правой части в сходящийся ряд и использовать его частичную сумму из T слагаемых для точной приближенной оценки значения схожести одиночной пары.

2. Метод Монте-Карло. Поскольку для задачи одиночного источника (значения схожести одного узла со всеми остальными) временная сложность линейной рекурсии составит $O(Tmn)$, что не является приемлемым, авторы используют метод Монте-Карло для выборки из R независимых пар случайных проходов. Здесь предлагается алгоритм, чья сложность сводится к $O(TR)$.

3. Зависимость от расстояния. Ключевое наблюдение SimRank: значение схожести между u и v затухает очень быстро с увеличением расстояния между этими узлами. Касумото *et al* [14] провели опыты с некоторыми реальными базами данных и выяснили, что top-1000 самых схожих узлов для случайных 100 узлов лежат не дальше чем на расстоянии в 7 дуг от исходного узла.

4. Верхняя оценка значения схожести. Этот и предыдущий пункты помогают сильно сократить временную сложность предлагаемого алгоритма с помощью верхней оценки значения схожести узлов, которая зависит только от расстояния между узлами. Здесь авторы находят 2 оценки в зависимости от того, высокая или низкая степень у рассматриваемого узла.

Собирая все пункты воедино, Касумото *et al* [14] предложили эффективный двухфазный алгоритм для вычисления top-k самых схожих узлов с заданным узлом. Первая фаза - фаза обработки - находит "кандидатов", которые могут иметь большое значение схожести, а вторая фаза уже проверяет этих "кандидатов", находит их точные значения схожести с заданным узлом и выдает ответ на top-k запрос.

Рассматривая задачу SimRank, важно уделить внимание случаю неопределенных графов. Неопределенный граф G можно рассматривать как распределение вероятностей для всех реальных исходов графа - детерминированных (определенных) графов G. Главным отличием неопределенных графов при вычислении коэффициентов похожести является несправедливость формулы для k-ой степени матрицы перехода. То есть $P^{(k)} \neq (P^{(1)})^k$.

Такое наблюдение сделал и доказал в своей статье Жу *et al* [16]. Он предложил новые обобщающие формулы для вероятностей случайных проходов по неопределенному графу и значений похожести узлов. Результатом его работы стали 3 предложенных алгоритма: 1) первый алгоритм с высокой точностью вычисляет необходимые n матриц перехода $P^{(1)}, P^{(2)}, \ldots, P^{(n)}$; 2) второй алгоритм использует формирование выборки для подсчета $P^{(1)}, P^{(2)}, \ldots, P^{(n)}$ с высокой эффективностью; 3)третий алгоритм объединяет приемы двух предыдущих и сравним по эффективности со вторым алгоритмом, но обладает на порядок величины меньшей погрешностью оценки.

3. Influence maximization.

Модели процессов, при которых информация распространяется по социальным сетям, изучаются и применяются уже довольно давно. В 2001 году Домингос и Ричардсон [17] поставили одну из фундаментальных алгоритмических задач для социальных сетей: если мы сможем убедить принять какой-либо продукт или новшество определенную группу людей с целью запустить череду принятия продукта или новшества другими людьми, то каким образом нужно выбрать эту определенную группу? Первыми, кто посмотрели на максимизацию влияния как на задачу оптимизации и предложили ее решение стали Кемпе *et al* [18]. Их работа смотивировала целый ряд исследований алгоритмов по сокращению вычислительных затрат [19-22], а сама задача максимизации влияния развивалась во многих направлениях: тематическая MB [23], пространственно-ориентированная MB [24,25], MB при наличии конкурентов [26,27]. В данной статье мы рассмотрим такие алгоритмы как *жадный* подход, *Reverse Influence Maximization*(*RIM*), *Two-phase Influence Maximization*(*TIM*) и *TIM+*, предложенные Кемпе *et al* [18], Боргс *et al* [28] и Танг *et al* [29] соответственно.

Кемпе *et al* [18] рассмотрел 2 модели диффузии (распространения влияния) в социальной сети: *independent cascade* (*IC*) и *linear threshold* (*LT*) модели. Здесь мы рассмотрим IC модель. Задача максимизации влияния ставится следующим образом:

Пусть G социальная сеть (граф отношений и взаимодействий) с набором узлов V и набором направленных дуг E. $|V| = n$, $|E| = m$. Предположим, что каждой направленной дуге e соответствует *вероятность распространения* (*propagation probability*) $p(e) \in [0,1]$. При данной G, модель *independent cascade* рассматривает пошаговое (относительно времени) распространение влияния следующим образом:

- На шаге 1, мы *активируем* выбранное множество узлов S из G, определяя все остальные узлы в G *неактивными.*
- Если узел u был активирован на шаге i, тогда для каждой дуги e, исходящей из u к неактивному узлу v, u имеет вероятность $p(e)$ активировать v на шаге $i + 1$. После шага $i + 1$, u больше не может активировать ни один узел.
- Если узел стал активным, то он остается таковым на всех последующих шагах.

Пусть $I(S)$ - количество активных узлов после завершения процесса (процесс останавливается, если ни один узел не может быть активирован). S называют начальным множеством (*seed set*), а $I(S)$ - распространением (*spread*) S.

При данном графе G, константном k, задача максимизации влияния при модели IC требует найти начальный набор S, $|S| = k$, с максимальным ожидаемым распространением $E[I(S)]$. Другими словами, мы ищем множество, которое с большей вероятностью сможет активировать наибольшее число узлов.

Подход, предложенный Кемпе *et al* (называемый жадным) фактически начинает с пустого начального множества $S = \varnothing$, а затем на каждой итерации добавляет в множество элемент u, при котором прирост распространения $E[I(S)]$ будет наибольшим.

Другими словами:

$$\arg\max_{v \in V} \left(E\left[I(S \cup \{v\})\right] - E\left[I(S)\right] \right) \quad (8)$$

Хотя *жадный подход* является довольно примитивным, вычисление ожидания распространения $E[I(S)]$ является *#P*-сложной задачей. В связи с этим, для адекватной оценки $E[I(S)]$ Кемпе *et al* используют метод Монте Карло: для каждой дуги e мы "подбрасываем монету" и с вероятностью $1 - p(e)$ убираем дугу e. Полученный граф назовем g, а $R(S)$ - множество узлов g, для которых существует путь из какого-либо узла S. Кемпе *et al* доказали, что ожидаемый размер $R(S)$ равен $E[I(S)]$. Таким образом, мы генерируем несколько экземпляров g, вычисляем $R(S)$ для каждого случая, а затем берем их среднее значение для оценки $E[I(S)]$.

При достаточно большом количестве экземпляров r для измерения $E[I(S)]$ *жадный* подход с достаточно большой вероятностью предоставляет $(1-1/e-\varepsilon)$ -приближенное решение при IC модели [18] где ε - константа, зависящая от G, и от r. Кемпе *et al* предлагает рассматривать r = 10000, и последние представленные работы используют этот же выбор.

Не смотря на то, что *жадный* подход является обобщенным и эффективным, он влечет за собой огромные вычислительные расходы, т. к. его временная сложность равняется $O(kmnr)$ (k итераций для r графов по n узлов и m дуг).

Только недавно (2014г.) Боргс *et al* [28] сделали теоретический прорыв и предложили алгоритм с временной сложностью $O\left(kl^2(m+n)\log^2 n/\varepsilon^3\right)$. Боргс *et al* показали, что их алгоритм предоставляет $(1-1/e-\varepsilon)$ - приближенное решение с как минимум $1-n^{-l}$ вероятностью и доказали, что этот результат является близким к оптимальному, так как любой алгоритм, который предоставляет такое приближение с хотя бы постоянной вероятностью должен работать за время $\Omega(m+n)$.

Основной причиной неэффективности *жадного* подхода является необходимость оценивать ожидаемое распространение множества узлов при каждой итерации алгоритма. Действительно, большинство из этих оценок проводятся впустую, так как при каждой итерации ищется набор узлов с наибольшим ожидаемым распространением. Таким образом, например, на первой итерации, при поиске первого узла для начального набора узлов, без каких-либо имеющихся данных об ожидаемом распространении узлов нам будет необходимо оценить $E[I(\{v\})]$ для каждого v, из G. В этом случае вычислительные расходы только первой итерации составляют $O(mnr)$.

Боргс *et al* [28] постарался избежать ограничений *жадного* подхода и предложил совершенно иной метод максимизации влияния для модели IC. В работе Танга *et al* [29], к которой мы обратимся позже, этот метод называют *Reverse Influence Sampling* (*RIS*). *RIS* основывается на понятии *Reverse Reachable Set* или *RR*(*v*) -- набор всех узлов *u*, из которых есть путь в *v* в *g*, где *g* - экземпляр вероятностного графа *G*. Боргс *et al* показали, что если *RR*(*v*) пересекается с каким-то набором узлов *S* с вероятностью *p*, то если мы выберем *S* в качестве начального множества, при распространении влияния в *G* мы будем иметь вероятность *p* активировать узел *v*. Основываясь на этих результатах, *RIS* выполняется в два шага:

1. Сгенерировать определенное количество случайных *RR* множеств из *G* (*RR* множество для случайного узла *v* из случайного экземпляра *g*)
2. Рассмотреть задачу максимального покрытия [30]: выбрать *k* узлов, которые покроют максимальное количество сгенерированных *RR* множеств. (*v покрывает S тогда и только тогда, когда* $v \in S$).

На интуитивном уровне этот алгоритм объясняется так: если узел принадлежит большому количеству *RR* множеств, то его ожидаемое распространение должно быть велико.

Несмотря на то, что сложность алгоритма является окололинейной, он может требовать большие вычислительные расходы в связи с необходимым количеством генерируемых *RR* множеств. Боргс *et al* предложили генерировать эти множества до тех пор, пока количество рассмотренных узлов и дуг не превысит порог τ. Они показали, что если для τ установлено значение $\Theta\left(k(m+n)\log^2 n/\varepsilon^3\right)$, то *RIS* работает за линейное по отношению к τ время и возвращает $(1-1/e-\varepsilon)$ -приближенное решение с хотя бы постоянной вероятностью. Затем они увеличивают вероятность успеха до хотя бы $1-n^{-l}$, увеличивая τ на множитель с l и запуская *RIS* $\Omega(l \log n)$ раз. Алгоритм не обладает практической эффективностью и требует больших временных затрат уже при обработке графов с десятками тысяч узлов и дуг. Множитель (ε^{-3}) в оценке временной сложности появляется из-за необходимости выбрать τ достаточно большим, т. к. при малом τ часто встречающиеся элементы *RR* множеств повлияют на выбор конечного ответа, и результат будет далек от оптимального.

Танг *et al* [29] предложил новый алгоритм *Two-phase Influence Maximization*(*TIM*), который использует идеи *RIS*, но обходит его ограничения, предлагая временную сложность $O\left((k+l)(m+n)\log n/\varepsilon^2\right)$. Такая сложность отличается на множитель ($\log n$) от сложности $\Omega(m+n)$, которая, как доказали Боргс *et al* [28], является нижней границей сложности любого алгоритма (при фиксированных *k*, *l* и ε). Как следует из названия и подобия алгоритму *RIS*, *TIM* состоит из 2х шагов:

1. <u>*Оценка параметра.*</u> Этот шаг оценивает нижнюю границу максимально

возможного ожидаемого распространения по всем k-размерным множествам узлов и использует эту границу для получения параметра θ.

2. *Выбор узлов.* На этом шаге генерируются θ случайных RR множеств из G, а затем выбирается k-размерное множество узлов S*, которое покрывает наибольшее число RR множеств. R* возвращается в качестве результата работы алгоритма.

Выбор узлов в *TIM* схож с выбором узлов в *RIS*, за исключением того, что здесь генерируется уже заранее определенное число случайных *RR* множеств. Выбор множества *S** происходит с помощью *жадного* подхода решения задачи *максимального покрытия*, т. е. выбором *k* узлов, *покрывающих* наибольшее количество множеств узлов. Этот подход возвращает $(1-1/e-\varepsilon)$ - приближенное решение и имеет реализацию с линейным временем:

1: *S* = ∅;*
2: *for j=1 to k do*
3: *Найти узел v(j), который покрывает наибольшее число RR множеств.*
4: *Добавить v(j) в S*.*
5: *Убрать все RR множества, покрытые v(j).*
6: *return S**

Для оценки параметра (шаг 1), авторы приводят и доказывают теорему:

Th: Если θ удовлетворяет неравенству

$$\theta \geq (8+2\varepsilon)n \cdot \frac{l\log n + \log C_k^n + \log 2}{OPT \cdot \varepsilon^2} \qquad (9)$$

то шаг 2 алгоритма TIM возвращает $(1-1/e-\varepsilon)$*-приближенное решение с хотя бы* $1-n^{-l}$ *вероятностью.*

Поскольку значение *θ* трудно выбрать согласно неравенству (так как значение *OPT* неизвестно) Танг *et al* [] находят нижнюю оценку значения *OPT* (называя ее *KPT*) *и* используют ее в конечной формуле для:

$$\theta = \lambda / KPT \qquad (10)$$

где

$$\lambda = (8+2\varepsilon)\frac{l\log n + \log C_k^n + \log 2}{\varepsilon^2} \qquad (11)$$

Отличие *TIM+* от *TIM* заключается в том, что *TIM+* добавляет промежуточный шаг для улучшения *KPT* (нижней оценки *OPT*), что влечет за собой некоторое небольшое уменьшение вычислительных расходов алгоритма.

4. Pattern Matching.

Проблема поиска и сопоставления с образцом в графовых базах данных имеет широкое применение в таких областях как социальные сети, компьютерный дизайн, машинное зрение, электроника и биология. Из-за разнообразия в характеристиках графов этих областей, а так же различия постановок проблемы, графовое сопоставление с образцом объединяет в себе целый набор взаимосвязанных задач.

Основная цель *pattern matching* – поиск всех вхождений заданного шаблона в графовой базе данных. Формально:

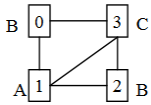
- *Граф* $G = (V, E)$, с множеством вершин V и множеством дуг E. Вершины и/или дуги могут быть помечены и/или иметь различные атрибуты.
- *Граф-образец* **(**или *образец для запроса***)** $P = (V_p, E_p)$ определяет структурные и семантические требования, которыми должен обладать подграф M графа G, чтобы соответствовать образцу P.

Требуется найти множество всех подграфов M, «соответствующих» шаблону P. Чаще всего, сопоставление с образцом описывается в терминах *изоморфизма подграфов*, что, как известно, является NP-сложной задачей. Как описывает Галахер [31] в своем обзоре работ по этой проблеме, все известные алгоритмы поиска изоморфизма подграфа имеют экспоненциальную сложность от размера графа, что не позволяет напрямую решать эту задачу на больших графах. В качестве возможного решения предлагаются либо аппроксимирующие алгоритмы поиска изоморфизма, либо точные алгоритмы, но применяемы только к конкретной части всего объема данных. Второй подход обычно достигается предварительной обработкой, которая позволяет отбросить части данных, которые наверняка не принесут результата. Эта фильтрация данных обычно называется *выборкой кандидатов.*

Первыми такой подход использовали в своем алгоритме *GraphGrep* Шаша *et al* [32]. Алгоритм содержит 3 компоненты: (1) построение метаданных для графа - индексирование с помощью наборов путей (этот шаг выполняется только единожды), (2) фильтрование базы данных на основе полученного шаблона и индексирования с целью уменьшения пространства поиска, (3) выполнение точного поиска.

1. Построение индекса. На этом этапе строится «представление графа с помощью путей». Авторы рассматривают базу данных из нескольких графов, что можно интерпретировать как несколько компонент связности одного графа. Так как каждый узел имеет метку (*label*) и уникальный идентификатор (*id*), то «представление с помощью путей» является набором путей меток или label-path (пути длиной от 1 до l_p, Shasha *et al* [32] берут значение $l_p = 4$), где каждый путь меток – набор из путей идентификаторов или id-path. Эти данные формируют хэш-таблице, где ключами являются хэш-значения путей меток, а в ячейки записываются количества путей идентификаторов по этому ключу. Такую хэш-

таблицу для графа авторы называют сигнатурой (*fingerprint*) графа. Пример построения показан на рис. 2.



(a)

A={(1)} AB={(1, 0), (1,2)} AC ={(1, 3)} ACBA={…}
ABCA={(1 ,0 ,3 ,1),(1, 2, 3, 1)} CB={(3,0),(3,2)} C={(3)}
CBAB={((3, 0, 1 ,2),(3 ,2 ,1 ,0)} B={(0),(2)} BA={(0,1),(2,1)}
BAB={(0,1,2), (2,1,0)} ABC ={(1, 3, 0), (1,3,2)} ACB={…}
ABCB={…} BC={…} BAC={…} BCB={…} CBA={…}
BABC={…}CBAC={…} CABC={…} CAB={(3,1,0), (3,1,2)}
BACB={…}BCBA={…}BCAB={…} BCA={…} CA={(3,1)}

(b)

Key	g_1
h(CA)	1
……	
h(ABCB)	2

(c)

Рис. 2. (*a*) *входной граф* g_1, (*b*) *его представление с помощью путей и* (*c*) *сигнатура.*

Fig. 2. (*a*) *input graph* g_1, (*b*) *its path representation and* (*c*) *fingerprint*.

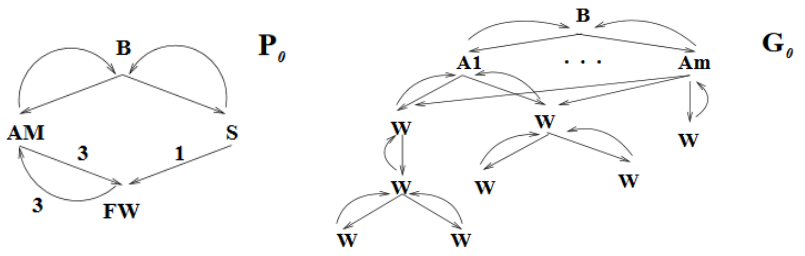
Пусть n и k количество узлов и максимальная степень узла в графе соответственно. Тогда в худшем случае сложность индексирования и представления с помощью путей составляет $O(nk^{l_p})$, а занимаемая память $O(l_p n k^{l_p})$ для каждой отдельной компоненты связности.

2. Выбор кандидатов. На этом этапе строится сигнатура графа-шаблона и сравнивается с сигнатурой графа данных. Если граф данных состоит из нескольких компонент связности (для каждой из которых мы предположительно построили сигнатуру), то компоненты, в чьей сигнатуре хотя бы одно значение меньше соответствующего значения в сигнатуре шаблона, можно отбросить. Остальные компоненты содержат одно или более вхождение подграфа, соответствующего образцу (шаблону).

3. Поиск соответствующего подграфа. Алгоритм обходит граф-образец в глубину и разбивает ветви обхода на последовательности (называемые *patterns*) накладывающихся друг на друга путей меток. Части графа-кандидата, чьи пути идентификаторов соответствуют этим последовательностям, склеиваются (избавляясь от наложений) для построения соответствующего образцу подграфа. Вычислительная сложность этого шага зависит от числа последовательностей (*patterns*) образца, которое сложно оценить в общем случае. Грубо говоря, оно прямо пропорционально размеру образца и максимальной степени среди узлов в образце и чем больше l_p, тем меньше p.

Если $\overline{n}$ – максимально число узлов с одной и той же меткой, то временная сложность этого шага в худшем случае составляет $O((\overline{n}k^{l_p})^p)$.

Задача сопоставления с образцом, описанная на языке изоморфизмов графа является NP-сложной. Это сильно затрудняет масштабируемость при поиске точных вхождений образца. Более того, требуется нахождение биекций, которые чаще всего ставят строгие ограничения на типы образцов как показано в примере, приведенном в [33].



(*a*) (*b*)

Рис. 3. Организация оборота наркотиков: (*a*) *граф-образец и* (*b*) *граф данных*

Fig. 3. Drug trafficking. Pattern (*a*) *and data graph* (*b*)

Пример. Рассмотрим структуру организации оборота наркотиков, изображенную в виде графа-образца P_0 на рис. 3 (*a*). Глава («boss» - B) осуществляет надзор и ведет контроль операций через группу управляющих помощников (assistant manager - AM). AM контролируют деятельность иерархической структуры работников (field worker – FW) вплоть до трех уровней иерархии, обозначенных меткой дуги «3». FW доставляет наркотики, собирает выручку и выполняет другие поручения. Они отчитываются перед AM напрямую или косвенно, а AM напрямую отчитываются перед боссом. Босс так же может передавать сообщения через секретаря (S) верхнему уровню FW (обозначено меткой дуги «1»).

Рассмотрим группировку по сбыту наркотиков, представленную графом на рис. 3 (*b*), где $A_1,\ldots A_m$ это AM'ы, в то время как A_m является и AM и секретарем (S).

Мы хотим определить всех подозреваемых, которые входят в группировку, с помощью поиска вхождений P_0 в G_0. Однако сопоставление с образцом через изоморфизм подграфов не справится с этой задачей по следующим причинам.

- Узлы AM и S из P_0 должны соответствовать одному и тому же узлу A_m из G_0, что противоречит биекции.
- Узел AM из P_0 соответствует нескольким узлам из G_0 ($A_1,\ldots,A_m$). Это отношение не может быть описано функцией от узлов P_0 в узлы G_0.
- Дуга из AM в FW в P_0 означает, что AM контролирует 3 уровня FW. Это должно соответствовать пути определенной длины в G_0, а не одной дуге. Изоморфизм, при котором дуга переходит в дугу не подойдет в этой ситуации.

К таким выводам пришел B. Фан *et al* [34]. В 2010 г. Он опубликовал работу, в которой пересмотрел практически целиком задачу сопоставления с образцом, введя понятие ограниченной симуляции (*bounded simulation*). Согласно его терминологии:

- *Граф данных* $G = (V, E, f_A)$, где f_A – функция от узлов, при которой

$f_A(u)$ – кортеж ($A_1=a_1,\ldots,A_n=a_n$), где a_i – константа (не обязательно числовая), а A_i – атрибут u, записываемый как $u.A_i=a_i$.

- <u>*Граф-образец*</u> $P = (V_p, E_p, f_v, f_e)$. f_v – функция от узлов образца, сопоставляющая узлу предикат, составленный из объединения атомарных формул вида A op a, где a – константа, A – атрибут, а op - оператор сравнения ($<, \leq, =, \neq, >, \geq$). f_e – функция от дуг образца, такая что $f_e(u,u')$ либо константа k, либо символ «*».

Тогда вводится следующее определение ограниченной симуляции:

<u>Опр.</u> *Граф G соответствует образцу P в терминах ограниченной симуляции, если существует бинарное отношение* $S \subseteq V_p \times V$, *т. ч.:*

- *Для каждого* $u \in V_p$ *существует* $v \in V$, *т. ч.* $(u,v) \in S$;
- *Для каждого* $(u,v) \in S$ (*а*) *атрибут узла* v $f_A(v)$ *удовлетворяет предикату* $f_v(u)$ *узла u. То есть для каждой атомарной формулы A* op *a из* $f_v(u)$, *v.A = a′ определено в* $f_A(v)$ *и a′* op *a.* (*б*) *Для каждой дуги* (u,u') *из* E_p *существует непустой путь* $\rho = v/\ldots/v'$ *из G, т. ч.* $(u,u') \in S$ *и длина* len (ρ) $\leq$ *k, если* $f_e(u,u')$ *это константа k.*

Такое бинарное отношение *S* называют вхождением (*match*) *P* в *G* и обозначают как $P \trianglelefteq G$.

Интуитивно, $(u,v) \in S$, если v соответствует критериям поиска, установленным $f_v(u)$, и каждая исходящая из u дуга (u,u') соответствует непустому пути ρ, длины $f_e(u,u')$, если $f_e(u,u') = k$, или неограниченной длины, если $f_e(u,u') = *$.

Поскольку вхождение *P* в *G* может быть несколько, Фан *et al* [30] ставят и доказывают утверждение:

<u>Утв.</u> *Для любого графа G и любого образца P, если* $P \trianglelefteq G$, *то существует единственное максимальное вхождение* S_M, *т. е. для любого вхождения S образца P в граф G верно, что* $S \subseteq S_M$.

На основе введенных определений графа, образца и ограниченной симуляции, авторы пересматриваю задачу сопоставления с образцом следующим образом: *Для данного графа G и образца P требуется найти максимальное вхождение P в G, если* $P \trianglelefteq G$.

Фан *et al* [34] приводят алгоритм, имеющий временную сложность

$$O\left(|V|\,|E| + |E_p|\,|V|^2 + |V_P|\,|V|\right) \qquad (12)$$

Такой алгоритм, в отличие от NP-сложных алгоритмов изоморфизма подграфов, завершается за кубическое время (т. к. в худшем случае $|E|$ это $O(|V^2|)$).

Так как кубическая сложность все еще все еще не подходит для больших графов, Фан *et al* [34] предлагают алгоритмы приращений (*incremental match algorithms*) для поиска вхождений в случае, когда граф G изменяется посредством удаления и вставки дуг. Это позволяет найти вхождение единожды, а затем его эффективно обновлять при изменении G, без нужды запускать весь процесс самого начала.

5. Network motifs.

Множество биологических сетей, представляемых с помощью графов, содержат определенные подсети (подграфы), которые появляются в данной сети с гораздо большей частотой, чем в случайных сетях. Например, в сети взаимодействия протеин-протеин некоторые трех- и четырехузельные подграфы встречаются гораздо чаще, чем в случайной сети со схожими математическими свойствами. В 2002 г. Мило *et al* [35] предложил использовать такие «топологические блоки» для изучения структуры и строения сложных сетей, назвав такие блоки *сетевыми мотивами* (*network motifs*). С тех пор было проведено много исследований на эту тему, некоторые из которых ориентировались на биологической составляющей сетевых мотивов, другие – на теории алгоритмов их поиска. В данной статье не будет обсуждаться само понятие сетевых мотивов, а будут рассмотрены некоторые алгоритмы, которые позволяют анализировать сложные природные и другие сети на предмет нахождения в них сетевых мотивов.

Процесс обнаружения сетевых мотивов обычно состоит из двух шагов: 1) генерация набора случайных сетей, чьи свойства зависят от свойств сети, в которой производится поиск (называемой *настоящая сеть*), 2) подсчет вхождений подграфов в настоящей и случайных сетях. Мило *et al* [35] в своей работе описал несколько сложных графов сетей, встречающихся в природе, содержащих сетевые мотивы. Его работа была ориентирована на обоснование важности задач, которую эти мотивы могут выполнять, а для вычислений использовала метод полного перебора всех возможных подграфов сети с данным количеством узлов. Очевидно, что временная сложность такого алгоритма очень быстро растет с увеличением количества узлов подграфа и ростом самого графа данных. Такой алгоритм справлялся с поиском небольших сетевых мотивов, но нахождение пяти- или шестиузловых мотивов не представлялось вычислительно возможным.

Первым алгоритмом, отличающимся от простого перебора стал *mfinder* – алгоритм, основанный на методе генерации выборки, предложенный Каштан *et al* [36]. Алгоритм оценивает концентрации подграфов направленной или ненаправленной сети, из которых можно сделать вывод о наличии или отсутствии сетевых мотивов. Выборка начинается с выбора произвольной дуги

графа, таким образом порождая подграф размера 2, а затем увеличивает подграф, добавляя в него случайную касающуюся его дугу. Затем, алгоритм продолжает выбирать случайные соседние дуги, пока размер подграфа не достигнет *n*. Наконец, в отобранный подграф добавляются все дуги между полученными *n* узлами. Используя такой подход, необходимо использовать несмещенную выборку, поскольку процедура выборки составляет экземпляры неоднородно. Для этого, Каштан *et al* [36] предложил систему взвешенных подграфов, используя вероятность генерации подграфа в процессе выборки и сопоставляя каждому экземпляру соответствующий вес. Таким образом, вероятные подграфы получают сравнительно меньший вес, чем маловероятные подграфы, обеспечивая тем самым справедливую оценку концентраций подграфов. Так, если положить *P* – вероятность генерации образца подграфа типа *i*, $W = 1/P$, тогда S_i накапливает весовой показатель для подграфов типа *i*: $S_i = S_i + W$. После генерации всех экземпляров, полагая, что было сгенерировано *L* различных типов подграфов, то концентрация подграфа типа *i* вычисляется формулой:

$$C_i = \frac{S_i}{\sum_{k=1}^{L} S_k} \tag{13}$$

Вычислительная сложность такого алгоритма асимптотически независима от размера графа данных и составляет $O(n^n)$ для каждого образца подграфа размером *n*. Но необходимо упомянуть, что процесс выборки может сгенерировать один и тот же подграф несколько раз, тратя время без получения какой-либо информации. Несмотря на это, алгоритм с выборкой является более эффективным подходом, нежели случайный перебор, хоть и оценивает концентрации лишь примерно. Алгоритм может находить мотив вплоть до размера 6 узлов.

За прошедшее десятилетие были предложены такие алгоритмы, как *FANMOD*[37], *Grochow-Kellis*[38], *FPF*[39], *MODA*[40], *G-trie*[41]. Среди упомянутых *g-trie* является самым быстрым, используя новую структуру хранения набора подграфов, предложенную Педро Рибейро и Фернандо Силва в 2010г. *G-trie* это многопутевое дерево, в котором каждая вершина хранит информацию об одном узле графа данных и соответствующих дугах, ведущих к его предкам. Путь от корня к вершине дерева соответствует конкретному графу, а потомки вершины дерева содержат одинаковый подграф. Построение *g-trie* подробно описано в [41]. Основной идеей подсчета количества вхождений подграфа является обратный обход дерева по всем возможным подграфам, при котором одновременно проводятся проверки на изоморфизм подграфов. Важное достоинство такого подхода к хранению при поиске сетевых мотивов заключается в том, что нет необходимости подсчитывать количество вхождений подграфа в случайную сеть, если такой подграф не присутствует в основной сети. Однако главным недостатком *g-trie* является

большой объем используемой памяти, что может ограничить размер находимых мотивов при использовании персонального компьютера со средним объемом памяти.

Стоит отметить еще одно направление развития теории сетевых мотивов – мотивы динамических сетях. С ростом популярности социальных сетей появился и интерес к нахождению сетевых мотивов в динамических сетях. Такие сети чаще всего предлагают временные метки для каждой дуги своего графа. В этом случае усложняется понятие «соседних дуг», так как, если дуга обозначает взаимодействие между двумя пользователями, то соприкасающиеся дуги считаются «соседними» только если их временные метки находятся друг от друга на расстоянии не больше чем некоторый временной порог ΔT. Существующие алгоритмы поиска мотивов статических графов игнорируют временной аспект динамических сетей, поэтому не решают поставленную задачу.

На данный момент, самым современным алгоритмом поиска сетевых мотивов в динамических сетях считается *COMMIT*, представленный Гурукар *et al* [42] в 2015г., который, согласно проведенным экспериментам, на 2 порядка величины быстрее представленных ранее алгоритмов. В целях экономии времени в данной статье *COMMIT* будет представлен только вкратце. За подробным описанием заинтересованный читатель может обратиться к оригинальной статье [42].

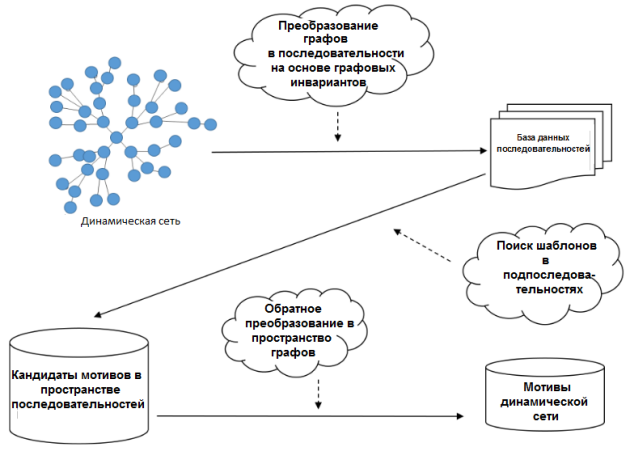


Рис. 4. Конвейер алгоритма COMMIT.

Fig. 4.Pipeline of COMMIT

На рис. 4 представлен конвейер алгоритма COMMIT. Основной идея алгоритма заключается в первом шаге: перевод каждой временно связанной компоненты (компонента связности, в которой метки дуг находятся в пределах одного временного периода, ограниченного порогом ΔT) в математическую последовательность ее взаимодействий. Здесь происходит необходимый анализ

полученной базы данных последовательностей и поиск в ней часто встречающихся шаблонов подпоследовательностей, которые могут представлять мотивы сети. Затем, эти подпоследовательности преобразовываются обратно в пространство графов для проверки и вычисления конечного результата. Такой подход к нахождению мотивов приводит к экономии времени, так как большинство вычислений проходят именно в пространстве последовательностей, чей размер значительно меньше размера пространства графов. Также, проверки на изоморфизм проводятся только на небольшой доле графов-кандидатов на статус мотива сети, полученных в ходе анализа.

Заключение

Таким образом, приведенный анализ показывает, что описанные методы с некоторыми оговорками позволяют решить поставленные задачи. Предложенные алгоритмы можно считать масштабируемыми, однако усовершенствование их временных сложностей как всегда является одним из приоритетных направлений дальнейших исследований. В качестве альтернативного пути развития отметить разработку инкрементальных версий алгоритмов. Такая версия алгоритма уже реализована для решения задачи максимизации влияния.

Далее, помимо рассмотренных в обзоре задач на графовых базах данных серьезный интерес представляют задачи поиска эффективных алгоритмов кластеризации и классификации, задачи статистического и пространственного анализа, визуализация данных. Хотя в настоящее время интерес специалистов к этим задачам менее выражен, их полное решение позволило бы резко повысить эффективность анализа и хранения больших данных.

Список литературы

[1]. Pettey C., Goasduff L. (2011) Gartner, Inc. Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data (online publication). Available at: http://www.gartner.com/newsroom/id/1731916, accessed 07.08.2016

[2]. DIS Group, (2014) “Big data.” http://www.dis-group.ru/solutions/data_ management/big_data/, accessed 07.08.2016.

[3]. Бартенев М., Вишняков И . “Использование графовых баз данных в целях оптимизации анализа биллинговой информации,” Инженерный журнал: наука и инновации, no. 11, 2013.

[4]. Manyika J., Chui M., Brown B., Bughin J., Dobbs R., Roxburgh C., Byers A. H . “Big data: The next frontier for innovation, competition, productivity,” 2011.

[5]. Jeh G., Widom J . “Simrank: a measure of structural-context similarity,” in Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 538–543, ACM, 2002.

[6]. Fogaras D. and R´acz B . “Scaling link-based similarity search,” in Proceedings of the 14th international conference on World Wide Web, pp. 641–650, ACM, 2005.

[7]. He G., Feng H., Li C., Chen H . “Parallel simrank computation on large graphs with iterative aggregation,” in Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 543–552, ACM, 2010.
[8]. Li C., Han J., He G., Jin X., Sun Y., Yu Y., Wu T . “Fast computation of simrank for static and dynamic information networks,” in Proceedings of the 13th International Conference on Extending Database Technology, pp. 465–476, ACM, 2010.
[9]. Li P., Liu H., Yu J. X., He J., Du X . “Fast single-pair simrank computation.,” in SDM, pp. 571–582, SIAM, 2010.
[10]. Yu W., Lin X., Le J . “Taming computational complexity: Efficient and parallel simrank optimizations on undirected graphs,” in International Conference on Web-Age Information Management, pp. 280–296, Springer, 2010.
[11]. Lizorkin D., Velikhov P., Grinev M., Turdakov D . “Accuracy estimate and optimization techniques for simrank computation,” The VLDB Journal The International Journal on Very Large Data Bases, vol. 19, no. 1, pp. 45–66, 2010.
[12]. Yu W., Lin X., Zhang W., Chang L., Pei J . “More is simpler: Effectively and efficiently assessing node-pair similarities based on hyperlinks,” Proceedings of the VLDB Endowment, vol. 7, no. 1, pp. 13–24, 2013.
[13]. Yu W., Zhang W., Lin X., Zhang Q., Le J . “A space and time efficient algorithm for simrank computation,” World Wide Web, vol. 15, no. 3, pp. 327–353, 2012
[14]. Maehara T., Kusumoto M., Kawarabayashi K.-i . “Scalable simrank join algorithm,” in 2015 IEEE 31st International Conference on Data Engineering, pp. 603–614, IEEE, 2015.
[15]. Fujiwara Y., Nakatsuji M., Shiokawa H., Onizuka M . “Efficient search algorithm for simrank,” in Data Engineering (ICDE), 2013 IEEE 29th International Conference on, pp. 589–600, IEEE, 2013.
[16]. Zhu R., Zou Z., Li J . “Simrank computation on uncertain graphs,” in 2016 IEEE 32nd International Conference on Data Engineering (ICDE), pp. 565–576, May 2016.
[17]. Domingos P., Richardson M . “Mining the network value of customers,” in Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 57–66, ACM, 2001.′
[18]. Kempe D., Kleinberg J., Tardos E . “Maximizing the spread of influence through a social network,” in Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 137–146, ACM, 2003.
[19]. Chen W., Wang C., Wang Y . “Scalable influence maximization for prevalent viral marketing in large-scale social networks,” in Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 1029–1038, ACM, 2010.
[20]. Goyal A., Bonchi F., L. Lakshmanan V . “A data-based approach to social influence maximization,” Proceedings of the VLDB Endowment, vol. 5, no. 1, pp. 73– 84, 2011.
[21]. Jung K., Heo W., Chen W . “Irie: Scalable and robust influence maximization in social networks,” in 2012 IEEE 12th International Conference on Data Mining, pp. 918–923, IEEE, 2012.
[22]. Kim J., Kim S.-K., Yu H . “Scalable and parallelizable processing of influence maximization for large-scale social networks?,” in Data Engineering (ICDE), 2013 IEEE 29th International Conference on, pp. 266–277, IEEE, 2013.
[23]. Kim D., Lee J.-G., Lee B. S . “,”
[24]. Li G., Chen S., Feng J., Tan K.-l., Li W.-s . “Efficient location-aware influence maximization,” in Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pp. 87–98, ACM, 2014.

[25]. Wang X., Zhang Y., Zhang W., Lin X . “Distance-aware influence maximization in geo-social network,”
[26]. Khan A., Zehnder B., Kossmann D . “Revenue maximization by viral marketing: A social network host’s perspective.”
[27]. Li H., Bhowmick S. S., Cui J., Gao Y., Ma J . “Getreal: Towards realistic selection of influence maximization strategies in competitive networks,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1525– 1537, ACM, 2015.
[28]. Borgs C., Brautbar M., Chayes J., Lucier B . “Maximizing social influence in nearly optimal time,” in Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 946–957, Society for Industrial and Applied Mathematics, 2014.
[29]. Tang Y., Xiao X., Shi Y . “Influence maximization: Near-optimal time complexity meets practical efficiency,” in Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pp. 75–86, ACM, 2014.
[30]. Mahdian M., Ye Y., Zhang J . “Improved approximation algorithms for metric facility location problems,” in International Workshop on Approximation Algorithms for Combinatorial Optimization, pp. 229–242, Springer, 2002.
[31]. Gallagher B . “Matching structure and semantics: A survey on graph-based pattern matching,” AAAI FS, vol. 6, pp. 45–53, 2006.
[32]. Giugno R., Shasha D . “Graphgrep: A fast and universal method for querying graphs,” in Pattern Recognition, 2002. Proceedings. 16th International Conference on, vol. 2, pp. 112–115, IEEE, 2002.
[33]. Natarajan M . “Understanding the structure of a drug traffcking organization: a conversational analysis,” Crime Prevention Studies, vol. 11, pp. 273–298, 2000.
[34]. Fan W., Li J., Ma S., Tang N., Wu Y., Wu Y . “Graph pattern matching: From intractable to polynomial time,” Proc. VLDB Endow., vol. 3, pp. 264–275, Sept. 2010.
[35]. Milo R., Shen-Orr S., Itzkovitz S., Kashtan N., Chklovskii D., Alon U . “Network motifs: simple building blocks of complex networks,” Science, vol. 298, no. 5594, pp. 824–827, 2002.
[36]. Kashtan N., Itzkovitz S., Milo R., Alon U . “Mfinder tool guide,” Department of Molecular Cell Biology and Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot Israel, Tech Rep, 2002.
[37]. Wernicke S., Rasche F . “Fanmod: a tool for fast network motif detection,” Bioinformatics, vol. 22, no. 9, pp. 1152–1153, 2006.
[38]. Grochow J. A., Kellis M . “Network motif discovery using subgraph enumeration and symmetry-breaking,” in Annual International Conference on Research in Computational Molecular Biology, pp. 92–106, Springer, 2007.
[39]. Schreiber F., Schwobbermeyer H . “Frequency concepts and pattern detection for the analysis of motifs in networks,” in Transactions on computational systems biology III, pp. 89–104, Springer, 2005.
[40]. Omidi S., Schreiber F., Masoudi-Nejad A . “Moda: an efficient algorithm for network motif discovery in biological networks,” Genes & genetic systems, vol. 84, no. 5, pp. 385–395, 2009.
[41]. Ribeiro P., Silva F . “G-tries: an efficient data structure for discovering network motifs,” in Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 1559–1566, ACM, 2010.
[42]. Gurukar S., Ranu S., Ravindran B . “Commit: A scalable approach to mining communication motifs from dynamic networks,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 475–489, ACM, 2015.

Some problems on graph databases

R.I. Guralnik <guralnikr@gmail.com>
Saint-Petersburg State University,
University Embankment, 7-9, St Petersburg, 199034, Russia

Abstract. Graph databases appear to be the most popular and relevant among non-relational databases. Its popularity is caused by its relatively easy implementation in the problems in which data have big numbers of relations such as protein-protein interaction and others. With the development of fast internet connection, graph database found another interesting application in representation of social networks. Moreover, graph edges are storable which lowers graph traversing calculation costs. Such system appeared to be natural and in-demand in the era of Internet and social networks. The most significant by size and matter section of graph databases problems is data mining. It contains such problems as associative rules learning, data classification and categorization, clustering, regression analysis etc. In this review, data mining graph database problems are considered which are most commonly presented in modern literature. Their popularity is represented by the big number of publications on these problems on several recent years' major conferences. Such problems as influence maximization, motif mining, pattern matching and simrank problems are examined. For every type of a problem we analyzed different papers and described basic algorithms which were offered 10-15 years ago. We also considered state-of-the-art solutions as well as some important in-between versions. This review consists of 6 sections. Besides introduction and conclusion, each section is dedicated to its own type of graph database problem.

Keywords: graph databases; motif mining; influence maximization; pattern matching; simrank.

DOI: 10.15514/ISPRAS-2016-28(4)-12

For citation: R.I. Guralnik. Some problems on graph databases. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 4, 2016, pp. 193-216 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-12

References

[1]. Pettey C., Goasduff L. (2011) Gartner, Inc. Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data (online publication). Available at: http://www.gartner.com/newsroom/id/1731916, accessed 07.08.2016

[2]. DIS Group, (2014) "Big data." http://www.dis-group.ru/solutions/data_ management/big_data/, accessed 07.08.2016.

[3]. Bartenev M.V., Vishnyakov I.E. "Graph database usage to optimize analysis of billing information", Engineering Journal: Science and Innovations [Inzhenernyi zhurnal: nauka i innovatsii], issue 11, 2013. Available at: http://engjournal.ru/catalog/it/hidden/1058.html

[4]. Manyika J., Chui M., Brown B., Bughin J., Dobbs R., Roxburgh C., Byers A. H . "Big data: The next frontier for innovation, competition, productivity," 2011.

[5]. Jeh G., Widom J . "Simrank: a measure of structural-context similarity," in Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 538–543, ACM, 2002.

[6]. Fogaras D. and R´acz B . “Scaling link-based similarity search,” in Proceedings of the 14th international conference on World Wide Web, pp. 641–650, ACM, 2005.
[7]. He G., Feng H., Li C., Chen H . “Parallel simrank computation on large graphs with iterative aggregation,” in Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 543–552, ACM, 2010.
[8]. Li C., Han J., He G., Jin X., Sun Y., Yu Y., Wu T . “Fast computation of simrank for static and dynamic information networks,” in Proceedings of the 13th International Conference on Extending Database Technology, pp. 465–476, ACM, 2010.
[9]. Li P., Liu H., Yu J. X., He J., Du X . “Fast single-pair simrank computation.,” in SDM, pp. 571–582, SIAM, 2010.
[10]. Yu W., Lin X., Le J . “Taming computational complexity: Efficient and parallel simrank optimizations on undirected graphs,” in International Conference on Web-Age Information Management, pp. 280–296, Springer, 2010.
[11]. Lizorkin D., Velikhov P., Grinev M., Turdakov D. “Accuracy estimate and optimization techniques for simrank computation,” The VLDB Journal The International Journal on Very Large Data Bases, vol. 19, no. 1, pp. 45–66, 2010.
[12]. Yu W., Lin X., Zhang W., Chang L., Pei J . “More is simpler: Effectively and efficiently assessing node-pair similarities based on hyperlinks,” Proceedings of the VLDB Endowment, vol. 7, no. 1, pp. 13–24, 2013.
[13]. Yu W., Zhang W., Lin X., Zhang Q., Le J . “A space and time efficient algorithm for simrank computation,” World Wide Web, vol. 15, no. 3, pp. 327–353, 2012
[14]. Maehara T., Kusumoto M., Kawarabayashi K.-i . “Scalable simrank join algorithm,” in 2015 IEEE 31st International Conference on Data Engineering, pp. 603–614, IEEE, 2015.
[15]. Fujiwara Y., Nakatsuji M., Shiokawa H., Onizuka M . “Efficient search algorithm for simrank,” in Data Engineering (ICDE), 2013 IEEE 29th International Conference on, pp. 589–600, IEEE, 2013.
[16]. Zhu R., Zou Z., Li J . “Simrank computation on uncertain graphs,” in 2016 IEEE 32nd International Conference on Data Engineering (ICDE), pp. 565–576, May 2016.
[17]. Domingos P., Richardson M . “Mining the network value of customers,” in Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 57–66, ACM, 2001.´
[18]. Kempe D., Kleinberg J., Tardos E . “Maximizing the spread of influence through a social network,” in Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 137–146, ACM, 2003.
[19]. Chen W., Wang C., Wang Y . “Scalable influence maximization for prevalent viral marketing in large-scale social networks,” in Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 1029–1038, ACM, 2010.
[20]. Goyal A., Bonchi F., L. Lakshmanan V . “A data-based approach to social influence maximization,” Proceedings of the VLDB Endowment, vol. 5, no. 1, pp. 73– 84, 2011.
[21]. Jung K., Heo W., Chen W . “Irie: Scalable and robust influence maximization in social networks,” in 2012 IEEE 12th International Conference on Data Mining, pp. 918–923, IEEE, 2012.
[22]. Kim J., Kim S.-K., Yu H . “Scalable and parallelizable processing of influence maximization for large-scale social networks?,” in Data Engineering (ICDE), 2013 IEEE 29th International Conference on, pp. 266–277, IEEE, 2013.
[23]. Kim D., Lee J.-G., Lee B. S . “Topical influence modeling via topic-level interests and interactions on social curation services,”

[24]. Li G., Chen S., Feng J., Tan K.-l., Li W.-s . “Efficient location-aware influence maximization,” in Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pp. 87–98, ACM, 2014.
[25]. Wang X., Zhang Y., Zhang W., Lin X . “Distance-aware influence maximization in geo-social network,”
[26]. Khan A., Zehnder B., Kossmann D . “Revenue maximization by viral marketing: A social network host’s perspective.”
[27]. Li H., Bhowmick S. S., Cui J., Gao Y., Ma J . “Getreal: Towards realistic selection of influence maximization strategies in competitive networks,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1525– 1537, ACM, 2015.
[28]. Borgs C., Brautbar M., Chayes J., Lucier B . “Maximizing social influence in nearly optimal time,” in Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 946–957, Society for Industrial and Applied Mathematics, 2014.
[29]. Tang Y., Xiao X., Shi Y . “Influence maximization: Near-optimal time complexity meets practical efficiency,” in Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pp. 75–86, ACM, 2014.
[30]. Mahdian M., Ye Y., Zhang J . “Improved approximation algorithms for metric facility location problems,” in International Workshop on Approximation Algorithms for Combinatorial Optimization, pp. 229–242, Springer, 2002.
[31]. Gallagher B . “Matching structure and semantics: A survey on graph-based pattern matching,” AAAI FS, vol. 6, pp. 45–53, 2006.
[32]. Giugno R., Shasha D . “Graphgrep: A fast and universal method for querying graphs,” in Pattern Recognition, 2002. Proceedings. 16th International Conference on, vol. 2, pp. 112–115, IEEE, 2002.
[33]. Natarajan M . “Understanding the structure of a drug traffcking organization: a conversational analysis,” Crime Prevention Studies, vol. 11, pp. 273–298, 2000.
[34]. Fan W., Li J., Ma S., Tang N., Wu Y., Wu Y . “Graph pattern matching: From intractable to polynomial time,” Proc. VLDB Endow., vol. 3, pp. 264–275, Sept. 2010.
[35]. Milo R., Shen-Orr S., Itzkovitz S., Kashtan N., Chklovskii D., Alon U . “Network motifs: simple building blocks of complex networks,” Science, vol. 298, no. 5594, pp. 824–827, 2002.
[36]. Kashtan N., Itzkovitz S., Milo R., Alon U . “Mfinder tool guide,” Department of Molecular Cell Biology and Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot Israel, Tech Rep, 2002.
[37]. Wernicke S., Rasche F . “Fanmod: a tool for fast network motif detection,” Bioinformatics, vol. 22, no. 9, pp. 1152–1153, 2006.
[38]. Grochow J. A., Kellis M . “Network motif discovery using subgraph enumeration and symmetry-breaking,” in Annual International Conference on Research in Computational Molecular Biology, pp. 92–106, Springer, 2007.
[39]. Schreiber F., Schwobbermeyer H . “Frequency concepts and pattern detection for the analysis of motifs in networks,” in Transactions on computational systems biology III, pp. 89–104, Springer, 2005.
[40]. Omidi S., Schreiber F., Masoudi-Nejad A . “Moda: an efficient algorithm for network motif discovery in biological networks,” Genes & genetic systems, vol. 84, no. 5, pp. 385–395, 2009.
[41]. Ribeiro P., Silva F . “G-tries: an efficient data structure for discovering network motifs,” in Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 1559–1566, ACM, 2010.

[42]. Gurukar S., Ranu S., Ravindran B . “Commit: A scalable approach to mining communication motifs from dynamic networks,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 475–489, ACM, 2015.

Динамическая компиляция выражений в SQL-запросах для СУБД PostgreSQL

[1] *Е.Ю. Шарыгин <eush@ispras.ru>*
[1] *Р.А. Бучацкий <ruben@ispras.ru>*
[2] *Л.В. Скворцов <leonidxo@gmail.com>*
[1] *Р.А. Жуйков <zhroma@ispras.ru>*
[1] *Д.М. Мельник <dm@ispras.ru>*
[1]*Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1, стр. 52, факультет ВМК

Аннотация. В последние годы по мере увеличения производительности и роста объема оперативной и внешней памяти производительность СУБД для некоторых классов запросов определяется непосредственно скоростью обработки запросов процессором. В СУБД PostgreSQL для исполнения SQL-запросов традиционно используется механизм интерпретации, который приводит к накладным расходам, например, связанным с множественным ветвлением, неявными вызовами функций-обработчиков и выполнением лишних проверок, которых можно избежать, используя информацию, доступную только во время выполнения запроса.

В данной работе рассматривается метод динамической компиляции запросов, в частности, компиляция выражений оператора WHERE и метода последовательного сканирования таблиц SeqScan для СУБД PostgreSQL с помощью компиляторной инфраструктуры LLVM. Рассматривается оптимизация доступа к атрибутам, заключающаяся в вычислении смещений атрибутов кортежа во время компиляции запроса, а также метод автоматической трансляции встроенных функций PostgreSQL во внутреннее представление LLVM IR, что позволяет использовать один и тот же исходный код как для JIT-компилятора, так и для имеющегося интерпретатора. Генерация машинного кода во время выполнения запроса дает возможность избежать накладных расходов традиционной системы интерпретации.

Метод реализован в виде расширения к СУБД PostgreSQL и не требует изменения исходного кода СУБД. Результаты проведенного тестирования показывают, что динамическая компиляция запросов с помощью JIT-компилятора LLVM позволяет получить ускорение в несколько раз на синтетических тестах.

Ключевые слова: динамическая компиляция; JIT-компиляция; базы данных; PostgreSQL; LLVM; языки запросов

DOI: 10.15514/ISPRAS-2016-28(4)-13

Для цитирования: Шарыгин Е.Ю., Бучацкий Р.А., Скворцов Л.В., Жуйков Р.А., Мельник Д.М. Динамическая компиляция выражений в SQL-запросах для СУБД PostgreSQL. Труды ИСП РАН, том 28, вып. 4, 2016 г., стр. 217-240. DOI: 10.15514/ISPRAS-2016-28(4)-13

1. Введение

Среди систем управления базами данных идёт постоянная борьба за производительность. Работы по улучшению производительности большинства реляционных СУБД традиционно были в основном направлены на оптимизацию доступа к памяти ценой менее эффективного использования процессора. Кроме того, реализация в СУБД алгебры реляционных операторов и модели итераторов [1] позволяет упростить как построение и оптимизацию планов, так и реализацию реляционных операторов в отдельности, но в то же время приводит к значительным накладным расходам при выполнении плана.

С ростом объёмов и улучшением операционных характеристик доступа к оперативной памяти накладные расходы, связанные с неэффективным использованием процессора, становятся всё более заметными.

Одно из решений — динамическая компиляция запросов, которая позволяет во время выполнения получить эффективный машинный код, оптимизированный с учётом структуры конкретного запроса, используемых в нём типов данных и функций, и параметров базы данных, таких как размер и схема используемых таблиц, типы индексов и т.д.

В данной работе рассматривается динамическая компиляция выражений оператора WHERE и метода последовательного сканирования SeqScan для СУБД PostgreSQL [2] с помощью компиляторной инфраструктуры LLVM [3].

2. Обзор схожих работ

Увеличение эффективности использования процессора в реляционных СУБД является одним из перспективных направлений исследования в области разработки систем обработки данных.

Динамическая компиляция запросов [4,5] или микро-специализация кода СУБД [6] с использованием информации, доступной во время выполнения, в оптимизированный машинный код позволяет добиться более эффективного использования процессора, сохранив при этом общую архитектуру СУБД и её подсистем, изменив только модуль вычисления запросов. Кроме того, динамическая компиляция открывает новые возможности для оптимизации, связанные с подстановкой констант и вычислением арифметических выражений, традиционно выполняемом при помощи интерпретации.

В [4,5] описывается алгоритм генерации эффективного машинного кода для запросов к реляционной СУБД на языке SQL с использованием компиляторной инфраструктуры LLVM [3]. В работе аргументируется отказ от модели итераторов (Volcano-модели) [1] и приводятся экспериментальные данные,

согласно которым использование динамической компиляции запросов позволяет добиться ускорения в 2 раза, а смена модели выполнения на модель явных вложенных циклов (data-centric) — ещё в 3–4 раза.

Микро-специализация [6] предполагает замену «горячих» участков кода СУБД на версии, оптимизированные под конкретную таблицу, запрос или кортеж. Наличие различных степеней гранулярности позволяет, например, сохранять код, специализированный под таблицы или кортежи, для долгосрочного хранения и использования. В качестве одного из примеров, исследователи показывают, как можно использовать подход микро-специализации для ускорения процедуры обращения к атрибутам в СУБД PostgreSQL.

Для СУБД PostgreSQL разработано коммерческое расширение Vitesse DB [7] с закрытым исходным кодом, в котором реализована динамическая компиляция запросов с использованием LLVM. На запросе Q1 из набора тестов TPC-H [8] компиляция предикатов позволила получить ускорение в 2 раза, а компиляция всего запроса в одну функцию — ускорение в 8 раз. Дальнейшая оптимизация с привлечением параллелизма и колоночного хранилища позволила получить ускорение до 180 раз.

Компиляция арифметических выражений применяется в системах распределённой обработки данных Cloudera Impala [9] и Spark SQL [10].

В Cloudera Impala для компиляции критичных для производительности функций в оптимизированный машинный код используется инфраструктура LLVM. Эксперименты показали увеличение производительности до 5 раз. Особое внимание исследователи уделяют оптимизации косвенных и виртуальных вызовов функций.

В оптимизаторе запросов Spark SQL выполняется трансляция арифметических выражений в код на языке Scala, который затем динамически компилируется в JVM-байткод и запускается. В работе отмечается сокращение количества ветвлений и косвенных вызовов в скомпилированном коде.

Динамическая компиляция и оптимизация с помощью LLVM [3] используются во многих проектах. LLVM используют как при разработке новых языков программирования (Julia [11]), так и при создании более производительных реализаций существующих: JavaScript (JavaScriptCore FTL [12] и LLV8 [13]), Python (Pyston [14]), Ruby (MacRuby [15]) и других.

3. Архитектура обработки запроса в СУБД PostgreSQL

Основной алгоритм выполнения SQL-запроса в PostgreSQL состоит из четырёх этапов:

1. Разбор и анализ запроса.
2. Преобразование.
3. Составление плана.
4. Выполнение плана.

На первом этапе PostgreSQL выполняет лексический и синтаксический анализ SQL запроса и строит дерево разбора. На следующем шаге процедура преобразования принимает от анализатора дерево разбора и выполняет семантический анализ, необходимый для понимания, к каким именно таблицам, функциям и операторам обращается запрос. Структура данных, которая создаётся для представления этой информации, называется деревом запроса.

На третьем этапе PostgreSQL на основе дерева запроса составляет план выполнения запроса. Планировщик работает со структурами данных, называемыми путями, которые представляют собой упрощённые схемы планов, содержащие информацию, необходимую планировщику для принятия решений. После выбора наиболее эффективного пути выполнения строится дерево плана, которое передаётся исполнителю. При подготовке плана PostgreSQL также оценивает время, которое будет затрачено на выполнение того или иного шага выполнения запроса. Итоговый план является наиболее эффективным с точки зрения имеющихся оценок затрат на его выполнение.

Структуру плана выполнения можно вывести с помощью команды EXPLAIN. Например, на рис. 1 показан план выполнения для простого запроса “*select count(*) from rtbl where x+y < 1”;*

```
                QUERY PLAN
---------------------------------------------------
Aggregate
  ->  Seq Scan on rtbl
        Filter: ((x + y) < '1'::double precision)
(3 rows)
```

Рис. 1. Пример плана выполнения запроса

Fig. 1. Example of execution plan

Финальным этапом является выполнение плана, которое реализовано при помощи модели итератора, также известной как Volcano Style Processing [1]. В этой модели запрос состоит из множества операторов, каждый оператор является итератором с интерфейсом “open()”, “next()”, “close()”.

Исполнитель принимает план, созданный планировщиком, и обрабатывает его рекурсивно сверху вниз по дереву, при этом каждый узел в дереве плана вызывает метод “next()” от узлов ниже в дереве плана для получения входных данных, обрабатывает и возвращает один кортеж на узел выше.

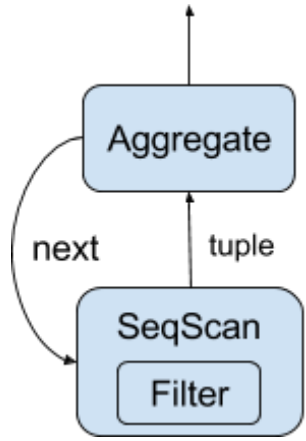


Рис. 2. Модель итераторов для запроса

Fig. 2. Iterator model for example query

В плане выполнения запроса на рис. 2 узел Aggregate будет обращаться за входными данными к дочернему узлу SeqScan, представляющему чтение таблицы. В результате выполнения узла SeqScan исполнитель выберет одну строку из таблицы и вернет её вызывающему узлу Aggregate. Надо заметить, что условие WHERE применено как фильтр (Filter на рис. 2) к узлу плана SeqScan, который проверяет условие для каждой прочтенной им строки и выводит только удовлетворяющие условию строки.

Динамической компиляции данного метода сканирования и фильтрации посвящена основная часть этой работы.

4. Компиляторная инфраструктура LLVM

LLVM [3] — компиляторная инфраструктура для компиляции и оптимизации программ. В LLVM используется низкоуровневое типизированное платформо-независимое промежуточное представление LLVM IR, основанное на SSA-форме, которое, в свою очередь, может быть представлено и использовано одним из трёх способов:

- как граф структур данных, представляющих функции, базовые блоки и инструкции в оперативной памяти — используется для генерации, анализа и оптимизации программ;
- как закодированное бинарное представление, называемое биткодом LLVM, — используется как формат ввода-вывода в различных инструментах, составляющих инфраструктуру LLVM;
- как человекочитаемое текстовое представление — используется для отладки.

Инфраструктура LLVM предоставляет богатый API на языках C++, C, Go, OCaml и Python для анализа и оптимизации программ. Кроме того, в её состав входит широкий набор инструментов, из которых в данной работе используются:

- Clang — компилятор с языков C, C++ и Objective-C во внутреннее представление LLVM;
- opt — статический оптимизатор LLVM-представления;
- llvm-link — компоновщик LLVM-модулей, позволяет скомпоновать несколько модулей в один;
- llc — статический компилятор из внутреннего представления LLVM под различные платформы (поддерживаются x86, x86_64, ARM и многие другие); в состав llc входит библиотека CPPBackend, которая позволяет компилировать в код на языке C++ с использованием LLVM C++ API.

Инфраструктура LLVM содержит модуль для динамической компиляции MCJIT [16], в котором задействованы механизмы LLVM для машинно-зависимой оптимизации и генерации кода под различные платформы. Используя MCJIT и LLVM API, можно динамически компилировать исполняемый код во время выполнения основной программы, что позволяет учитывать при оптимизации больше информации, например, типы переменных (для динамически типизированных языков).

5. Особенности хранения данных и реализации последовательного сканирования в PostgreSQL

Рассмотрение реализации SeqScan в PostgreSQL предварим кратким описанием структуры данных, используемой в PostgreSQL для хранения таблиц, а именно heap-файла [17].

Heap-файл — это файл на диске, содержащий данные таблицы. Одна таблица в PostgreSQL может быть представлена несколькими heap-файлами, особенно если таблица большая: размер heap-файла ограничен (по умолчанию) 1 ГБ.

Heap-файл состоит из последовательности страниц фиксированного размера (по умолчанию 8 КБ). Структура страницы представлена на рис. 3.

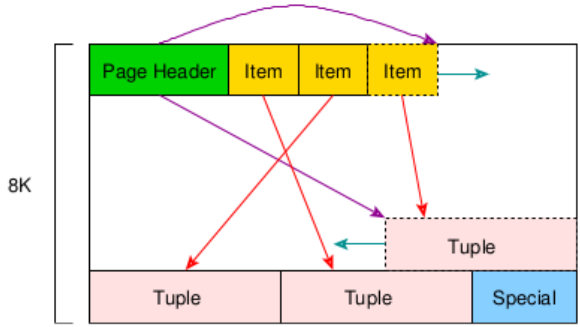


Рис. 3. Структура страницы в heap-файле

Fig. 3. Structure of a heap file page

Данные таблицы представляются набором кортежей (tuple), причём одной строке таблицы может соответствовать больше одного кортежа в силу используемого в PostgreSQL механизма многоверсионности (MVCC).

Кортежи пишутся с конца страницы в начало, а с начала в конец пишутся указатели на кортежи (ItemId), состоящие из смещения на странице и длины кортежа, что позволяет эффективно последовательно обходить все кортежи, присутствующие на странице, — что и составляет основу оператора SeqScan.

Таким образом, оператор SeqScan состоит из двух вложенных циклов:

1. Внешний цикл по всем страницам таблицы. Начиная с версии PostgreSQL 9.6 внешний цикл может выполняться и в параллельном режиме на нескольких процессах.
2. Внутренний цикл по всем ItemId и соответствующим им кортежам на странице. Для каждого кортежа производится
   - проверка соответствия выполняемой транзакции и
   - вычисление предиката условия WHERE.

Следуя Volcano-модели, оператор SeqScan представляется итератором, с каждым вызовом соответствующего которому метода next() производится выполнение не более одной итерации внешнего цикла и не более одной итерации внутреннего и возврат следующего кортежа или индикатора конца потока. При этом каждый следующий вызов продолжает выполнение циклов с позиций, достигнутых предыдущим вызовом, и обновляет значения счётчиков, сохранённых в объекте состояния SeqScanState для того, чтобы выполнение можно было продолжить при следующем вызове.

Для вычисления предиката условия WHERE используется интерпретатор выражений (подробнее в 6.2). Для обеспечения доступа к участвующим в выражении атрибутам кортежа используется функция slot_getattr, которая по

мере необходимости производит частичную десериализацию атрибутов кортежа.

Процесс десериализации предполагает последовательный обход кортежа как байтового массива и восстановление атрибутов одного за другим. В механизме частичной десериализации для каждого следующего атрибута этот обход запускается, начиная с позиции последнего восстановленного атрибута, что позволяет частично избежать восстановления не используемых в запросе атрибутов.

Неэффективность процедуры доступа к атрибутам кортежа является прямым следствием того, насколько компактно таблицы представляются с помощью heap-файлов. Размер кортежа и расположение атрибутов могут варьироваться в зависимости от хранимых данных:

- значения NULL хранятся в битовой маске в заголовке кортежа, а при сериализации атрибутов кортежа пропускаются;
- атрибуты переменной длины (varlena — variable-length attributes; например, строки) могут храниться как в самом кортеже, так и вне его (в т.н. TOAST-таблице);
- атрибутов в кортеже может оказаться меньше, чем колонок в таблице, — в таком случае значения остальных атрибутов считаются равными NULL.

Таким образом, в общем случае смещение атрибута с порядковым номером N определяется наличием NULL-атрибутов и длиной атрибутов переменной длины среди атрибутов с номерами от 1 до N - 1 и потому требует линейного обхода по всем атрибутам с меньшими номерами.

6. Реализация динамической компиляции в PostgreSQL

JIT-компилятор запросов для PostgreSQL, о котором идёт речь в этой статье, реализован в виде расширения к СУБД.

Механизм расширений в PostgreSQL предоставляет весьма широкие возможности: при помощи расширений можно определять новые типы данных, типы индексов (access methods), новые функции и операторы для использования в SQL-запросах, а также перехватывать управление на определённых этапах обработки запроса при помощи регистрации функций-обработчиков.

Во время загрузки расширение регистрирует обработчик выполнения запроса, который вызывается после этапа оптимизации непосредственно перед выполнением плана. В обработчике проверяется, поддерживаются ли все выражения, используемые в запросе, в случае чего производится динамическая компиляция и выполнение кода, оптимизированного под конкретный запрос.

В 6.1 описывается динамическая компиляция метода сканирования SeqScan, а в 6.2 — выражений оператора WHERE.

6.1 Динамическая компиляция метода сканирования SeqScan

SeqScan — это один из методов сканирования таблиц, который выполняет последовательное сканирование таблицы в поисках подходящих под условие кортежей. Он регулярно обращается к фильтру, который вычисляет результат логического выражения, стоящего за оператором WHERE. Последовательное сканирование определено для всех таблиц и является базовым методом доступа к данным в PostgreSQL.

При разработке рассматриваемого в данной работе JIT-компилятора для PostgreSQL оператор SeqScan, реализация которого подробно описана в разд. 5, было решено переписать с использованием LLVM C API. Несмотря на несколько возросшую сложность, такое переписывание позволило:

- пересмотреть используемую вычислительную модель (раздел 6.1.1);
- спроектировать и реализовать ряд оптимизаций, возможных только в динамически компилируемом окружении (6.1.2, 6.1.3);
- динамически компилировать и оптимизировать код вычисления арифметических выражений и предикатов (6.2).

Рассмотрим некоторые самые значимые из применённых оптимизаций.

6.1.1 Отказ от итеративной модели

В используемой в PostgreSQL Volcano-модели [1] каждый оператор реализуется при помощи итератора, метод “next()” которого возвращает следующий кортеж. Реализация метода “next()” нелистового оператора в дереве запроса использует “next()” для получения данных от дочерних операторов. Таким образом, для каждого конкретного запроса операторы в Volcano-модели организуются в конвейер, в котором поток данных управляется корневым оператором запроса, через цепочку вызовов “next()” продвигающим циклы сканирования на следующую итерацию.

Описанная модель обладает следующими достоинствами:

1) позволяет останавливать выполнение дочерних операторов (например, из оператора Limit), что позволяет избежать лишних вычислений;

2) позволяет распределять вычисления на несколько вычислительных узлов [1].

Эти достоинства, однако, никак не проявляются на рассматриваемых в данной работе простых запросах вида:

select <columns> from <table> where <condition>;

В то же время, в недостатки итеративной модели, проявляющиеся даже на таких простых запросах, можно отнести существенные накладные расходы. Так, для оператора SeqScan необходимость сохранения состояния между вызовами next() означает, что для каждого считываемого из таблицы кортежа необходимо

вначале загрузить переменные состояния, в том числе счётчики циклов, из объекта SeqScanState и только потом продолжить выполнение с нужной итерации, при этом записав обновлённые значения переменных для последующих вызовов.

Без использования модели итераторов тот же запрос мог бы быть представлен непосредственно при помощи двух вложенных циклов:

```
for page ← table
    for tuple ← page
        if condition(tuple)
            print(columns(tuple))
```

Код в таком виде позволяет представить счётчики циклов и другие переменные состояния локальными переменными на стеке или регистрах процессора и загружать их только при необходимости (например, при переходе на следующую страницу).

Оператор SeqScan в рассматриваемом в данной статье JIT-компиляторе реализован без использования модели итераторов.

6.1.2 Оптимизация доступа к атрибутам

Можно выделить следующие не реализованные в PostgreSQL возможности для оптимизации последовательного доступа к атрибутам:

1. Заголовок таблицы содержит свойства attnotnull (атрибуты, у которых это свойство не установлено, называются nullable и могут принимать значения NULL) и attlen (длина атрибута, отрицательна для атрибутов переменной длины) для каждого атрибута. Это позволяет вычислить заранее смещения первых нескольких атрибутов, которые имеют фиксированную длину и не могут принимать значение NULL.

2. Несмотря на то, что, начиная с атрибута, следующего за первым nullable или атрибутом переменной длины, смещения атрибутов фиксированными не являются, длину всякой последовательности not-nullable атрибутов фиксированной длины можно вычислить заранее, что позволяет пропускать такие последовательности, состоящие из атрибутов, не используемых в предикате.

3. Зная наперёд, какие атрибуты той или иной таблицы используются в запросе, а какие нет, можно извлекать только используемые атрибуты при первом считывании кортежа и отказаться от затратного механизма ленивой десериалиации.

Рассмотрим пример (рис. 4). Пусть в кортежах таблицы по 12 атрибутов, из которых атрибуты 5, 8 и 11 являются nullable или имеют переменную длину (помечены символом N/V). Пусть в запросе встречаются только атрибуты 3 и 8.

Рис. 4. Оптимизация доступа к атрибутам. Цветом выделены атрибуты, участвующие в запросе.

Fig. 4. Attribute access optimization. Highlighted: attributes involved in the query.

Тогда согласно оптимизации (1) смещение атрибута 3 вычисляется на этапе компиляции, согласно оптимизации (2) длина атрибута 4 и атрибутов 6–7 также вычисляются на этапе компиляции, и согласно оптимизации (3) атрибуты 9–12 не считываются и десериализация происходит не во время вычисления выражения, а во время загрузки кортежа. Таким образом, при обработке кортежа требуется прочитать всего три атрибута: атрибуты 3, 5 и 8.

Для сравнения, используемый в PostgreSQL механизм доступа к атрибутам требует чтения восьми атрибутов 1–8 за два вызова функции slot_getattr (первый вызов считывает атрибуты 1–3, второй — 4–8).

Предложенные оптимизации возможны только при динамической компиляции кода под конкретный запрос или конкретную таблицу и реализованы в рассматриваемом в данной статье JIT-компиляторе.

6.1.3 Подстановка констант и дальнейшая специализация

Применение динамической компиляции позволяет учитывать при оптимизации кода информацию, доступную только во время выполнения, в частности подставлять в код константные параметры, такие как количество страниц, направление обхода и т.д.

Например, на рис. 5 представлен фрагмент исходного кода PostgreSQL, который отвечает за извлечение атрибута из кортежа.

Во время выполнения для каждого конкретного атрибута известны параметры attbyval (тип передачи атрибута) и attlen, что позволяет подставить эти значения непосредственно в код и избежать излишних ветвлений.

Оптимизация подстановки констант реализована через разделение всех переменных, используемых в коде JIT-компилятора, на два класса: переменные времени компиляции и переменные времени выполнения. Оптимизации свёртки и продвижения констант выполняются после генерации кода вместе с прочими оптимизациями LLVM.

Преимуществом такого подхода является то, что при правильной реализации подстановка констант будет применена единообразно для всего запроса и что значения переменных времени компиляции могут использоваться при генерации кода для, например, условной генерации части функций.

Недостатком такого подхода является необходимость разделения всех переменных на два класса вручную, что, во-первых, подвержено ошибкам реализации и, во-вторых, может приводить к неоптимальному результату (когда разработчик по причине отсутствия доказательства противного из соображений корректности отвёл переменную-константу в класс переменных времени выполнения).

```
#define fetch_att(T,attbyval,attlen) \
( \
   (attbyval) ? \
   ( \
      (attlen) == (int) sizeof(Datum) ? \
         *((Datum *)(T)) \
      : \
     ( \
      (attlen) == (int) sizeof(int32) ? \
         Int32GetDatum(*((int32 *)(T))) \
      : \
      ( \
         (attlen) == (int) sizeof(int16) ? \
            Int16GetDatum(*((int16 *)(T))) \
         : \
         ( \
            AssertMacro((attlen) == 1), \
            CharGetDatum(*((char *)(T))) \
         ) \
      ) \
     ) \
   ) \
   : \
   PointerGetDatum((char *) (T)) \
)
```

Рис. 5. Код извлечения атрибута из кортежа (PostgreSQL)

Fig. 5. Source code of attribute extraction routine (PostgreSQL)

Постановка констант вместе с дальнейшими оптимизациями позволяет получить код оператора SeqScan, специализированный под конкретный запрос и конкретные таблицы базы данных.

6.2 Динамическая компиляция выражений оператора WHERE

WHERE — необязательный оператор PostgreSQL, имеющий форму: WHERE <*предикат*>, где предикат — это любое выражение, возвращающее результат логического типа. Кортеж удовлетворяет условию предиката, если для значений атрибутов данного кортежа результат условия является истиной. Кортежи, для которых значение предиката оператора WHERE является ложью или NULL, исключаются из результата запроса.

Для определения доли времени выполнения оператора WHERE, по которому фильтруются кортежи, было проведено профилирование выполнения SQL запросов из тестового набора TPC-H [8], по результатам которого выяснилось, что на некоторых запросах из TPC-H доля времени вычисления предикатов оператора WHERE достигает более 50%.

Для вычисления предиката оператора WHERE PostgreSQL выполняет интерпретацию дерева выражений, где каждое выражение состоит из дерева отдельных операторов и функций, как показано в левой части рис. 6. Каждая вершина дерева вызывает функции соответствующих дочерних вершин неявным образом (через указатель на функцию). Для вычисления самих операций вызываются встроенные функции PostgreSQL, а для доступа к атрибутам используется функция slot_getattr, которая по мере необходимости извлекает атрибуты из кортежа. Это приводит к большим накладным расходам во время выполнения, так как оптимизация встраивания функций (inlining) не может быть выполнена, что позволило бы компилятору делать дальнейшие оптимизации, такие как удаление общих подвыражений (common subexpression elimination, CSE) и т.д.

Поскольку во время выполнения известны вызываемые функции и операции, можно использовать кодогенерацию для замены неявных вызовов функций на явные, которые в дальнейшем могут быть встроены.

Для динамической компиляции выражений, стоящих за оператором WHERE, сперва анализируется дерево выражений с целью проверки на то, реализована ли поддержка всех необходимых выражений и типов значений. В случае, если выражение не поддерживается, запрос выполнится с использованием интерпретации, как обычно в PostgreSQL.

Далее производится рекурсивный обход дерева выражений в обратном порядке и вызов функций-генераторов, реализованных с применением LLVM C API и использующих функции-генераторы встроенных функций PostgreSQL для генерации кода операций на языке LLVM IR. Используемые атрибуты загружаются заранее при считывании кортежа (как описано в разделе 6.1.2).

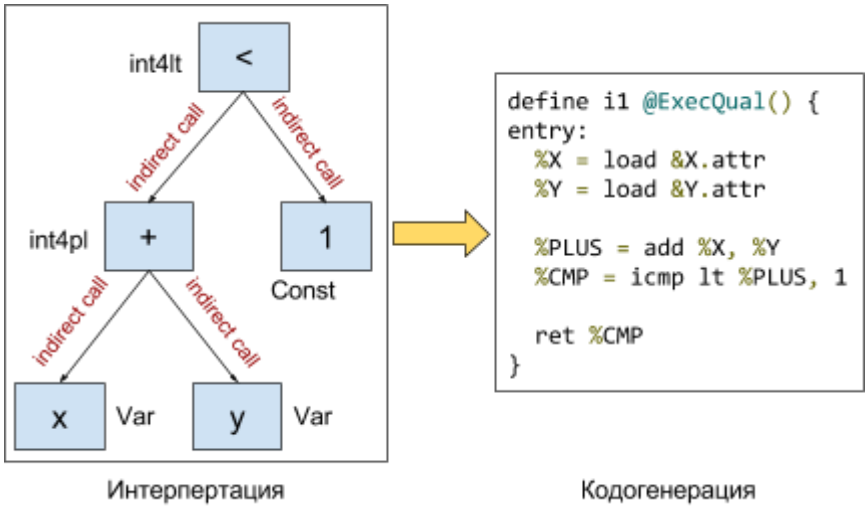


Рис. 6. Слева — интерпретация дерева выражений, справа — сгенерированный LLVM IR

Fig. 6. Left — interpretation of an expression tree, right — generated LLVM IR

В результате этого обхода генерируется код в виде функции на языке LLVM IR (ExecQual на рис. 6), которая будет вызвана из SeqScan для вычисления предикатов.

Как показано на правой части рис. 6, путем замены неявных вызовов функций на вызовы функций-генераторов LLVM IR и дальнейшей генерации кода с использованием оптимизации встраивания функций (inlining), код для дерева выражений становится линейным и может быть динамически скомпилирован и выполнен без каких-либо накладных расходов на неявные вызовы функций.

В данной работе рассматриваются два метода кодогенерации встроенных функций PostgreSQL, используемых при вычислении выражений: реализация вручную и предкомпиляция с использованием компилятора clang.

6.2.1 Реализация встроенных функций вручную

Одним из подходов к реализации функций-генераторов LLVM IR для выражений, используемых в операторе WHERE, является ручное переписывание встроенных функций PostgreSQL с использованием LLVM C API, для дальнейшей генерации кода на языке LLVM IR.

Например, на рис. 7 представлен фрагмент исходного кода PostgreSQL для функции сложения двух целых чисел (int4pl), а на рис. 8 — переписанная с использованием LLVM C API версия той же функции, при вызове которой сгенерируется код на языке LLVM IR.

```
Datum int4pl(PG_FUNCTION_ARGS)
{
  int32   arg1 = PG_GETARG_INT32(0);
  int32   arg2 = PG_GETARG_INT32(1);
  int32   result;

  result = arg1 + arg2;

  /*
   * Overflow check. If the inputs are of different signs then
   * their sum cannot overflow. If the inputs are of the same sign,
   * their sum had better be that sign too.
   */
  if (SAMESIGN(arg1, arg2) && !SAMESIGN(result, arg1))
    ereport(ERROR,
          (errcode(ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE),
           errmsg("integer out of range")));
  PG_RETURN_INT32(result);
}
```

Рис. 7. Исходный код встроенной функции int4pl (PostgreSQL)

Fig. 7. Source code of built-in function int4pl (PostgreSQL)

Реализация конкретных функций может отличаться от оригинальной версии, например, проверками на переполнение (в LLVM есть встроенные функции для быстрой проверки на переполнение арифметических операций: *llvm.sadd.with.overflow.i32* на рис. 8), но и функция, и её переписанная версия, должны возвращать одинаковый результат. Данный метод позволяет учитывать при генерации кода больше информации и тем самым дает возможность, в частности, избавиться или изменить множество проверок, которые накладывают дополнительные расходы во время выполнения.

К недостаткам данного метода можно отнести жёсткую привязанность к реализации функций на LLVM IR. Для поддержки всех встроенных функций PostgreSQL необходимо вручную написать функции-генераторы с помощью LLVM C API, отслеживать изменения в коде PostgreSQL и вносить соответствующие изменения в переписанные версии, что, учитывая суммарное число встроенных функций в PostgreSQL (больше 2000), является трудоёмкой задачей, чреватой возникновением ошибок при переписывании.

LLVM C API

```
LLVMValueRef define_int4pl (LLVMBuilderRef LLVMValueRef left,
LLVMValueRef right)
{
 LLVMValueRef int4pl = LLVMAddFunction("int4pl");
 LLVMBasicBlockRef entry_block, overflow_block, result_block;
 LLVMValueRef sadd_func, args[2] = {left, right}, result, over_bit, call;

 sadd_func = LLVMAddFunction("llvm.sadd.with.overflow.i32");

 LLVMPositionBuilderAtEnd(entry_block);
 call = LLVMBuildCall(sadd_func, args, 2);
 result = LLVMBuildExtractValue(call, 0);
 over_bit = LLVMBuildExtractValue(call, 1);
 over_bit = LLVMBuildIsNotNull(over_bit);
 LLVMBuildCondBr(over_bit, overflow_block, result_block);

 LLVMPositionBuilderAtEnd(overflow_block);
 LLVMBuildCall(OVERFLOW_ERROR_f, NULL, 0);
 LLVMBuildUnreachable();

 LLVMPositionBuilderAtEnd(result_block);
 LLVMBuildRet(result);
 return int4pl;
}
```

LLVM IR

```
define i64 @int4pl (i32 %0, i32 %1) {
entry_block:
 %call = call @llvm.sadd.with.overflow.i32(i32 %0, i32 %1)
 %result = extractvalue {i32, i1} %call, 0
 %over_bit = extractvalue {i32, i1} %call, 1
 %over_bit = icmp eq i1 %over_bit, null
 br i1 %over_bit, label %overflow_block, %result_block

overflow_block:
 %error = call @overflow_error
 unreachable

result_block:
 ret %result
 }
```

Рис. 8. Функция-генератор LLVM IR для функции int4pl

Fig. 8. LLVM IR generator function for int4pl

Аналогично с типами переменных и констант, которые внутри PostgreSQL хранятся в виде 64-битных значений (Datum), что значит, что для каждого типа необходимо написать функцию, конвертирующую 64-битное значение в значение необходимого типа и обратно.

6.2.2 Предварительная компиляция встроенных функций

Недостаток метода ручной реализации всех встроенных функции PostgreSQL привел к необходимости рассмотрения альтернативных способов получения генераторов встроенных функций PostgreSQL на языке LLVM IR.

В состав LLVM (до версии 3.8) входила библиотека CPPBackend, которая переводит (транслирует) биткод LLVM в соответствующий код на языке C++, при выполнении которого вызываются функции из LLVM C++ API для генерации модуля исходного кода LLVM IR.

Используя библиотеку CPPBackend, был реализован метод автоматического получения функций-генераторов LLVM IR встроенных функций PostgreSQL путем предкомпиляции этих функций.

Метод работает следующим образом: множество файлов исходного кода PostgreSQL, содержащие встроенные функции, с помощью компилятора *clang* транслируются в объектные файлы биткода LLVM. Затем, с помощью компоновщика *llvm-link*, полученные файлы компонуются в единый биткод-файл, который оптимизируется модульным оптимизатором *opt*. На основе оптимизированного биткода статический компилятор *llc,* в котором реализован интерфейс библиотеки CPPBackend (-march=cpp), строит C++ файл, содержащий функции-генераторы на LLVM C++ API, вызовы которых сгенерируют код соответствующих встроенных функций исходного кода PostgreSQL на языке LLVM IR. Трансляция биткода в C++ код, содержащий вызовы функций из LLVM API на языке C++, показана на рис. 9. Общая схема метода показана на рис. 10.

Стоит отметить, что генерация биткод-файлов, их компоновка, оптимизация и трансляция в C++ файл и дальнейшая компиляция этого файла происходит один раз во время сборки расширения, что оставляет больше времени на оптимизацию кода, специфичного для конкретного запроса.

Другими преимуществами данного метода являются простота и универсальность реализации, упрощенная поддержка, поскольку отпадает необходимость в ручной реализации каждой встроенной функции и отслеживании изменений в коде PostgreSQL.

LLVM IR

```
define i64 @int4pl(%struct.FunctionCallInfoData* %fcinfo) {
entry:
 %1 = getelementptr %struct.FunctionCallInfoData, %struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
 %2 = load i64, i64* %1
 %3 = trunc i64 %2 to i32
 %4 = getelementptr %struct.FunctionCallInfoData, %struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 1
 %5 = load i64, i64* %4
 %6 = trunc i64 %5 to i32
 %7 = add nsw i32 %6, %3
 %.lobit = lshr i32 %3, 31
 %.lobit1 = lshr i32 %6, 31
 %8 = icmp ne i32 %.lobit, %.lobit1
 %.lobit2 = lshr i32 %7, 31
 %9 = icmp eq i32 %.lobit2, %.lobit
 %or.cond = or i1 %8, %9
 br i1 %or.cond, label %ret, label %overflow

overflow:
 tail call void @ereport(...)

ret:
 %10 = zext i32 %7 to i64
 ret i64 %18
}
```

LLVM C++ API

```
Function* define_int4pl(Module *mod) {

Function* func_int4pl = Function::Create(..., /*Name=*/"int4pl", mod);
// Block (entry)
Instruction* ptr_1 = GetElementPtrInst::Create(NULL, fcinfo, 0, "", entry);
LoadInst* int64_2 = new LoadInst(ptr_1, "", false, entry);
CastInst* int32_3 = new TruncInst(int64_2, IntegerType::get(..., 32), "", entry);
Instruction* ptr_4 = GetElementPtrInst::Create(NULL, fcinfo, 1, "", entry);
LoadInst* int64_5 = new LoadInst(ptr_4, "", false, entry);
CastInst* int32_6 = new TruncInst(int64_5, IntegerType::get(..., 32), "", entry);
BinaryOperator* int32_7 = BinaryOperator::Create(Add, int32_6, int32_3, "", entry);
BinaryOperator* lobit = BinaryOperator::Create(LShr, int32_3, 31, ".lobit", entry);
BinaryOperator* lobit1 = BinaryOperator::Create(LShr, int32_6, 31, ".lobit1", entry);
ICmpInst* int1_8 = new ICmpInst(*entry, ICMP_NE, lobit, lobit1, "");
BinaryOperator* lobit2 = BinaryOperator::Create(LShr, int32_7, 31, ".lobit2", entry);
ICmpInst* int1_9 = new ICmpInst(*entry, ICMP_EQ, lobit2, lobit, "");
BinaryOperator* int1_or_cond = BinaryOperator::Create(Or, int1_8, int1_9, "or.cond", entry);
BranchInst::Create(ret, overflow, int1_or_cond, entry);

// Block (overflow)
CallInst* void_err = CallInst::Create(func_erreport, void, "...", overflow);

// Block (ret)
CastInst* int64_10 = new ZExtInst(int32_7, IntegerType::get(..., 64), "", ret);
ReturnInst::Create(mod->getContext(), int64_10, ret);

return func_int4pl;
}
```

Рис. 9. Трансляция LLVM IR в LLVM C++ API с помощью CPPBackend

Fig. 9. Translation from LLVM IR into LLVM C++ API using CPPBackend library

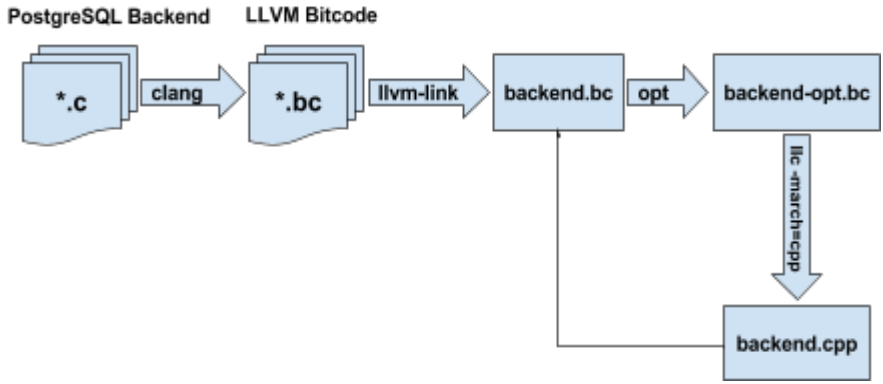


Рис. 10. Схема работы метода предкомпиляции функций PostgreSQL

Fig. 10. Scheme of the PostgreSQL backend function precompilation method

Одним из недостатков данного метода является то, что библиотека CPPBackend не обновлялась с версии LLVM 3.4, а с версии 3.9 не входит в состав LLVM. Реализация метода требует обновления CPPBackend до используемой в данной работе версии LLVM (3.7) и всех последующих версий.

К недостаткам данного метода также можно отнести ухудшение производительности по сравнению с реализацией вручную переписанных встроенных функций PostgreSQL. Ухудшение производительности является следствием того, что при ручном методе генерируется более специализированный под конкретный запрос (выражение) код.

7. Результаты

Для тестирования производительности был использован следующий SQL-запрос:

```
select a0 from widetbl where a199+a198 < 4;
```

Таблица widetbl содержит 201 столбец и 3000000 кортежей. Первый столбец имеет тип text (переменной длины), а остальные 200 имеют тип integer not null. Размер таблицы widetbl составляет 2605 МБ.

Динамическая компиляция оператора SeqScan (раздел 6.1) и предиката оператора WHERE (раздел 6.2) позволяет получить специализированный код под данный запрос, например, при выполнении данного запроса оптимизация доступа к атрибутам (раздел 6.1.2) позволяет пропускать при считывании кортежа атрибуты со 2-го по 199-й.

Тестирование производительности выполнялось на компьютере с четырёхъядерным процессором Intel Core i7 860 с тактовой частотой 2.80 ГГц и с 16 гигабайтами оперативной памяти под управлением 64-битной операционной системы Ubuntu Linux версии 14.04. Время выполнения

измерялось путём многократного выполнения запроса и подсчёта медианы полученных результатов.

Результаты тестирования отражены в табл. 1. Таким образом, время выполнения запроса сократилось в 4,32 раза с использованием метода предкомпиляции встроенных функций (раздел 6.2.2) и в 4,8 раза с использованием метода ручной реализации встроенных функций (раздел 6.2.1) по сравнению с версией PostgreSQL 9.6 Beta 2 с отключенным параллелизмом.

Табл. 1. Сравнение времени выполнения на первом запросе.

Table 1. Comparison of execution times on the first test query.

	PostgreSQL	Предкомпиляция	Ручная реализация
Время (мс)	2623,542	606,674	546,916
Ускорение (раз)	1,00	4,32	4,80

Для тестирования производительности был также использован следующий SQL-запрос:

```
select x, y from rtbl where sqrt((x-256)^2 + (y-128)^2) < 40;
```

Таблица rtbl содержит пять столбцов и 10000020 кортежей. Столбцы x и y имеют тип double precision. Размер таблицы rtbl составляет 1116 МБ.

Результаты тестирования отражены в табл. 2. Время выполнения этого запроса с использованием метода предкомпиляции сократилось в 4,7 раза. Результаты метода ручной реализации на этом запросе отсутствуют по причине отсутствия поддержки всех встроенных функций, используемых в запросе.

Табл. 2. Сравнение времени выполнения на втором запросе.

Table 2. Comparison of execution times on the second test query.

	PostgreSQL	Предкомпиляция	Ручная реализация
Время (мс)	3341,431	711,278	—
Ускорение (раз)	1,00	4,70	—

8. Заключение

В данной работе рассмотрен метод динамической компиляции запросов как одно из средств, позволяющих значительно увеличить производительность СУБД на запросах, скорость обработки которых в первую очередь определяется эффективностью использования процессора.

Метод применён к существующей СУБД PostgreSQL. Рассмотрена компиляция оператора последовательного сканирования SeqScan и выражений оператора WHERE. Метод позволяет совместить производительность, свойственную компилируемым языкам, с развитой инфраструктурой расширений и богатыми возможностями, предоставляемыми PostgreSQL.

Результаты проведенного тестирования показывают, что динамическая компиляция запросов с помощью JIT-компилятора LLVM позволяет получить ускорение в несколько раз на синтетических тестах.

В будущем планируется добавить поддержку нескольких операторов в запросе. Эта задача требует:

1) разработки LLVM-аналогов всех операторов, реализованных в PostgreSQL;

2) замены абстракции итератора (open(), next(), close()) на абстракцию, более подходящую для генерации кода под конкретный запрос и позволяющую реализовывать новые операторы и совмещать несколько операторов в рамках одного запроса.

Изменение модели вычислений в сочетании с применением динамической компиляции позволит получить более эффективный код. Недостатками, свойственными (1), являются:

- трудоёмкость: по существу, (1) требует переписывания части исходного кода PostgreSQL, ответственного за вычисление планов;
- сложность поддержки: альтернативные реализации реляционных операторов требуют постоянной поддержки и обновления до новых версий PostgreSQL.

Ещё одним возможным направлением исследований является автоматическое изменение порядка следования атрибутов в таблице, так чтобы атрибуты, которые могут принимать значение NULL и атрибуты переменной длины шли строго после атрибутов фиксированной длины. Это позволит вычислять во время компиляции запроса смещения всех используемых атрибутов фиксированной длины и получить константное время доступа к ним во время выполнения. Недостатками данного подхода являются:

- ограниченная применимость: одним из свойств оптимизации, описанной в главе 6.1.2, является то, что она не накладывает никаких ограничений на используемые в запросе таблицы базы данных;

- ресурсоемкость преобразования: изменение порядка атрибутов для существующих таблиц требует полной перезаписи heap-файла и пересоздания всех индексов, определённых для данной таблицы;
- недостаточная эффективность: подобная оптимизация порядка следования атрибутов уже входит в типичные сборники советов для DBA и поэтому зачастую выполняется ими вручную.

Описанный в данной статье динамический компилятор запросов находится в стадии подготовки исходного кода для опубликования в открытом доступе (open-source).

Список литературы

[1]. Graefe G. Volcano — an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng.,6(1): 120–135, 1994.
[2]. PostgreSQL, an open source object-relational database system. https://www.postgresql.org
[3]. Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.
[4]. Neumann T. Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. , vol. 4, no. 9, pp. 539–550, Jun. 2011.
[5]. Neumann T., Leis V. Compiling Database Queries into Machine Code. IEEE Data Engineering Bulletin, March 2014.
[6]. Zhang R., Debray S., and Snodgrass R.T. Micro-specialization: dynamic code specialization of database management systems. In Code Generation and Optimization, pages 73, 2012.
[7]. Tan CK. Vitesse DB: 100% Postgres, 100X Faster For Analytics. https://docs.google.com/presentation/d/1R0po7_Wa9fym5U9Y5qHXGlUi77nSda2LlZXPuAxtd-M/pub
[8]. TPC-H, an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/
[9]. Kornacker M., Behm A. и другие. Impala: A Modern, Open-Source SQL Engine for Hadoop. CIDR, 2015.
[10]. Armbrust M., Xin R. S. и другие. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). ACM, New York, NY, USA, 1383-1394, 2015.
[11]. Julia, a high-level, high-performance dynamic programming language for technical computing. http://julialang.org/
[12]. FTL JIT, JavaScriptCore's top-tier optimizing compiler. https://trac.webkit.org/wiki/FTLJIT
[13]. Варданян В.Г., Иванишин В.А., Асрян С.А., Хачатрян А.А, Акопян Дж. А. Динамическая компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM. Труды ИСП РАН, том 27 (выпуск 6), 2015 г., стр. 33-48. DOI: 10.15514/ISPRAS-2015-27(6)-3
[14]. Pyston, an open-source Python implementation using JIT techniques. https://pyston.org
[15]. MacRuby, an implementation of Ruby 1.9 directly on top of Mac OS X core technologies such as the Objective-C runtime and garbage collector, the LLVM compiler infrastructure and the Foundation and ICU frameworks. http://www.macruby.org

[16]. MCJIT Design and Implementation.
http://llvm.org/docs/MCJITDesignAndImplementation.html
[17]. Momjian B. PostgreSQL Internals through Pictures.
http://momjian.us/main/writings/pgsql/internalpics.pdf

Dynamic compilation of expressions in SQL queries for PostgreSQL

[1] *E.Y. Sharygin <eush@ispras.ru>*
[1] *R.A. Buchatskiy <ruben@ispras.ru>*
[2] *L.V. Skvortsov <leonidxo@gmail.com>*
[1] *R.A. Zhuykov <zhroma@ispras.ru>*
[1] *D.M. Melnik <dm@ispras.ru>*
[1]*Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University, CMC Department*
bldg. 52, GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. In recent years, as performance and capacity of main and external memory grow, performance of database management systems (DBMSes) on certain kinds of queries is more determined by raw CPU speed. Currently, PostgreSQL uses the interpreter to execute SQL queries. This yields an overhead caused by indirect calls to handler functions and runtime checks, which could be avoided if the query were compiled into native code "on-the-fly", i.e. just-in-time (JIT) compiled: at run time the specific table structure is known as well as data types and built-in functions used in the query as well as the query itself. This is especially important for complex queries, performance of which is CPU-bound.
We've developed a PostgreSQL extension that implements SQL query JIT compilation using LLVM compiler infrastructure. In this paper we show how to implement JIT compilation to speed up sequential scan operator (SeqScan) as well as expressions in WHERE clauses. We describe some important optimizations that are possible only with dynamic compilation, such as precomputing tuple attributes offsets only for attributes used by the query.
We also discuss the maintainability of our extension, i.e. the automation for translating PostgreSQL backend functions into LLVM IR, using the same source code both for our JIT compiler and the existing interpreter.
Currently, with LLVM JIT we achieve up to 5x speedup on synthetic tests as compared to original PostgreSQL interpreter.

Keywords: dynamic compilation; just-in-time compilation; database management system engines; PostgreSQL; LLVM; query languages.

DOI: 10.15514/ISPRAS-2016-28(4)-13

For citation: Sharygin E.Y., Buchatskiy R.A., Skvortsov L.V., Zhuykov R.A., Melnik D.M. Dynamic compilation of expressions in SQL queries for PostgreSQL. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016. pp. 217-240 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-13

References

[1]. Graefe G. Volcano — an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng.,6(1): 120–135, 1994.
[2]. PostgreSQL, an open source object-relational database system. https://www.postgresql.org
[3]. Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.
[4]. Neumann T. Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. , vol. 4, no. 9, pp. 539–550, Jun. 2011.
[5]. Neumann T., Leis V. Compiling Database Queries into Machine Code. IEEE Data Engineering Bulletin, March 2014.
[6]. Zhang R., Debray S., Snodgrass R.T. Micro-specialization: dynamic code specialization of database management systems. In Code Generation and Optimization, pages 73, 2012.
[7]. Tan CK. Vitesse DB: 100% Postgres, 100X Faster For Analytics. https://docs.google.com/presentation/d/1R0po7_Wa9fym5U9Y5qHXGlUi77nSda2LlZXPuAxtd-M/pub
[8]. TPC-H, an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/
[9]. Kornacker M., Behm A. et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. CIDR, 2015.
[10]. Armbrust M., Xin R.S. et al. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). ACM, New York, NY, USA, 1383-1394, 2015.
[11]. Julia, a high-level, high-performance dynamic programming language for technical computing. http://julialang.org/
[12]. FTL JIT, JavaScriptCore's top-tier optimizing compiler. https://trac.webkit.org/wiki/FTLJIT
[13]. Vardanyan V., Ivanishin V., Asryan S., Khachatryan A., Hakobyan J. Dynamic Compilation of JavaScript Programs to the Statically Typed LLVM Intermediate Representation. Trudy ISP RAN / Proc. ISP RAS], vol. 27, issue 6, 2015. pp. 33-48 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-3
[14]. bbPyston, an open-source Python implementation using JIT techniques. https://pyston.org
[15]. MacRuby, an implementation of Ruby 1.9 directly on top of Mac OS X core technologies such as the Objective-C runtime and garbage collector, the LLVM compiler infrastructure and the Foundation and ICU frameworks. http://www.macruby.org
[16]. MCJIT Design and Implementation. http://llvm.org/docs/MCJITDesignAndImplementation.html
[17]. Momjian B. PostgreSQL Internals through Pictures. http://momjian.us/main/writings/pgsql/internalpics.pdf

Обзор современных методов планирования движения[1]

[1]К.А. Казаков <kazakov@ispras.ru>
[1, 2]В.А. Семенов <sem@ispras.ru>
[1]Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2]Московский физико-технический институт,
141700, Московская область, г. Долгопрудный, Институтский пер., 9

Аннотация. Автоматизация технологически сложных процессов в машиностроении, энергетике, транспорте, медицине, строительстве, а также создание новых продуктов и сервисов невозможны без решения задач планирования движения. В последнее время интерес к ним заметно возрос в связи с развитием средств компьютерного моделирования и становлением таких дисциплин как комплексное планирование индустриальных программ, реалистичная анимация трехмерных сцен, роботизированная хирургия, навигация в динамическом окружении, автоматическая сборка продуктов, организация транспортных потоков в мегаполисах. Данная работа посвящена обзору и сравнительному анализу современных математических методов планирования движения.

Ключевые слова: планирование движения, поиск пути, маршрутные сети, определение столкновений.

DOI: 10.15514/ISPRAS-2016-28(4)-14

Для цитирования: Казаков К.А., Семенов В.А. Обзор современных методов планирования движения. Труды ИСП РАН, 2016 г., стр. 241-294. DOI: 10.15514/ISPRAS-2016-28(4)-14

1. Введение

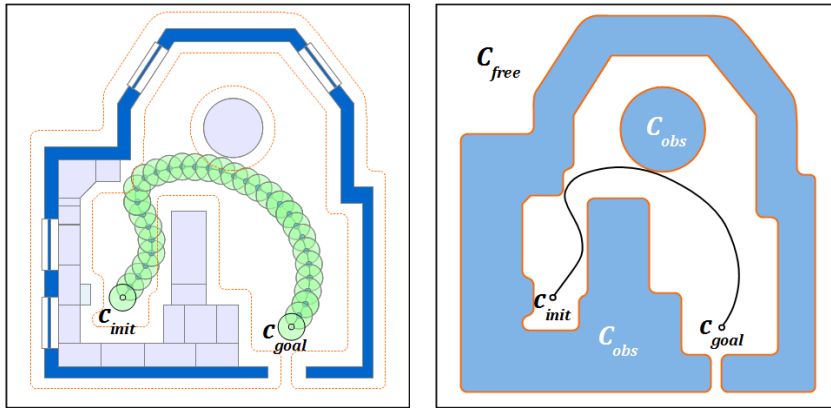
Автоматизация технологически сложных процессов в машиностроении, энергетике, транспорте, медицине, строительстве, а также создание новых продуктов и сервисов невозможны без решения задач планирования движения. В последнее время интерес к ним заметно возрос в связи с развитием средств

[1] Работа поддержана РФФИ (грант 16-07-00606)

компьютерного моделирования и становлением таких дисциплин как комплексное планирование индустриальных программ, реалистичная анимация трехмерных сцен, роботизированная хирургия, навигация в динамическом окружении, автоматическая сборка продуктов, организация транспортных потоков в мегаполисах. Данная работа посвящена обзору и сравнительному анализу современных математических моделей, методов и программных средств планирования движения [1].

Обычно под планированием движения понимается поиск бесконфликтного пути для перемещения твердого тела или кинематической конструкции в пространственно-трехмерной сцене. Искомый путь представляет собой непрерывную кривую в конфигурационном пространстве объекта, которая соединяет его начальное и конечное положения, исключает столкновения с препятствиями сцены и удовлетворяет всем установленным кинематическим и динамическим ограничениям.

Пусть задана сцена как непустое множество препятствий $O \subset W$ в области эвклидова пространства $W \subset E^N$, $N \in \{2,3\}$. Пусть также задано твердое тело $A \subset W$ либо кинематическая цепь $A\langle B,J\rangle$, где $B = \{B_1, B_2, \dots, B_n\} \subset W$ – множество твердотельных звеньев, а $J = \{J_1, J_2, \dots, J_k\}$ – множество кинематических ограничений таких, что при корректной конфигурации цепи предикаты ограничений $J_1(c), J_2(c), \dots, J_k(c)$ принимают истинное значение. Под конфигурацией $c \in C_A$ здесь понимается набор значений параметров, однозначно определяющий положение точек объекта A в пространстве сцены. Обычно используется минимальный набор параметров, соответствующий количеству степеней свободы объекта и определяющий пространство состояний или конфигурационное пространство объекта C_A.



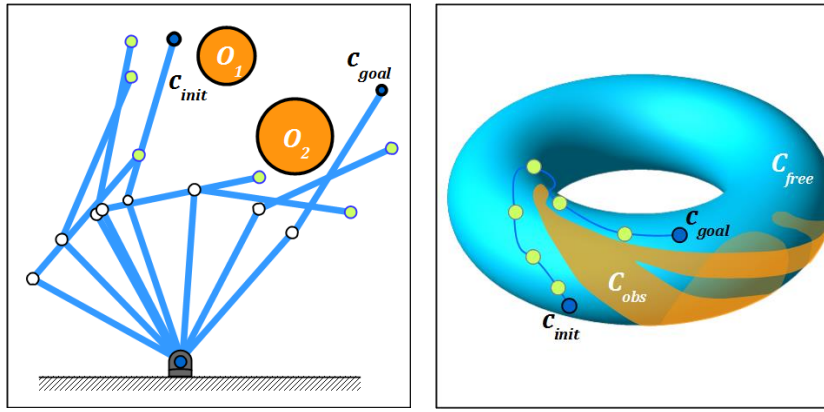
(a) (b)

Рис. 1. Конфигурационное пространства двумерного твердого тела

Fig. 1. Configuration space of two dimensional solid body

Определение. Пространством допустимых состояний назовем множество всех конфигураций объекта $c \in C_A$, удовлетворяющих кинематическим ограничениям и исключающих столкновения с препятствиями сцены $C_{free} = \{c \in C_A \mid J_1(c) \wedge J_2(c), \ldots, \wedge J_k(c) \wedge B_1(c) \cap O = \emptyset, \ldots, \wedge B_n(c) \cap O = \emptyset\}$. Для простого твердого тела свободное множество определяется как $C_{free} = \{c \in C_A \mid A(c) \cap O = \emptyset\}$.

Тогда постановка задачи поиска пути может быть сформулирована следующим образом. Для пары заданных бесконфликтных конфигураций $c_{init}, c_{goal} \in C_{free}$ требуется найти непрерывный путь $p(\tau)\colon [0,1] \to C_{free}$ такой, что $p(0) = c_{init}$ и $p(1) = c_{goal}$.



(a) *(b)*

Рис. 2. Конфигурационное пространства двухзвенного манипуляционного робота

Fig. 2. Configuration space of 2-DOF manipulation robot

Поскольку планирование маршрута, как правило, допускает бесконечное множество решений (хотя может не существовать ни одного решения), иногда данную задачу формулируют в постановке оптимизационной задачи с целевой функцией, соответствующей минимальной длине маршрута или максимальной удаленности перемещаемого объекта от препятствий [2,3].

На практике поиск пути даже в простых сценах с относительно небольшим количеством препятствий становится трудноразрешимой задачей, если перемещаемый объект имеет сложную геометрию или высокое число степеней свободы. В современных индустриальных приложениях часто требуется моделировать поведение сложных кинематических систем с шестью и более степенями свободы в статическом или динамическом окружении, насчитывающим тысячи препятствий.

Функционал существующих прикладных программных пакетов, таких как Kineo CAM (Kineo Computer Aided Motion) [4], ROS (Robot Operating System)

[5], OMPL (Open Motion Planning Library) [6], OpenRAVE (The Open Robotics Automation Virtual Environment) [7] и CuikSuite [8] главным образом обеспечивает решение задач моделирования неголономных механических систем в режиме реального времени и локального планирования движения. Математический арсенал упомянутых пакетов в основном ограничен сэмплинг-методами, которые эффективны в приложениях, связанных, в частности, с управлением движением промышленных роботов при сборке компонентов, программированием координатно-измерительных машин при контроле точности производимых изделий. Однако данные методы демонстрируют несостоятельность в тех случаях, когда требуется определение протяженных бесконфликтных траекторий в трехмерном окружении со сложной топологией. Подобные задачи возникают, например, при пространственно-временной верификации календарно-сетевых графиков архитектурно-строительных проектов и нуждаются в более развитом математическом аппарате [9–11]. При этом архитектура программных пакетов и особенности организации интерфейсов препятствуют реализации в их составе новых глобальных методов планирования движения и интеграции в индустриальные системы, ориентированные на работу с масштабными сценами, состоящими из сотен тысяч и миллионов объектов с индивидуальными геометрическими и динамическими характеристиками.

Таким образом, разработка эффективных математических методов и программных средств планирования движения представляет собой актуальную научную проблему. Настоящая статья посвящена систематическому обзору и сравнительному анализу современных математических методов планирования движения. Важное внимание уделяется ключевым подходам, основанным на пространственной декомпозиции, маршрутных сетях и физических аналогиях с потенциальными полями. Предполагается, что обзор поможет в выработке общих рекомендаций по использованию методов, а также в их выборе при решении практических классов задач. Ожидается также, что обзор послужит конструктивной основой для концептуализации теории планирования движения и создания единой программно-инструментальной среды разработки приложений.

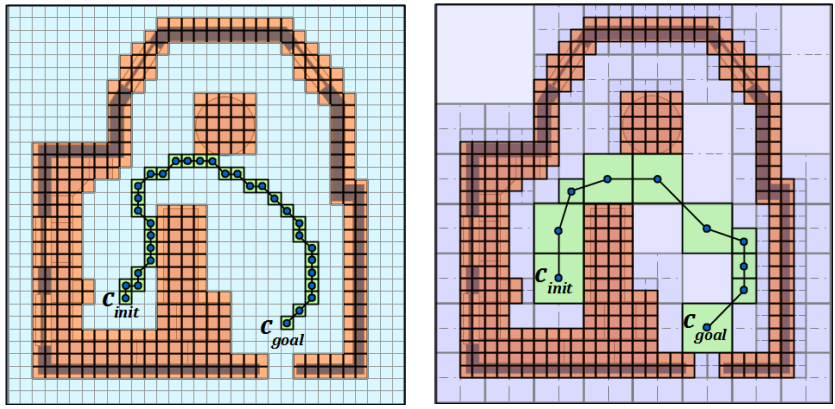
2. Методы на основе пространственной декомпозиции

Наиболее простой и в то же время распространенный подход к планированию движения состоит в применении методов пространственной декомпозиции. Он предполагает разбиение свободных областей сцены на множество простых регионов с использованием тех или иных методов декомпозиции, определение смежности регионов и формирование графа связности, который в дальнейшем может использоваться для навигации в сцене. Фактически, подход реализует схему редукции вычислительно сложной задачи планирования движения в эвклидовом пространстве к типовой задаче поиска пути в графе.

2.1 Регулярная декомпозиция

Одним из способов реализации данного подхода является применение методов регулярной пространственной декомпозиции. Данные методы предполагают разбиение всего объема сцены сеткой с фиксированным размером ячеек, ориентированных по осям координат (рис. 3, a). Для всех ячеек определяется статус занятости. Ячейки могут быть целиком заполненными объектами сцены, частично заполненными или свободными. Маршрут строится путем анализа смежных свободных ячеек, выбора направлений для распространения и проверки принадлежности перемещаемого объекта свободным ячейкам сетки. Таким образом, вместо исходной точной геометрической модели сцены используется ее упрощенное дискретное представление, что существенно упрощает процесс маршрутизации. Вопросы использования методов регулярной декомпозиции применительно к планированию движения широко освещены в литературе [12,13].

Метод относительно прост в реализации, однако требует значительных вычислительных ресурсов из-за необходимости детальной дискретизации всего пространства сцены и обеспечения приемлемой точности маршрута.



(a) (b)

Рис. 3. Пример построения декомпозиции двумерной сцены с помощью (a) равномерной сетки и (b) дерева квадрантов

Fig. 3. An example of decomposition of 2D scene using (a) regular grid and (b) quad tree

Указанного недостатка лишены методы регулярной адаптивной декомпозиции пространства, одним из известных представителей которых является метод на основе октодеревьев занятости [10,14,15]. Благодаря рекурсивной процедуре декомпозиции такие структуры оказываются более экономичными (рис. 3, b). Если принимать во внимание возможную неравномерность распределения объектов по объему сцены и существенную вариацию их размеров, октарные структуры оказываются более рациональными при дискретизации сцены и

исполнении типовых запросов, связанных с пространственной локализацией объектов, поиском соседей, определением столкновений. Однако процедура маршрутизации становится более сложной, а с учетом несбалансированности октодеревьев может приводить и к неестественным траекториям, содержащим большое число изломов. Для преодоления этой проблемы был предложен гибридный метод, использующий окаймление на границах ячеек [16]. Однако он вряд ли может рассматриваться в качестве универсального, поскольку неизбежно порождает избыточное представление сцены и требует существенных дополнительных расходов.

2.2 Объектно-зависимая декомпозиция

Альтернативу методам пространственной декомпозиции составляют методы объектно-зависимой декомпозиции пространства. За счет выделения свободных областей непосредственно по границам объектов сцены устраняются проблемы, обусловленные погрешностью дискретизации сцены. К сожалению, методы этого семейства обладают высокой вычислительной сложностью и в большинстве случаев не могут конкурировать с методами пространственной декомпозиции. Наиболее известными методами объектно-зависимой декомпозиции являются метод вертикального разбиения или трапецеидальной декомпозиции (Trapezoidal Decomposition) [17], триангулированное разбиение [18], цилиндрическая декомпозиция (Cylindrical decomposition) [19], а также декомпозиция Морса [1,20].

3. Методы на основе маршрутных сетей

Важное семейство методов глобального планирования движения составляют методы на основе маршрутных сетей, которые обычно строятся в конфигурационном пространстве перемещаемого объекта или в пространстве сцены. Подобные сети объединяют в себе бесконфликтные переходы или участки путей и расширяются по мере совершения новых успешных попыток перемещения. Иногда сети накапливают также информацию о предпринятых неудачных попытках и связанных с ними вычислительных затратах, что позволяет в дальнейшем выбирать более перспективные направления и ускорить процесс маршрутизации при решении как одиночных, так и множественных задач планирования движения.

Формально маршрутная сеть представляет собой топологический граф $G(V, E)$, вершины V которого соответствуют бесконфликтным конфигурациям перемещаемого объекта $S \subset C_{free}$, а ребра E – бесконфликтным переходам между ними $P(\tau)$: $[0,1] \to C_{free}$, $P(0) \subset S, P(1) \subset S$. Обычно маршрутная сеть строится в предположении выполнения условий достижимости и связности.

Определение. Маршрутная сеть достижима для конфигурационного пространства объекта, если для любой его точки $c \in \mathrm{C}_{free}$ существует вершина

маршрутной сети $s \in S$ и бесконфликтный путь к ней $p(\tau)\colon [0,1] \to C_{free}$, $p(0) = \mathrm{c}$ и $p(1) = s$.

Определение. Маршрутная сеть объекта связна для конфигурационного пространства объекта, если для любой его пары точек $c_{init}, c_{goal} \in \mathrm{C}_{free}$, между которыми существует бесконфликтный путь и найдены сопряженные вершины маршрутной сети $s_{init}, s_{goal} \in S$, также существует маршрут $p'(\tau)\colon [0,1] \to S$, где $p'(0) = c_{init}$ и $p'(1) = c_{goal}$.

Данные условия обеспечивают гарантированное нахождение бесконфликтного маршрута между любыми двумя положениями объекта при условии, что он существует. Первое условие позволяет соединить любую пару заданных точек свободного пространства с маршрутной сетью. Второе условие обеспечивает навигацию по маршрутной сети. К сожалению, данные условия носят декларативный характер и не могут быть конструктивно применены при разработке и реализации методов планирования движения.

Однако сама идея маршрутной сети достаточно плодотворна и нашла воплощение в ряде методов, прежде всего в методах PRM и RRT, подробно обсуждаемых в следующих разделах. Маршрутные сети, развернутые непосредственно в пространстве сцены, предоставляют дополнительные возможности для решения задач планирования движения с разными объектами. Однако в этом случае переходы, бесконфликтные для одного перемещаемого объекта, могут оказаться конфликтными для других объектов. Поэтому построенные маршруты нуждаются в дополнительной верификации, а сама сеть – в достоверных методах анализа переходов для маршрутизации новых объектов.

3.1 Графы видимости

Один из известных методов организации маршрутных сетей основан на применении графов видимости (Visibility Graphs) [18]. В простейших случаях граф видимости строится на множестве вершин полигонов или полиэдров, являющихся геометрическими моделями препятствий сцены, и дополняется начальными и конечными точками маршрутов. Все вершины попарно соединяются линейными отрезками, которые принимаются в качестве ребер графа при условии попарной “видимости” инцидентных им вершин и отсутствия пересечения с препятствиями сцены (рис. 4, а). Сложность построения графа видимости для сцены, представленной полигонами с общим числом вершин n, составляет $O(n^2 log\ n)$ [1], что является довольно затратным для применения в индустриальных приложениях.

Кроме того, преобразование топологического графа в маршрутную сеть также представляет собой сложную задачу, поскольку ребра графа видимости проходят точно через вершины препятствий, и для успешной навигации требуется коррекция путей или предварительное расширение границ препятствий. Избыточное количество порождаемых маршрутов, крайне

нежелательное при моделировании сложных сцен, также является недостатком данного метода. Тем не менее, в двумерном окружении с относительно небольшим числом препятствий данный метод может успешно применяться, например, для решения задач поиска кратчайшего пути в сцене. Более того, существуют попытки использования графов видимости для планирования движения в динамическом окружении [21,22].

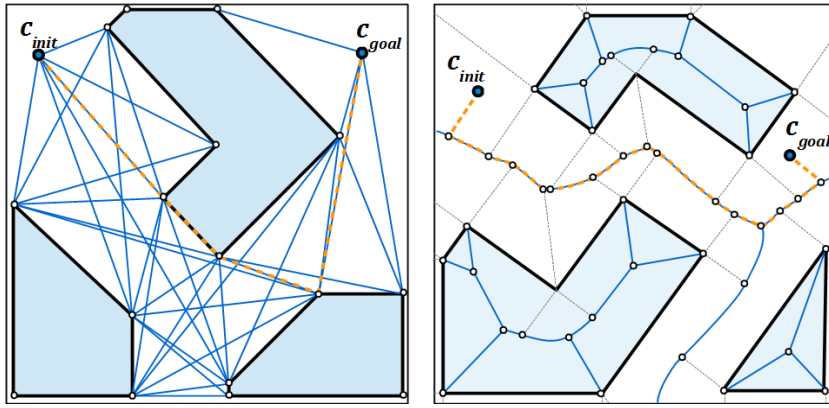


Рис. 4. Пример построения (a) графа видимости и (b) диаграммы Вороного в двумерной полигональной сцене

Fig. 4. An example of (a) visibility graph and (b) Voronoi diagram in two dimensional polygon scene

3.2 Диаграммы Вороного

Альтернативный метод определения маршрутов основан на использовании обобщенных диаграмм Вороного (Generalized Voronoi Diagrams) [1]. Наиболее привлекательное свойство диаграммы Вороного заключается в том, что она является деформационным ретрактом свободного пространства. Тем самым, любой бесконфликтный путь может быть непрерывно отображен в диаграмму Вороного. Более того, метод обеспечивает наибольшее удаление от границ препятствий при условии, что движение осуществляется вдоль ребер диаграммы. Данное свойство позволяет определять наиболее “безопасные” маршруты перемещения в сцене (рис. 4, b).

Наивная реализация данного метода обладает высокой вычислительной сложностью $O(n)^4$, где n – общее количество вершин и граней препятствий [23]. Известны эффективные алгоритмы с лучшими асимптотическими оценками сложности, однако они неустойчивы к вырожденным случаям и трудны для надежной реализации.

Диаграмма Вороного определяется для произвольного метрического пространства. Несмотря на это, ее применение для навигации в эвклидовых пространствах размерности выше двух является затруднительным ввиду того, что маршрутная сеть уже не представляет топологически одномерную структуру. В этих случаях обычно применяются методы на основе обобщенных диаграмм Вороного [24].

4. Дискретный поиск

Методы на основе пространственной декомпозиции и маршрутных сетей позволяют редуцировать вычислительно трудные задачи навигации в сложном многомерном окружении к более простым задачам теории графов, для решения которых имеется развитый математический аппарат. Тем не менее, задачи планирования движения привносят свои особенности в постановку и решение подобных задач [25–27].

Например, маршрутные сети не только представляют топологию свободных областей сцены, но и часто накапливают дополнительную информацию для выбора более достоверных и оптимальных маршрутов навигации. Такими данными могут быть, например, протяженность переходов в маршрутной сети, расстояние до ближайших препятствий сцены, информация о предпринятых удачных и неудачных попытках расширения сети и затраченных для этого вычислительных ресурсах. Подобные данные позволяют использовать эвристические оценки рисков и перспективности маршрутов и, тем самым, ускорить решение одиночных и множественных задач маршрутизации.

4.1 Неинформированный поиск

Классические алгоритмы теории графов реализуют поиск в ширину, поиск в глубину, поиск по критерию стоимости [25]. Данные алгоритмы предполагают полный обход вершин графа без какой-либо информации о том, насколько каждый шаг приближает процесс поиска к решению. Применение данных алгоритмов сопряжено со значительными вычислительными затратами, и поэтому они редко применяются при решении задач планирования движения.

4.2 Эвристический поиск

Эвристические алгоритмы поиска пути предназначены для быстрого отыскания маршрута в графе путем распространения в направлении более перспективных вершин. Для этого обычно организуется очередь из непосещенных вершин, порядок в которой определяется на основе эвристических правил приоритизации.

Одним из наиболее популярных алгоритмов поиска пути данного семейства является A* [28]. Приоритет вершины в данном алгоритме определяется суммой: $f(v) = g(v) + h(v)$, где $g(v)$ – стоимость пути из начальной вершины в данную, а $h(v)$ – эвристическая оценка стоимости пути до целевой вершины.

При этом значение $h(v)$ должно быть неотрицательным и удовлетворять неравенству треугольника. Таким образом, A* является обобщением алгоритма Дейкстры при $f(v) = g(v)$ и алгоритма жадного поиска при $f(v) = h(v)$.

Использование эвристики, как правило, позволяет значительно уменьшить количество посещаемых вершин и, как следствие, повысить производительность поиска. Кроме того, алгоритм находит оптимальные решения при условии, что эвристическая оценка является оптимистической, т.е. значение функции $h(v)$ всегда меньше либо равно точной стоимости пути до целевой вершины.

В наихудшем случае выполнение алгоритма A* может быть сопряжено с экспоненциальным ростом числа непосещенных вершин в очереди, что при большом коэффициенте ветвления дерева поиска порождает проблему чрезмерного потребления памяти. В таких случаях более предпочтительным является использование алгоритма A* с итеративным углублением (Iterative Deeping A*) [29] или алгоритма A* с ограничением памяти (Memory Bounded A*) [30].

4.3 Поиск в регулярной сетке

Методы пространственной декомпозиции на основе регулярных равномерных сеток приводят к дискретному представлению сцены в виде графов с большим количеством вершин и неопределенностью в их приоритизации из-за равной стоимости переходов. Одним из способов повысить эффективность алгоритмов в таких случаях является привлечение идей иерархического поиска. Алгоритмы Hierarchical A* [31] и HPA* [32,33] реализуют данные идеи в результате построения многоуровневого иерархического представления сцены и группировки смежных ячеек сетки. Данное представление затем применяется для построения путей с необходимой степенью детализации на каждом уровне иерархии.

Альтернативный алгоритм Jump Point Search, не требующий дополнительных расходов памяти, предложен в работе [34]. Он использует набор простых эвристических правил для выбора перспективных направлений, в которых могут быть предприняты длинные прямолинейные и диагональные переходы (“прыжки”). В результате из анализа исключается значительное число вершин, не влияющих на стоимость переходов и качество конечного решения, а эффективность поиска пути существенно повышается.

4.4 Алгоритм Lifelong Planning A*

Маршрутные сети, построенные в пространстве сцены для некоторых типовых объектов, могут содержать переходы, непреодолимые для объектов нового типа. В подобных случаях следует попытаться проложить новый маршрут, исключающий конфликтные переходы. Наивный алгоритм мог бы состоять в повторении процедуры поиска в измененном графе с самого начала. Однако с

вычислительной точки зрения такой способ не является рациональным, поскольку не учитывает предшествующие результаты.

Алгоритм LPA* [35] является развитием A* для инкрементального поиска кратчайшего пути в условиях динамического изменения весов ребер. Для каждой вершины алгоритм предусматривает хранение ее фактической стоимости $g(v)$ и предварительной оценки стоимости в соответствии со следующей формулой:

$$rhs(v) = \begin{cases} 0 & , \ v = v_{initial} \\ min_{v' \in pred(v)}\big(g(v') + c(v', v)\big), & v \neq v_{initial} \end{cases},$$

где $pred(v)$ – множество ранее посещенных смежных вершин, а $c(v', v)$ – стоимость перехода из вершины v' в вершину v. Для непосещенных вершин $rhs(v) = g(v) = \infty$.

Приоритетная очередь LPA* содержит множество локально неустойчивых вершин, в которых оценка расходится с фактической стоимостью $g(v) \neq rhs(v)$. Приоритет определяется составным ключом $k(v) = [k_1(v), k_2(v)]$, где $k_1(v) = min\big(g(v), rhs(v)\big) + h(v)$, $k_2(v) = min\big(g(v), rhs(v)\big)$, а $h(v)$ – эвристическая оценка стоимости пути до целевой вершины.

На каждом шаге построения пути из очереди выбирается вершина с минимальным ключом. Если вершина является недооцененной $g(v) < rhs(v)$ и значит изменения могли повлиять на ее стоимость, то она помечается как непосещенная $g(v) = \infty$. В противном случае, если вершина является переоцененной $g(v) > rhs(v)$, то она переводится в локально устойчивое состояние $g(v) = rhs(v)$. Далее производится переоценка смежных вершин. При любом изменении веса одного из ребер графа переоцениваются инцидентные ему вершины, а локально неустойчивые вершины добавляются к приоритетной очереди. Поиск останавливается, если выполняется условие локальной устойчивости целевой вершины или условие, при котором значение ключа целевой вершины оказалось меньше минимального значения в приоритетной очереди.

4.5 Алгоритм D*

Часто в приложениях планирования движения возникает необходимость поиска пути в сценах с неполной, перманентно обновляемой информацией. Примером может служить задача навигации мобильного робота, получающего информацию об окружении с датчиков в режиме реального времени. При установленных изменениях в окружении робота следует перепланировать его маршрут. Для этих целей более подходят такие алгоритмы, как Focussed D* [36] и D*-light [37]. Последний использует принципы, лежащие в основе LPA* и, как правило, демонстрирует лучшую производительность. В работе [38] отмечается простота его реализации.

5. Метод потенциальных полей

Важное семейство методов планирования движения составляют методы потенциальных полей, первоначально разработанные для навигации мобильных роботов и обхода локальных препятствий в режиме реального времени [39]. Методы основаны на физической аналогии с движением заряженной частицы в электростатическом поле. Препятствия сцены генерируют отталкивающие силы, а целевая точка маршрута – значительную притягивающую силу. Направление и скорость движения тела определяются градиентом потенциального поля (рис. 5).

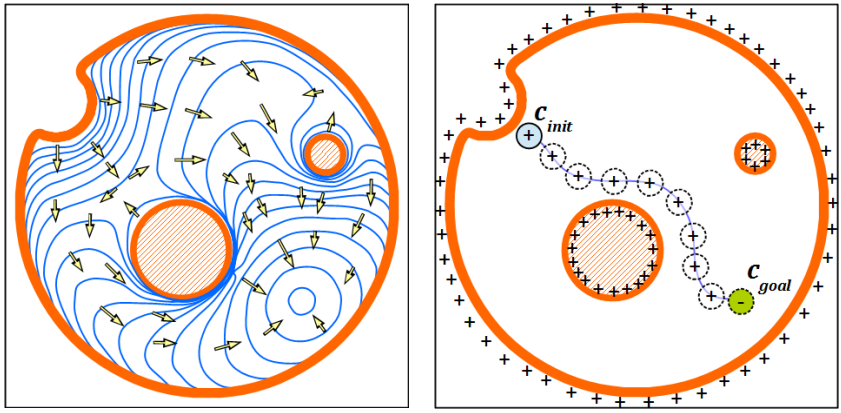


Рис. 5. Движение заряженной частицы в электростатическом поле

Fig. 5. The movement of charged particle in electrostatic field

Навигационная функция, как правило, задается суммой притягивающих и отталкивающих потенциалов: $\mathrm{U(c)} = \mathrm{U_{att}(c)} + \mathrm{U_{rep}(c)}$. При продвижении робота к целевой точке c_{goal} функция притягивающего потенциала $\mathrm{U_{att}}$ должна монотонно возрастать, однако ее конкретный вид может существенно варьироваться. Например, в работе [1] предлагается следующая формула:

$$U_{att}(\mathrm{c}) = \begin{cases} \frac{1}{2}\zeta d^2(\mathrm{c}, \mathrm{c}_{goal}), & d(\mathrm{c}, \mathrm{c}_{goal}) \le D^* \\ D^*\zeta d(\mathrm{c}, \mathrm{c}_{goal}) - \frac{1}{2}\zeta (D^*)^2, & d(\mathrm{c}, \mathrm{c}_{goal}) > D^* \end{cases},$$

где d – функция, определяющая расстояние между точками, ζ – масштабирующий коэффициент. D^* является константным параметром, который определяет пороговое значение расстояния до цели, при достижении которого скорость движения объекта начинает снижаться. Таким способом удается избежать проблемы, связанной с “дрожанием” объекта вблизи целевой точки пути при недостаточно малом временном шаге.

Функция отталкивающего потенциала, формируемого множеством препятствий $O = \{o_1, o_2, \dots o_N\}$, задается следующим образом:

$$U_{rep}(c) = \sum_{i=1}^{N} U_{rep_i}(c, o_i),$$

$$U_{rep_i}(c, o_i) = \begin{cases} \frac{1}{2} n(\frac{1}{d_{min}(c,o_i)} - \frac{1}{D'})^2, & d_{min}(c, o_i) \le D' \\ 0, & d_{min}(c, o_i) > D' \end{cases},$$

где минимальное расстояние от заданной точки до препятствия задается функцией $d_{min}(\mathrm{c}, o) = min_{p \in o} \|c - p\|$, а константный параметр D' определяет предельное расстояние от заданной точки до удаленных препятствий сцены, вклад от которых учитывается при подсчете потенциала.

5.1 Метод рандомизированных потенциальных полей

Основным недостатком методов потенциальных полей, использующих градиентный спуск, является проблема локальных минимумов. Один из способов ее преодоления заключается в попытке “вытолкнуть” перемещаемый объект в случайном направлении при попадании в потенциальную яму. Данную идею реализует известный метод рандомизированных потенциальных полей (Randomized Potential Fields) [40,41]. Она же получила развитие в семействе сэмплинг-методов, подробно обсуждаемых в следующих разделах.

6. Сэмплинг методы

Применение методов планирования движения, основанных на точной или приближенной реконструкции пространства допустимых конфигураций, зачастую невозможно из-за высокой вычислительной сложности, которая становится критичной для сцен с большим количеством препятствий и перемещаемых объектов со значительным количеством степеней свободы. Избежать данной проблемы позволяют популярные на сегодняшний день сэмплинг-методы планирования движения, часто ассоциируемые с методами квази-Монте Карло.

Они предусматривают исследование областей конфигурационного пространства путем многократных испытаний, результатом которых является вердикт о допустимости отдельно выбранных конфигураций. Это обычно осуществляется путем генерации случайных конфигураций и проверки для них кинематических ограничений и столкновений с препятствиями сцены. На множестве полученных в результате сэмплирования бесконфликтных конфигураций строится маршрутная сеть. Важным достоинством методов данного семейства является независимость от геометрического представления и размерности моделируемого окружения.

Недетерминированный характер построения маршрута привносит ряд особенностей, связанных с невозможностью обеспечения полноты алгоритмов и оптимальности найденных решений. Вместе с тем, эвристические стратегии сэмплирования обеспечивают перманентный рост вероятности нахождения бесконфликтного пути с увеличением числа итераций, естественно, если

решение существует (в противном случае алгоритм может выполняться бесконечно долго). В связи с этим вводится понятие вероятностной успешности алгоритма.

Определение. Алгоритм называется вероятностно успешным, если для любой задачи поиска пути $\langle C_{free}, c_{init,} c_{goal} \rangle$, имеющей решение, имеет место $lim_{N\to\infty} P_{success} = 1$, где $P_{success}$ – вероятность найти бесконфликтный путь $p(\tau)$: $[0,1] \to C_{free}$, $p(0) = c_{init}$, $p(1) = c_{goal}$ за N итераций.

Современные алгоритмы, основанные на сэмплировании пространства, как правило, справляются с решением типовых задач поиска пути за приемлемое время. Кроме того, их применение позволяет во многих случаях компенсировать нежелательные эффекты, связанные с неизбежными погрешностями представления геометрических моделей.

Методы сэмплирования представлены двумя основными группами, а именно: методами вероятностных маршрутных сетей [42] и методами на основе деревьев поиска [43–45]. Методы успешно применяются на практике для планирования движения в пространствах высокой размерности. В частности, они широко используются для моделирования движения твердых и деформируемых тел [46,47], а также для анализа неголономных систем со сложными кинематическими связями [48,49] и дифференциальными ограничениями [50].

6.1 Выборка конфигураций

Вообще говоря, множество допустимых состояний объекта в непрерывном конфигурационном пространстве, как правило, является бесконечным. Однако маршрутная сеть должна строится за разумное число испытаний, ограниченное как размером сети, так и успешностью выбора точек при ее построении. В связи с этим, стратегия сэмплирования может являться ключевым фактором эффективности обсуждаемых методов.

В идеальном случае необходимо, чтобы множество выбранных конфигураций было плотным, чтобы отражать все особенности допустимых и недопустимых областей в конфигурационном пространстве. На интуитивном уровне это означает, что наибольшая несэмплированная область пространства должна быть как можно меньше. Следующие определения параметров дисперсии и расхождения обычно используются для оценки качества выборки конфигураций [19].

Определение. Дисперсией выборки точек P в метрическом пространстве $\langle X, dist \rangle$ называется величина, определяемая формулой $\delta(P) = sup_{x\in X} \left\{ min_{p\in P} \{dist(x,p)\} \right\}$.

Рис. 6 дает ее геометрическую интерпретацию в двумерном пространстве с метриками L_2 и L_∞.

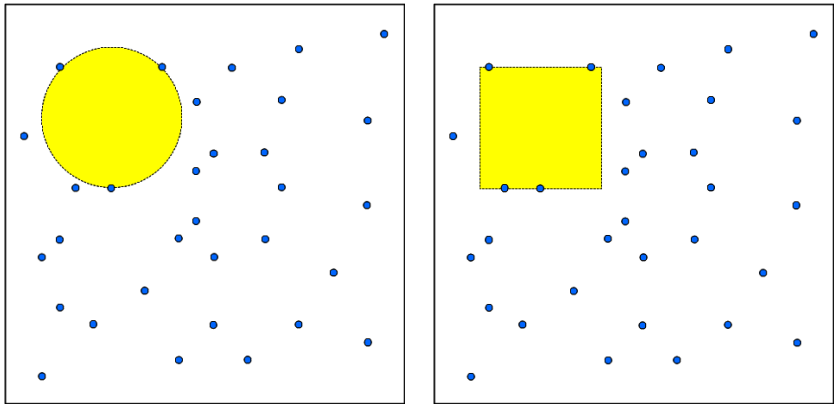


Рис. 6. Геометрическая интерпретация дисперсии выборки в метрическом пространстве L_2(a) и L_∞ (b)

Fig. 6. Geometric interpretation of the dispersion of the set of samples in metric space L_2(a) and L_∞ (b)

Параметр расхождения определяет меру того, насколько выбранные точки равномерно распределены в заданном объеме.

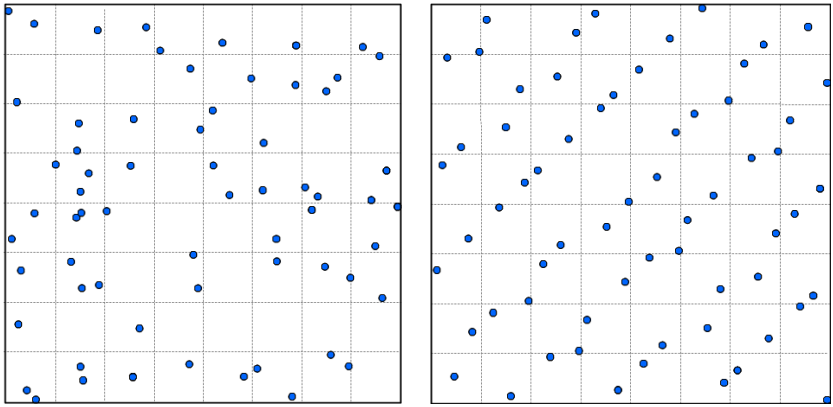
Определение. Пусть заданы компактная область $D \subset X$ в n-мерном метрическом пространстве $\langle X, dist \rangle$ и набор прямоугольников в ней $R = \{r_1, r_2, \dots, r_M\} \subset D$. Расхождение для выборки точек $P = \{p_1, p_2, \dots, p_N\}$ определяется как $\theta(P,R) = sup_{r_i \in R} \left| \frac{|P \cap r_i|}{N} - \frac{\mu(r_i)}{\mu(D)} \right|$, где $\mu(r)$ – функция объема области r.

Точки могут выбираться случайно с равномерным распределением. Однако ввиду недостатков генераторов псевдослучайных чисел данный подход, как правило, не позволяет достичь достаточно равномерного и плотного покрытия за небольшое число итераций. Улучшить качество выборки, выражаемое введенными параметрами, можно за счет использования квазислучайных последовательностей [51].

Известно, что алгоритмы квази-Монте Карло, использующие квазислучайные последовательности точек вместо псевдослучайных чисел, сходятся к искомому результату по крайней мере не медленнее, а обычно быстрее, чем соответствующие алгоритмы Монте Карло [52]. Примером таких последовательностей точек могут служить последовательности Холтона и ЛП$_\tau$-последовательности [53,54].

На рис. 7 представлен единичный квадрат, который разбит на 64 подквадрата и на него нанесены 64 псевдослучайных точки (a) и точки ЛП$_\tau$-последовательности (b). Видно, что в каждый квадрат на правом рисунке

попало ровно по одной точке, в то время, как для левого рисунка это далеко не так.



(a) *(b)*

Рис. 7. Примеры выборок с использованием псевдослучайных (a) и квазислучайных точек (b)

Fig. 7. An example of pseudorandom (a) and quasirandom sampling (b)

Вопросы применения квазислучайных последовательностей в алгоритмах планирования движения подробно рассматриваются в работах [55,56].

6.2 Метрика конфигурационного пространства

Во всех методах маршрутных сетей и сэмплирования конфигурационного пространства требуется определять расстояние между точками выборки. В частности, это требуется при оценке дисперсии выборки в локальной области конфигурационного пространства. Вопрос о выборе функции метрики оказывается нетривиальным, поскольку влияет не только на частные характеристики выборок, но и на качество результирующей маршрутной сети в целом.

Функция метрики может быть реализована путем вычисления евклидова расстояния между заданными конфигурациям по обобщенным координатам. Однако, использование такой метрики оправдано лишь в простых случаях, когда перемещение объекта ограничено поступательным движением. В более общем случае, допускающем группы вращений, например, SE(3), правильно оценить перемещение объекта без учета его геометрической модели уже невозможно.

Наиболее удачный способ оценки расстояния между точками в конфигурационном пространстве заключается в определении наибольшего пути, проходимого точками тела в пространстве сцены (Robot Displacement) [57]. Метрика задается следующим образом: $\rho(c_1, c_2) = \max_{p \in A}\{\|p(c_2) -$

$p(c_1)\|\}$, где $p(c_i)$ – положение точки тела A в конфигурации c_i. Данная метрика позволяет оценить наибольшее расстояние, проходимое каждой точкой твердого тела или любого компонента кинематической системы в пространстве сцены при переходе из положения q_1 в положение q_2 (рис. 8).

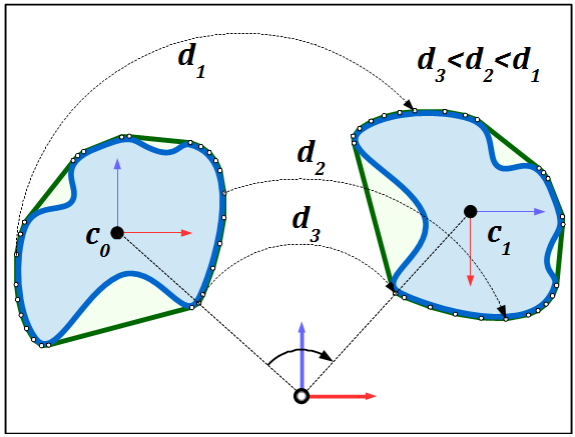


Рис. 8. Определение расстояния между точками в конфигурационном пространстве SO(2)

Fig. 8. Distance calculation between points in SO(2) configuration space

Очевидно, что вычислительные затраты при использовании такой метрики существенно зависят от сложности геометрического представления объекта и его кинематических связей. Для выпуклого твердого тела, заданного полигональной поверхностью, метрика ρ определяется расстоянием, которое проходят только его вершины. Поэтому один из способов ее вычисления заключается в использовании предварительно построенных выпуклых оболочек или ограничивающих параллелепипедов.

6.3 Определение столкновений

В ходе сэмплирования, предусматриваемого методами маршрутных сетей, необходимо определить статус допустимости или бесконфликтности каждой выбранной конфигурации объекта, в том числе, с учетом препятствий сцены. Данная задача относится к известной проблеме определения столкновений в сцене [58,59].

Как правило, определение столкновений представляет собой процесс, условно состоящий из двух основных фаз: широкой и узкой. Задачей первой фазы является быстрая локализация потенциальных столкновений с использованием относительно дешевых негативных тестов. Она эффективно решается с помощью методов ограничивающих объемов [60] и методов пространственной

декомпозиции [61–63]. В узкой фазе выполняется точная проверка выбранных пар объектов на пересечение с учетом их детального геометрического представления. Существует довольно развитый математический аппарат, позволяющий эффективно выполнять данную операцию для различных видов геометрических моделей, в том числе для полиэдральных моделей [59], сплайн-поверхностей [64], твердых тел в граничном представлении [65], CSG-моделей [66] и облаков точек [67].

Формально процедуру определения столкновений можно рассматривать как исчисление предиката $\varphi(\mathrm{c}) = \begin{cases} false, \mathrm{c} \in C_{free}, \\ true, otherwise \end{cases}$ для заданной конфигурации объекта $\mathrm{c} \in C_{free}$. Однако для объединения найденных допустимых конфигураций в маршрутную сеть требуется разрешение более сложных запросов, связанных с существованием бесконфликтных переходов между ними $p(\tau)$: $[0,1] \rightarrow C_{free}$.

На практике данная проблема может быть решена путем соединения пар точек конфигураций непрерывными переходами и анализа их бесконфликтности. Для этого переходы могут быть разбиты на множество отрезков с некоторым фиксированным шагом, а каждый из концов $p(\tau_i)$, $1 \leq i \leq N$ проверен на принадлежность пространству допустимых конфигураций. Выбор шага разбиения представляет отдельную проблему. Недостаточное количество выбранных точек приведет к тому, что перемещаемый объект будет “пролетать” сквозь препятствия. Чтобы избежать данной проблемы, обычно накладывается условие $\|p(\tau_i), p(\tau_{i+1}))\| \leq \varepsilon$, для всех $0 \leq i < N$, которое определяет точность дискретного представления перехода.

Повысить эффективность анализа переходов удается с помощью построения протяжек ограничивающих объемов, а также при использовании результатов проверок предыдущих точек переходов [68].

6.4 Поиск ближайших соседей

Шансы на то, что с помощью простого алгоритма поиска удастся проложить бесконфликтный путь между удаленными друг от друга конфигурациями, как правило, крайне малы. Поэтому построение маршрутной сети на множестве допустимых конфигураций производится путем соединения преимущественно близлежащих точек. Существенно повысить производительность поиска ближайших конфигураций возможно с помощью их предварительного индексирования. В частности, удается эффективно разрешать типовые запросы, связанные с поиском k ближайших точек и поиск ближайших точек в заданном радиусе, за сублинейное время.

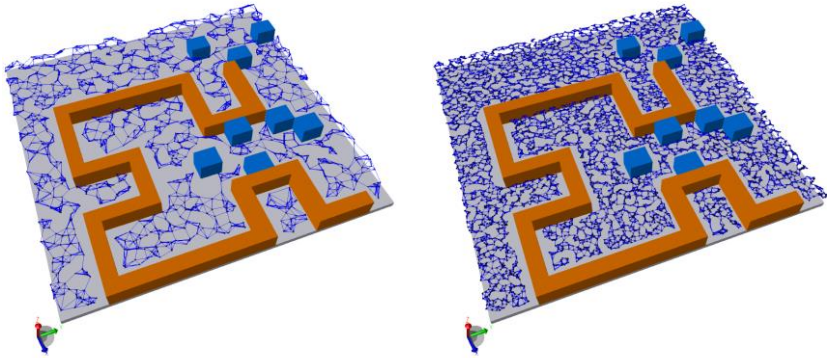
Наиболее широко используемыми для этих целей индексными структурами являются бинарные kd-деревья [69–71], которые строятся путем рекурсивного разбиения пространства гиперплоскостями, ориентированными по осям координат.

Альтернативу им составляют метрические деревья, такие как, M-деревья [72], GNAT (Geometric Near-neighbor Access Tree) [73,74], iDistance-структуры [75] и деревья покрытий (Cover Tree) [76], в основе построения которых лежат алгоритмы иерархической кластеризации. Например, в ходе построения структуры GNAT формирование узлов на каждом уровне дерева выполняется путем выбора фиксированного числа опорных точек с помощью жадного рандомизированного алгоритма *k*-средних. При этом принадлежность точек ветвям дерева определяется на основе матрицы расстояний до центров кластеров.

При решении задач планирования движения *kd*-деревья демонстрируют лучшую производительность по сравнению с другими упомянутыми выше структурами [69]. Однако их практическая реализация сталкивается с проблемой адаптации процедуры пространственной декомпозиции к конкретному способу представления трансформаций объектов. Сопутствующая задача построения ограничивающих параллелепипедов становится нетривиальной в сложных конфигурационных пространствах, включающих подпространства вращения [77]. Наиболее простыми с реализационной точки зрения являются метрические деревья, для реализации которых требуется определить функцию расстояния между конфигурациями. При этом обеспечивается хорошая сбалансированность результирующей структуры. К недостаткам метрических деревьев следует отнести необходимость выбора опорных точек, от расположения и количества которых может существенно зависеть эффективность поиска ближайших соседей.

7 Оффлайн планирование. Методы вероятностных маршрутных сетей

Метод вероятностных маршрутных сетей (Probabilistic Roadmap Method) [42] широко применяется для решения задач поиска путей как в локальной, так и глобальной постановке. Как и в случае ранее рассмотренных детерминированных методов маршрутных сетей, процесс планирования движения включает фазу анализа сцены и фазу построения пути. Отличительной особенностью в данном случае является способ формирования сети из локальных маршрутов, полученных в результате сэмплирования конфигурационного пространства.



(a) *(b)*

Рис. 9. Пример построения вероятностной маршрутной сети (k=5)

((a) 2000 итераций, (b)10000 итераций)

Fig. 9. An example of probabilistic roadmap (k=5)

((a) 2000 iterations, (b)10000 iterations)

В листинге 1 приводится обобщенный алгоритм построения маршрутной сети сэмплинг методом. На каждом из N_{steps} шаге алгоритма генерируется точка c_{rand} в конфигурационном пространстве объекта планирования. Способ генерации новой точки определяется предопределенной стратегией сэмплирования, например, с помощью выбора точки случайным образом с равномерным распределением координат в границах конфигурационного пространства $D.bounds$. Затем выполняется проверка найденной конфигурации на столкновения с препятствиями сцены. Если точка оказывается конфликтной, то она отбрасывается. В успешном случае она включается в сформированное на текущий момент представление вершин маршрутной сети $G.V$. На следующей фазе проводится попытка расширить множество ребер маршрутной сети $G.E$ путем отыскаяни бесконфликтных переходов между ее вершинами. В простейшем случае определяется k ближайших вершин относительно точки c_{rand}. В качестве альтернативного способа могут быть определены все вершины, лежащие внутри n-мерного шара радиуса r с центром в c_{rand}. Значения данных параметров могут фиксироваться или адаптивно меняться в ходе детализации маршрутной сети. Возможность создания ребра с одной из ближайших вершин определяется локальным планировщиком, в качестве которого обычно используют относительно простой и быстрый эвристический алгоритм поиска прямолинейного бесконфликтного пути. Алгоритмические особенности его реализации обсуждались в предыдущем разделе. Однако в ряде случаев возможно применение и более сложных методов. Анализ эффективности методов

локального планирования при построении вероятностных маршрутных сетей приводится в работе [78].

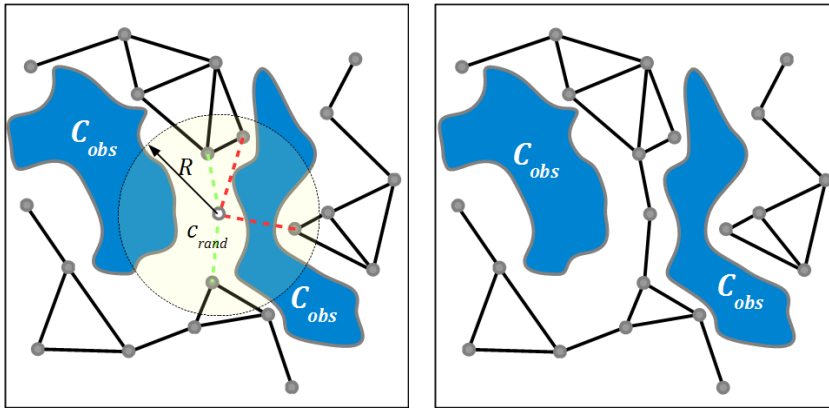


Рис. 10. Включение новой конфигурации маршрутную сеть алгоритмом PRM

Fig. 10. Construction of new roadmap vertex by PRM algorithm

BuildPRM *(N_{steps})*
1. $G(V,E) \leftarrow \emptyset$
2. $step \leftarrow 0$
3. ***while*** *(step <* N_{steps}*)*
4. $c_{rand} \leftarrow SamplingStrategy.GenerateState(D.bounds)$
5. ***if*** *(*$c_{rand} \in C_{free}$*)*
6. $G.V \leftarrow G.V \cup c_{rand}$
7. $U \leftarrow NearestNeighbours(c_{rand}, G)$
8. ***for each*** $c_{neighbour} \in U$
9. ***if*** *(LocalPlanner.VerifyPath(*c_{rand}*,*$c_{neighbour}$*))*
10. $G.E \leftarrow G.E \cup (c_{rand}, c_{neighbour})$
11. $step = step \leftarrow 1$

Листинг 1. Алгоритм построения вероятностной маршрутной сети

Algorithm. 1. An algorithm of probabilistic roadmap construction

Известный недостаток рассмотренного метода связан с недостаточно плотным покрытием, что может приводить к нарушению связности маршрутной сети в узких областях допустимых конфигураций. Решение проблемы путем простого увеличения количества испытаний крайне нежелательно, поскольку определение столкновений является вычислительно затратной операцией. Кроме того, успешные испытания приводят к разрастанию маршрутной сети и увеличению размерности решаемых задач поиска соседних вершин и

верификации переходов. Использование индексных структур, в частности *kd*-деревьев, отчасти решает проблему деградации производительности.

Другим недостатком метода является сложность выбора параметров, таких как радиус поиска соседних вершин и предельное количество инцидентных ребер. Данные параметры существенно влияют на качество маршрутной сети и часто требуют тонкой настройки с учетом специфики прикладной задачи.

7.1 Visibility PRM

Одним из способов уменьшения числа вершин в графе является метод построения рандомизированных маршрутных сетей на основе областей видимости [79]. Данный метод позволяет исключить из анализа избыточные точки, принадлежащие обширным областям допустимых конфигураций, при этом выделив ключевые точки, существенные для навигации в труднопреодолимых зонах.

Для этого в ходе формирования маршрутной сети вершины сети классифицируются как guard (страж) и connector (звено). Случайно выбранная точка относится к первому классу, если не существует вершины, помеченной как guard и лежащей в ее области видимости. Точка классифицируется как connector, если она попадает в область видимости, по меньшей мере, двух вершин guard. И наконец, если точку не удалось отнести ни к одному из двух классов, она вовсе не включается в маршрутную сеть. Тем самым достигается существенное сокращение размера маршрутной сети.

К недостаткам метода следует отнести тот факт, что положение опорных точек класса guard формируется на основе случайного выбора. Поэтому области допустимых конфигураций могут быть построены и распределены неоптимальным образом, что может негативно сказаться на эффективности поиска маршрутов в определенных зонах.

7.2 Vertex Enhancement

Для идентификации слабо связных зон пространства допустимых конфигураций и детализации маршрутной сети в них в основную схему метода часто добавляют процедуру пост-обработки присоединяемых к сети вершин и ребер. В работе [42] для этих целей предлагается использовать информацию о неудачных попытках присоединения вершин, накапливаемую непосредственно в процессе построения сети.

Вес вершины v маршрутной сети рассчитывается на основе статистики испытаний: $s(v) = \frac{n_f}{n_t+1}$, где n_t – общее число попыток присоединения точек к v, а n_f – количество неудавшихся попыток. Таким образом, значение $s(v)$ отражает условную сложность сцены в окрестности v. Распространяя данную оценку на множество всех вершин сети V, можно определить функцию

распределения вероятностей неуспешных попыток как $P(v) = \frac{s(v)}{\sum_{v \in V} s(v)}$ и ее использовать для генерации дополнительных точек в проблемных зонах.

7.3 Obstacle-based PRM

В ряде случаев, например, при наличии узких переходов между обширными областями допустимых конфигураций, представляется разумным выбирать точки преимущественно вдоль границ препятствий и, тем самым, снижать общее количество испытаний и размер сети. Существует ряд модификаций метода построения вероятностных маршрутных сетей, реализующих данный принцип.

В методе Obstacle-based PRM, описанном в работах [80,81], для этих целей предлагается модифицировать базовый алгоритм сэмплирования следующим образом. На каждом шаге выбирается точка c_{rand} с равномерным распределением в границах конфигурационного пространства. Если найдена недопустимая конфигурация $c_{rand} \notin C_{free}$, то проводится некоторое количество попыток обнаружить в ее окрестности допустимую конфигурацию $c' \in C_{free}$. На интервале прямой (c_{rand}, c') ищется новая точка $c_{new} \in C_{free}$, которая лежит как можно ближе к границе препятствия. Чтобы найти такую точку, можно использовать алгоритмы разбиения интервала пополам, золотого сечения или квадратичной интерполяции.

Позже была предложена модификация метода UOBPRM, основанная на использовании ограничивающих параллелепипедов объектов сцены и позволяющая получать бесконфликтные точки с равномерным распределением вдоль границ препятствий [82].

7.4 Bridge-test sampling

Аналогичный принцип применяется в эвристической стратегии сэмплирования узких проходов (Bridge-test Sampling) [83]. Данная стратегия основана на предположении о том, что если для случайно выбранной точки $c \in C_{obs}$ существует отрезок проходящей через нее прямой $s(t) \subset C$ такой, что $c = s(t_0)$, $c' = s(t_1) \in C_{free}$ и $c'' = s(t_2) \notin C_{free}$ при $t_0 < t_1 < t_2$, то точка c' предположительно находится внутри узкого прохода и должна быть включена в маршрутную сеть.

Стоит отметить, что данная стратегия, как правило, отсекает большую часть случайно сгенерированных конфигураций. Однако вычислительные затраты на определение допустимости отдельных избыточных конфигураций существенно ниже, чем на построение и анализ бесконфликтных переходов между ними.

7.5 Gaussian PRM

Другие модификации базового алгоритма используют Гауссово распределение для формирования выборок с большей плотностью вблизи границ препятствий

[84]. По аналогии с формулой размытия, применяемой в алгоритмах обработки изображений, нормальное распределение Гаусса в конфигурационном пространстве C определяется как $\varphi(\mathrm{c},\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}^n} e^{-\frac{\mathrm{c}^2}{2\sigma^2}}$, где $\mathrm{c} \in \mathrm{C}$, n – размерность пространства, а σ – среднеквадратическое отклонение.

Функция размытия определяется следующим образом:

$f(\mathrm{c},\sigma) = \int Obs(y)\,\varphi(\mathrm{c}-y,\sigma)dy$, где $Obs(q) = \begin{cases}1, \mathrm{c} \notin \mathrm{C}_{free}\\ 0, \mathrm{c} \in \mathrm{C}_{free}\end{cases}$.

Для того, чтобы исключить недопустимые конфигурации, распределение задается как $g(\mathrm{c},\sigma) = \max(0, f(\mathrm{c},\sigma) - Obs(\mathrm{c}))$. Тем самым, функция $g(\mathrm{c},\sigma)$ принимает значение 0 для всех недопустимых конфигураций и тождественна функции $f(\mathrm{c})$ для допустимых. Параметр σ определяет близость генерируемых конфигураций к границам препятствий.

Алгоритм построен таким образом, что первая точка выбирается случайным образом с равномерным распределением, а каждая следующая – с распределением, соответствующим приведенной выше формуле.

7.6 Medial axis PRMs

Альтернативный принцип формирования выборок заключается в том, что предпочтение отдается конфигурациям равноудаленным от препятствий сцены [85–87]. Один из возможных способов его реализации заключается в предварительном построении срединной оси в пространстве сцены, которая используется для генерации новых точек.

7.7 Lazy-PRM

Характерным недостатком вероятностных маршрутных сетей является избыточность их представления, поскольку на практике для построения путей обычно требуется гораздо меньшее количество вершин. Это приводит к тому, что значительная часть времени выполнения алгоритма расходуется на вычислительно затратную операцию определения столкновений для конфигураций, несущественных для построения маршрута. Для устранения данного недостатка были предложены методы Fuzzy-PRM [88] и Lazy-PRM [89,90] с отложенной проверкой на бескофликтность переходов между конфигурациями.

Алгоритм поиска пути может быть представлен следующим образом (листинг 2). Вначале строится маршрутная сеть алгоритмом, аналогичным PRM, с тем исключением, что ребра не верифицируются локальным планировщиком (строка 1). Далее предпринимается попытка построить путь в сети (строка 5). В процессе построения ребра проверяются на бесконфликтность (строки 7-11), а конфликтные ребра удаляются из представления маршрутной сети (строка 10). В случае, если не удалось построить достоверный маршрут, соединяющий начальную и конечную конфигурации, разрешение маршрутной сети

увеличивается путем дополнительной генерации точек в слабо связных областях допустимых конфигураций (строка 13).

LazyPRM $(c_{init}, c_{goal}, K_{steps}, N_{samples})$
1. $G(V,E) \leftarrow PRM(N_{samples})$
2. $success \leftarrow false$
3. $step \leftarrow 0$
4. ***while*** $((step < K_{steps})$ & $(success = false))$
5. $p(V',E') \leftarrow FindShortestPath(G, c_{init}, c_{goal})$
6. ***if*** $(p \neq \emptyset)$
7. $success \leftarrow true$
8. ***for each*** $e \in p.E'$
9. ***if*** $(e \notin C_{free})$
10. $G.E = G.E \backslash e$
11. $success \leftarrow false$
12. ***else***
13. *VertexEnhancement(G,* $N_{samples}$ *)*
14. $step = step \leftarrow 1$

Листинг 2. Алгоритм поиска пути на основе вероятностной маршрутной сети с отложенной проверкой на бесконфликтность

Algorithm 2. Probabilistic roadmap algorithm with lazy collision checking

7.8 Dynamic PRM

Вероятностные маршрутные сети позволяют значительно повысить эффективность поиска пути в случае повторных запросов, поскольку вычислительные затраты на предварительный анализ сцены и формирование исходной маршрутной сети уже осуществлены. Однако данное утверждение справедливо исключительно для статических сцен. В случае динамики использование метода затруднительно, поскольку разворачивание новой маршрутной сети при каждом изменении в сцене представляется крайне неэффективным. Существует ряд модификаций метода, позволяющих перестраивать маршрутную сеть более рациональным образом [91–93].

8. Онлайн планирование. Деревья поиска

8.1 Нить Ариадны

Одним из известных сэмплинг-методов планирования движения на основе деревьев поиска является алгоритм “Нить Ариадны” [94,95]. Алгоритм строит маршрутную сеть таким образом, чтобы на каждом шаге покрыть как можно большую новую область конфигурационного пространства, не подлежащую анализу на предыдущих шагах. В зависимости от успешности распространения

алгоритм переключается в один из двух возможных режимов: режим поиска или режим анализа.

В режиме анализа осуществляется рандомизированный выбор опорных точек, или “ориентиров”, максимально удаленных от вершин разворачиваемой сети. В режиме поиска предпринимается попытка присоединить целевую конфигурацию к одному из ориентиров, найденных в режиме анализа.

Ключевым принципом алгоритма является перераспределение вычислительных ресурсов на анализ пространства сцены вместо исключительного продвижения к целевой конфигурации. Это позволяет повысить эффективность маршрутизации в сложных сценах за счет снижения роли “жадной” эвристики в поиске пути и его глобализации. Примечательно, что данный принцип получил свое развитие и в других алгоритмах данного семейства.

В качестве недостатка алгоритма следует указать высокую вычислительную сложность сопутствующей оптимизационной задачи выбора новой вершины в режиме анализа. Для преодоления этой проблемы предлагалось использовать генетические алгоритмы [95], но целесообразность их применения в данном случае вряд ли можно считать оправданной ввиду необходимости настройки множества параметров, специфичных для каждой прикладной задачи планирования движения [19].

8.2 Random-Walk Planner

Простой в реализации и в то же время достаточно эффективный алгоритм поиска пути Random-Walk Planner описан в работе [96]. Представленный алгоритм построен исключительно на рандомизированных перемещениях в конфигурационном пространстве, не используя никакой маршрутной сети.

Направление движения и длина шага определяются на каждой итерации случайным образом на основе нормального гауссова распределения. При этом матрица ковариации формируется на основе фиксированного числа последних испытаний. Таким образом, параметры, определяющие способ распространения, адаптивно настраиваются в ходе работы алгоритма. При этом добавление в путь каждой новой вершины выполняется за константное время в отличие от методов на основе маршрутных сетей.

Основную проблему для описанного алгоритма составляют сцены с узкими проходами и длинными коридорами. Для ее преодоления предложен гибридный вариант, объединяющий базовый алгоритм Random-walk Planner и рандомизированный метод потенциальных полей [97].

8.3 Expansive-Spaces Tree Planner

Планировщик на основе EST-деревьев (Expansive-Spaces Tree planner), описанный в работах [43,98,99], реализует некоторые принципы методов

вероятностных маршрутных сетей применительно к разрешению одиночных запросов поиска пути.

Стратегия сэмплирования в данном методе построена таким образом, чтобы предпочтение отдавалось областям конфигурационного пространства с наименьшей плотностью покрытия маршрутной сетью. Для этого вводится понятие веса конфигурации, который определяется количеством вершин маршрутной сети, лежащих в некоторой окрестности радиуса R от данной точки: $w(c,R) = |\{v \in V | Dist(c,v) < R\}|$. На каждой итерации алгоритма выбирается вершина дерева с вероятностью обратно пропорциональной ее весу. Далее в заданном радиусе генерируется K случайных бесконфликтных конфигураций и предпринимается попытка их включения в маршрутную сеть (листинг 3).

В работе [98] изложен способ повышения производительности алгоритма за счет двунаправленного поиска на основе EST деревьев с отложенной проверкой на бесконфликтность конфигурации (Single Query, Bidirectional Lazy Collision Checking).

К недостаткам рассмотренного алгоритма можно отнести то, что успех поиска сильно зависит от значений константных параметров R и K, которые следует выбирать с учетом особенностей решаемой прикладной задачи.

EST *(c_{init}, N_{steps},K, R)*
1. $T(V,E) \leftarrow (\{c_{init}\}, \emptyset)$
2. $step \leftarrow 0$
3. ***while*** *(step <* N_{steps}*)*
4. $v \leftarrow PickNodeWithProbability\left(T, \frac{1}{w(v,R)}\right)$
5. $U \leftarrow GenerateStates(K, \{c \in C_{free} | Dist(c,v) < R\})$
6. ***for each*** $u \in U$
7. ***if*** *(* $RetainWithProbability\left(\frac{1}{w(u,R)}\right)$*)*
8. ***if*** *(* $e(u,v) \in C_{free}$ *)*
9. $T.V \leftarrow T.V \cup u$
10. $T.E \leftarrow T.E \cup e(v,u)$
11. $step \leftarrow step + 1$

Листинг 3. Алгоритм построения EST дерева

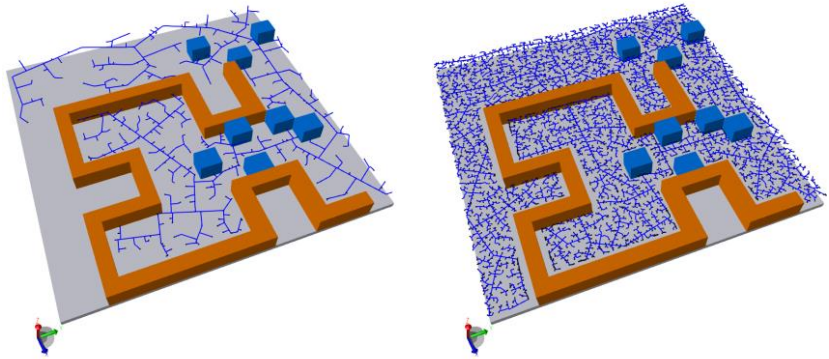
Algorithm 3. EST tree construction algorithm

8.4 Rapidly Exploring Random Trees

Алгоритм на основе быстро растущих случайных деревьев (Rapidly Exploring Random Tree) был изначально разработан для планирования движения неголономных механических систем в режиме реального времени [44,50]. По сравнению с другими известными сэмплинг-методами, эффективность которых

зависит от большого количества настраиваемых входных параметров, алгоритм RRT является наиболее универсальным средством решения широкого круга задач планирования движения.

Данный алгоритм использует в качестве дискретного представления конфигурационного пространства дерево, корень которого соответствует исходному положению объекта. Дерево достраивается таким образом, чтобы разрешение получаемой маршрутной сети увеличивалось на всем пространстве допустимых конфигураций (рис. 11). Таким образом, RRT деревья обладают свойствами, во многом схожими с кривыми, заполняющими пространство (Space-Filling Curves) [100].



(a) *(b)*

Рис. 11. Пример построения дерева поиска алгоритмом RRT

((a) 1000 итераций, (b)20000 итераций)

Fig. 11. An example of search tree construction by RRT algorithm

((a) 1000 iterations, (b)20000 iterations)

Рассмотрим алгоритм распространения RRT дерева (*листинг 4*). На каждой итерации выбирается случайная точка $\mathrm{c}_{\mathrm{rand}} \in \mathrm{C}$. Далее ищется ближайшая к ней вершина дерева $c_{near} \in T$. Отрезок прямой $p' = [\mathrm{c}_{near}, c_{rand}]$ проверяется на конфликтность. В процессе проверки определяется точка $\mathrm{c}_{new} = \begin{cases} c_{rand}, & p' \in C_{free} \\ c_{stop}, & p' \notin \mathrm{C}_{free} \end{cases}$, где c_{stop} – бесконфликтная конфигурация на данном отрезке такая, что $[\mathrm{c}_{near}, c_{stop}] \in p' \cap C_{free}$ и $\left\|\mathrm{c}_{stop}, C_{obs} \cap p'\right\| \leq \varepsilon$, а ε – погрешность определения конфликтов (рис. 12). Точка c_{new} и ребро $[\mathrm{c}_{near}, c_{new}]$ включаются в маршрутную сеть при условии $\mathrm{c}_{new} \neq \mathrm{c}_{rand}$.

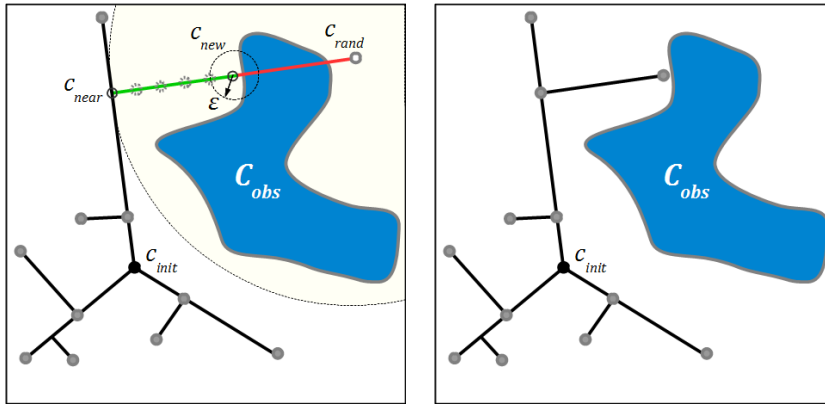


Рисунок 12. Включение новой конфигурации в дерево поиска алгоритмом RRT

Fig. 12. Construction of new tree vertex by RRT algorithm

Важно отметить, что алгоритм распространения построен таким образом, что вероятность выбора каждой вершины дерева пропорциональна объему региона вороного, которому она принадлежит. Данная особенность алгоритма обуславливает высокую скорость “разрастания” дерева в пространстве.

RRT $(c_{init}, c_{goal}, N_{steps}, N_{extend}, \rho)$
$T(V,E) \leftarrow (\{c_{init}\}, \emptyset)$
$step \leftarrow 0$
$success \leftarrow false$
while *(step <* N_{steps}*) & (success=false))*
  if $(step \bmod N_{extend} \neq 0)$
    $c_{rand} \leftarrow GenerateState()$
  else
    $c_{rand} \leftarrow c_{goal}$
  $c_{near} \leftarrow NearestNeighbour(c_{rand}, T)$
  $c_{new} \leftarrow FindStoppingState(c_{near}, c_{rand}, \rho)$
  if *(* $c_{new} \neq c_{near}$ *)*
    $T.V = T.V \cup c_{new}$
    $T.E = T.E \cup (c_{near}, c_{new})$
    $success \leftarrow \left(Distance(c_{new}, c_{goal}) \leq \rho\right)$
$step \leftarrow step + 1$
return *success*

Листинг 4. Алгоритм поиска пути с использованием RRT дерева

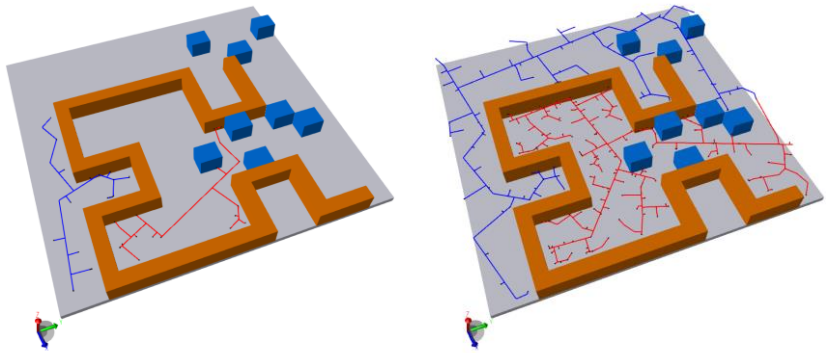
Algorithm 4. Path planning using RRT algorithm

Поиск пути осуществляется в результате периодически предпринимаемых попыток включить целевую конфигурацию в маршрутную сеть. Следует

отметить, что частое выполнение данной операции повлечет за собой неоправданное завышение вычислительных затрат, свойственное методам локального планирования и, в частности, методам потенциальных полей.

8.5 RRT-Connect

В качестве одного из способов повышения эффективности классического метода RRT-деревьев было предложено использовать двунаправленный поиск [50,101].



(a) *(b)*

Рис. 13. Пример построения деревьев поиска алгоритмом RRT-connect

((a) 250 итераций, (b)964 итерации)

Fig. 13. An example of search trees construction by RRT-connect algorithm

((a) 250 iterations, (b)964 iterations)

Для формирования маршрутной сети в алгоритме RRT-connect (листинг 5.) используется пара деревьев T_1 и T_2, корням которых соответствуют начальная конфигурация c_{init} и конечная конфигурация c_{goal} искомого пути. На каждом шаге одно из деревьев дополняется новыми бесконфликтными конфигурациями в соответствии с алгоритмом RRT. Однако вместо периодических попыток включения c_{goal} выполняется операция сращивания деревьев. При каждом включении новой вершины в T_1 предпринимается аналогичная попытка включить ее и в T_2.

При выполнении некоторого количества шагов деревья меняются ролями, что обеспечивает их более сбалансированный рост. Алгоритм построен таким образом, что предпочтение отдается тому дереву, распространение которого затруднено (например, в связи с наличием большого количества препятствий в его области). Количественным критерием в данном случае может служить общее число вершин дерева или суммарная длина ребер.

RRTConnect *(c_{init}, c_{goal}, N_{steps},ρ)*
1.$T_1(V,E) \leftarrow (\{c_{init}\},\emptyset)$
2.$T_2(V,E) \leftarrow (\{c_{goal}\},\emptyset)$
3.$step \leftarrow 0$
4.$success \leftarrow false$
*5.****while*** *((step <* N_{steps}*) & (success=false))*
6. $c_{rand} \leftarrow GenerateState()$
7. $c_{near} \leftarrow NearestNeighbour(c_{rand}, T_1)$
8. $c_{new} \leftarrow FindStoppingState(c_{near}, c_{rand}, \rho)$
9. ***if*** *(* $c_{new} \neq c_{near}$ *)*
10. $T_1.V = T_1.V \cup c_{new}$
11. $T_1.E = T_1.E \cup (c_{near}, c_{new})$
12. $c^*_{near} \leftarrow NearestNeighbour(c_{new}, T_2)$
13. $c^*_{new} \leftarrow FindStoppingState(c^*_{near}, c_{new}, \rho)$
14. ***if*** *(* $c^*_{new} \neq c^*_{near}$ *)*
15. $T_2.V = T_2.V \cup c^*_{new}$
16. $T_1.E = T_1.E \cup (c^*_{near}, c^*_{new})$
17. $success \leftarrow (Distance(c_{new}, c^*_{new}) \leq \rho)$
18. ***if*** *(* $|T_2| > |T_1|$ *)*
19. *Swap(* T_1, T_2 *)*
20. $step \leftarrow step + 1$
*21.****return*** *success*

Листинг 5. Алгоритм двунаправленного поиска пути с использованием RRT деревьев (RRT-connect)

Algorithm 5. Bidirectional version of RRT algorithm (RRT-connect)

Существует ряд работ, посвященных применению многих быстро растущих деревьев для локального планирования [102–104], однако вопрос о разумном компромиссе между количеством разворачиваемых деревьев и затратами на их распространение остается открытым [19].

8.6 Execution Extended RRT

В работе [105] предлагается способ повысить эффективность исполнения множественных запросов планирования движения за счет кэширования бесконфликных конфигураций. Таким образом, каждый последующий запрос планирования использует бесконфликтные переходы, обнаруженные в результате работы RRT алгоритма ранее.

Данная идея хорошо сочетается с двунаправленным поиском. Предложенный алгоритм Multi-Bridge ERRT [106] заимствует основной принцип RRT-connect, однако процедура распространения деревьев продолжается и после сращивания. Ввиду того, что структура, полученная в результате работы

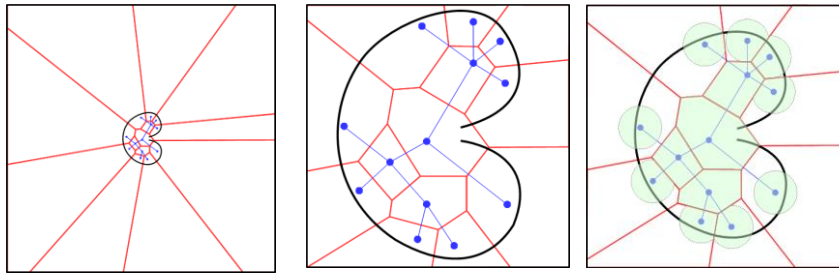
алгоритма уже не является деревом, результирующий путь ищется в графе с помощью алгоритма A^*.

8.7 Dynamic Domain RRT

Как было отмечено, на каждой итерации RRT алгоритма вероятность выбора вершины дерева прямо пропорциональна объему региона Вороного, которому она принадлежит. Данный факт обуславливает высокую скорость разрастания дерева в конфигурационном пространстве. Однако распространение может быть затруднено в тех случаях, когда внешние вершины дерева распределены преимущественно вдоль границ препятствий, а покрываемая деревом область конфигурационного пространства относительно невелика.

В качестве примера можно привести ситуацию, когда начальная точка пути находится внутри небольшой комнаты с единственным узким проходом (рис. 14 a, b). Если новые конфигурации генерируются с равномерным распределением во всем доступном объеме, то вероятность попадания случайно выбранной точки в область прохода может быть ничтожно мала.

Для решения описанной проблемы в работе [107] был предложен алгоритм Dynamic Domain RRT, динамически ограничивающий область сэмплирования в ходе формирования дерева. Для RRT дерева, построенного на множестве вершин V, вводится понятие динамической области $O = \bigcup_{v \in V} Domain(v, R)$. Каждый фрагмент области определяется как $Domain(v, R) = \begin{cases} D(v) \cap B_R(v), & min(\|v, C_{obs}\|) \leq \varepsilon \\ D(v), & min(\|v, C_{obs}\|) > \varepsilon \end{cases}$, где $D(v)$ – регион Вороного, которому принадлежит вершина v, а $B_R(v)$ – шар радиуса R с центром в v (рис. 14, c).



(a) *(b)* *(c)*

Рис. 14. Регионы Вороного, построенные на множестве вершин RRT дерева (a, b) и динамическая область сэмплирования (c).

Fig. 14. Voronoi regions associated with the nodes of tree constructed by RRT planner (a, b) and dynamic sampling domain (c).

Алгоритм построен таким образом, что распространение маршрутной сети осуществляется за счет конфигураций, принадлежащих динамической области O (листинг 6). В ходе построения дерева вершины помечаются как внешние или

внутренние. Включение случайно выбранной точки $c_{rand} \in C$ в дерево выполняется лишь при условии, что ближайшая к ней вершина является внутренней или лежит от нее на расстоянии, меньшем R. Значение радиуса R рекомендуется выбирать, исходя из максимально допустимой погрешности определения конфликтов, определяемой, например, линейной функцией $R = \lambda\varepsilon$, где λ – положительный целочисленный коэффициент.

DynamicDomainRRT *(c_{init},N_{steps},R)*
1. $T(V,E) \leftarrow (\{c_{init}\},\emptyset)$
2. $step \leftarrow 0$
3. ***while*** *(step <* N_{steps}*)*
4. repeat
5. $c_{rand} \leftarrow GenerateState(T.bounds)$
6. $c_{near} \leftarrow NearestNeighbour(c_{rand},T)$
7. ***until*** *(* $Distance(c_{near},c_{rand}) < c_{near}.radius$ *)*
8. $c_{new} \leftarrow FindStoppingState(c_{near},c_{rand})$
9. ***if*** *(* $c_{new} \neq c_{near}$ *)*
10. $c_{new}.radius \leftarrow \infty$
11. $T.V = T.V \cup c_{new}$
12. $T.E = T.E \cup (c_{near},c_{new})$
13. else
14. $c_{new}.radius \leftarrow R$
15. UpdateBounds(T.bounds, c_{new} *)*
16. $step = step \leftarrow 1$

Листинг 6. Построение дерева в алгоритме Dynamic-Domain-RRT

Algorithm 6. Construction of tree using Dynamic-Domain-RRT algorithm

8.8 Iterative Diffuse Path Planner

Применение метода RRT деревьев вместе с лежащей в его основе “жадной” эвристикой может быть крайне нежелательным для ряда задач планирования движения. Примером может служить задача автоматической разборки изделий, состоящих из множества деталей. Движение детали вблизи исходной конфигурации, как правило, сильно ограничено, и возможно лишь за счет малых поворотов. И наоборот, в области целевой конфигурации, находящейся на значительном удалении, препятствия вовсе отсутствуют.

Принцип отложенной проверки на бесконфликтность конфигураций в сочетании с динамически изменяемой погрешностью лежит в основе диффузионного алгоритма планирования движения (Iterative Diffuse Path Planner) (листинг 7). Алгоритм позволяет значительно повысить эффективность поиска в постановках, подобных приведенной [45].

В данном алгоритме используется лес растущих деревьев. Распространение осуществляется из произвольной вершины в направлении, определяемом случайным вектором, с шагом, равному текущему значению погрешности

определения конфликтов σ. После включения каждой новой вершины в дерево предпринимается попытка его слияния. Верификация ребер на конфликты в процессе распространения не выполняется.

Значение σ инициализируется достаточно большим значением, например, равным расстоянию между c_{init} и c_{goal}, и итерационно уменьшается вплоть до заданного минимально допустимого значения. На каждой итерации выполняется поиск пути описанным выше способом с последующей его верификацией. В случае, если путь оказался конфликтным, из представления деревьев исключаются все ребра, не прошедшие проверку, а процедуры распространения и поиска пути повторяются.

DiffuseRRT *(T(V,E), σ)*
1.$v_{rand} \leftarrow ChooseRandomNode(T.V)$
2.$c_{rand} \leftarrow ShootNearNode(v_{rand})$
3.$v_{near} \leftarrow NearestNeighbour(c_{rand}, T)$
4.$c_{new} \leftarrow FindStoppingState(v_{near}, c_{rand})$
5.***if*** $(c_{new} \neq v_{near})$
6. $T.V \leftarrow T.V \cup c_{new}$
7. $T.E \leftarrow T.E \cup (v_{near}, c_{new})$
8. ***for each*** $T^* \in G \backslash T$
9. *LinkTrees*(c_{new}, T^*)

IterativeDiffusionPathPlanner $(c_{init}, c_{goal}, \varepsilon)$
1.$G(V,E) \leftarrow \emptyset$
2.$\sigma \leftarrow Distance(c_{init}, c_{goal})$
3.***while*** $(\sigma > \varepsilon)$
4. $p(V', E') \leftarrow DiffuseRRT(c_{init}, c_{goal}, G, \sigma)$
5. $\sigma \leftarrow {}^{\sigma}/_{\alpha}$
6. ***for each*** $e \in p.E'$
7. ***if*** $(e \notin C_{free})$
8. $G.E \leftarrow G.E \backslash e$
9.$p \leftarrow DiffuseRRT(c_{init}, c_{goal}, G)$
10.***return*** *p*

Листинг 7. Итеративный дифузионный алгоритм поиска пути

Algorithm 7. Iterative diffuse path planning algorithm

9. Поиск оптимальных путей сэмплинг-методами

Удовлетворяя требованиям вероятностной успешности, рассмотренные выше сэмплинг-методы с успехом применяются к практическим задачам планирования движения, прежде неразрешимым за приемлемое время. Однако рандомизированный характер поиска, присущий PRM и RRT методам, крайне негативно влияет на качество получаемых решений. Подразумевается, что в

качестве количественных критериев качества решений могут выступать длина и гладкость траектории, расстояние между движущимся объектом и препятствиями сцены, стоимость преодоления препятствий, затраты энергии и т.п.

9.1 Офлайн оптимизация

Один из подходов к решению данной проблемы состоит в пост-обработке найденных бесконфликтных путей, основанной на их итерационной локальной оптимизации.

Довольно простым и популярным является итерационный алгоритм укорачивания пути [2]. На каждом шаге алгоритма выбирается пара точек, лежащих на пути. Первая точка выбирается случайно, а вторая – в некотором радиусе от нее. Причем точки не обязаны совпадать с вершинами пути. Далее точки соединяются отрезком прямой в конфигурационном пространстве, а отрезок проверяется на бесконфликтность или, другими словами, на принадлежность пространству допустимых конфигураций. В случае успеха все точки, лежащие между выбранными на исходном пути, выбрасываются, а отрезок включается в представление пути.

В результате пост-обработки, как правило, получается семейство гомотопных путей и, следовательно, оптимальное решение может быть недостижимо. Альтернативный способ оптимизации пути состоит в гибридизации или объединении участков из нескольких ранее найденных бесконфликтных путей [108].

9.2 Онлайн оптимизация

Несмотря на то, что RRT и PRM методы не учитывают характер переходов между конфигурациями, улучшить получаемые решения возможно путем модификации базовых алгоритмов. Оптимизация может осуществляться как за счет изменения способов распространения в пространстве [109,110], так и за счет изменения правил включения бесконфликтных конфигураций в маршрутную сеть [3].

9.3 Heuristic RRT

Несколько иной способ повысить качество путей предложен в работе [109]. В алгоритме hRRT (Heuristic RRT) вводится дополнительная эвристика, корректирующая функцию распределения вероятностей при выборе новых конфигураций таким образом, чтобы отдавать предпочтение путям наименьшей стоимости. Вероятность выбора конфигурации определяется двумя факторами. Во-первых, как и в базовом алгоритме RRT, она зависит от объема региона Вороного, которому она принадлежит. А во-вторых, она определяется предполагаемой стоимостью пути, проходящего через данную

конфигурацию. Для этого вводится следующая оценка: $m_{quality} = 1 - \frac{c_{vertex} - c_{opt}}{c_{max} - c_{opt}}$, где c_{vertex} – суммарная стоимость пути из начальной конфигурации в данную вершину и из данной вершины до целевой конфигурации, c_{opt} – предполагаемая стоимость оптимального пути из начальной конфигурации в целевую, а c_{max} – максимальная стоимость пути до любой вершины дерева на текущем шаге. Таким образом, числитель представляет собой оценку отклонения от предполагаемого оптимального пути, а знаменатель служит масштабирующим коэффициентом для нормирования результата. Данная оценка используется при сэмплировании конфигурационного пространства. На каждом шаге RRT алгоритма случайным образом выбирается несколько конфигураций, однако в дерево включается конфигурация с наименьшей стоимостью пути.

9.4 Transition-based RRT

Эвристика, применяемая в алгоритме hRRT, заставляет прорастать дерево к целевой конфигурации, часто игнорируя при этом более оптимальные пути. Метод T-RRT (Transition-base RRT) больше подходит для планирования движения в сложных стоимостных пространствах [110]. Для выбора конфигураций наименьшей стоимости в данном методе используется алгоритм имитации отжига (Simulated Annealing). Решение о включении новой конфигурации в дерево принимается с учетом ее стоимости относительно стоимости ближайшей вершины. Вероятность включения конфигурации c при заданной функции стоимости $f(c)$ определяется следующим образом: $P(\mathrm{c}) = \begin{cases} \exp\left(-\frac{\Delta f}{KT}\right), \Delta f > 0 \\ 1 \quad\quad , \Delta f \leq 0 \end{cases}$, где Δf – отношение приращения стоимости между конфигурациями к расстоянию между ними $\Delta f = \frac{f(c_{new}) - f(c_{near})}{dist(c_{near}, c_{new})}$, $K = \frac{f(c_{init}) + f(c_{goal})}{2}$ – нормирующий коэффициент, рассчитываемый как среднее значение стоимости начальной и целевой конфигураций, а T – числовой параметр, называемый температурой.

Таким образом, чем больше приращение стоимости, тем меньше вероятность включения конфигурации в дерево. При этом допускаются все переходы в конфигурации с меньшей стоимостью. Параметр T позволяет контролировать характер поиска. При очень высоких значениях температуры принимаются почти все новые точки. При низких значениях – распространение происходит преимущественно за счет исключительных точек, увеличивающих стоимость пути незначительно. Процесс поиска начинается при низкой температуре, а затем продолжается при возрастающих значениях температуры, которая повышается всякий раз когда количество точек, не удовлетворяющих критерию включения в дерево, достигает заданного предельного значения.

9.5 PRM*, RRT*

В работе [3] предложены модификации алгоритмов PRM и RRT, предназначенные для получения асимптотически оптимальных решений.

Определение. Алгоритм ALG асимптотически оптимален, если он вероятностно успешен, а также для любой задачи поиска пути $\langle C_{free}, c_{init,} c_{goal}\rangle$ со стоимостной функцией $f_c(p)$ и оптимальным решением p^* имеет место $P(\{lim_{n\to\infty} \sup Y_n^{ALG} = f_c(p^*)\}) = 1$, где Y_n^{ALG} – минимальное значение стоимости для всех решений, полученных за n шагов алгоритма.

RRT* повторяет базовый алгоритм RRT за исключением процедуры включения новых конфигураций в дерево поиска (листинг 8). Данная процедура построена таким образом, чтобы стоимость путей, входящих в дерево, уменьшалась с каждым шагом алгоритма.

В ходе построения дерево поиска дополняется значениями стоимости пути, ведущего в каждую из его вершин. Как и в базовом алгоритме, на каждом шаге для случайным образом выбранной точки c_{rand} ищется ближайшая вершина графа c_{near}, а также точка $c_{new} \in C_{free}$, полученная в результате распространения из c_{near} в c_{rand} (строки 4-7). Далее определяется подмножество вершин дерева C^*, лежащих внутри шара радиуса r с центром в c_{new}. Среди них находят вершину c_{min} такую, что стоимость пути из начальной конфигурации c_{init} в c_{new} была минимальной (строки 8-10).

После того как c_{new} включается в дерево в качестве дочернего узла c_{min} (строки 11-12), проводится локальная оптимизация путей внутри шара. Для каждой конфигурации в нем $c' \in C^*$ предпринимается попытка построения пути, ведущего в c_{new}, и в случае, если он имеет меньшую стоимость, то конфигурация c' становится дочерней вершиной c_{new} (строка 13).

Радиус поиска в данном алгоритме представляет собой функцию от количества вершин дерева и адаптивно уменьшается с его детализацией. В работе [3] доказывается, что алгоритм RRT* асимптотически оптимален при $r(|V|) = \gamma_{RRG}\left(\frac{\log(|V|)}{|V|}\right)^{\frac{1}{d}}$, где $\gamma_{RRG} > 2\left(1+\frac{1}{d}\right)^{\frac{1}{d}}\left(\frac{\mu(C_{free})}{\mu(B_1(\cdot))}\right)$, $|V|$ – количество вершин дерева на данном шаге алгоритма, d – размерность конфигурационного пространства, $\mu(C_{free})$ – объем пространства допустимых конфигураций, $\mu(B_1(\cdot))$ – объем d-мерного единичного шара.

RRTStar *(c_{init},N_{steps})*
1. $T(V,E) \leftarrow (\{c_{init}\}, \emptyset)$
2. $step \leftarrow 0$
3. ***while*** *(step <* N_{steps}*)*
4. $c_{rand} \leftarrow GenerateState()$
5. $c_{near} \leftarrow NearestNeighbour(c_{rand}, T)$
6. $c_{new} \leftarrow FindStoppingState(c_{near}, c_{rand})$
7. ***if*** *(* $c_{new} \neq c_{near}$ *)*
8. $r \leftarrow SearchRadius(step)$

9. $C^* \leftarrow NearestNeighbours(c_{new}, T, r)$
10. $c_{min} \leftarrow MinCostParent(C^*)$
11. $T.V = T.V \cup c_{new}$
12. $T.E = T.E \cup (c_{min}, c_{new})$
13. *Rewire(T, C*, c_{min}, c_{new})*
14. $step = step \leftarrow 1$

*Листинг 8. Построения дерева поиска алгоритмом RRT**

Algorithm 8. Construction of tree using RRT algorithm*

В последние годы методы PRM* и RRT* получили дальнейшее развитие. Разработано большое количество алгоритмов, демонстрирующих более высокие показатели эффективности поиска и обладающих лучшей сходимостью к оптимальным решениям.

В работах [111,112] предложены алгоритмы Lazy-PRM* и Lazy-RRG*, реализующие принцип отложенной проверки на конфликты. Оба алгоритма асимптотически оптимальны и в ряде случаев позволяют ускорить поиск пути.

Для устранения проблемы избыточного количества ребер в маршрутных сетях, полученных в результате работы алгоритма PRM*, были предложены модификации для поиска путей, близких к оптимальным. В основе алгоритма IRS [113–115] лежит инкрементальный способ построении α-спаннера графа. При количестве итераций, стремящемся к бесконечности, и заданном коэффициенте α алгоритм сходится к оптимальному решению с точностью $1 + \alpha$ и вероятностью, равной единице. Схожий принцип используется в алгоритме SPARS, который заимствует идеи инкрементального построения спаннера и отсечения точек на основе областей видимости [116,117].

Для повышения скорости сходимости к оптимальному решению было предложено дополнить алгоритм RRT* процедурой перепланирования, не ограничивающейся локальной оптимизацией ребер. В алгоритмах RRT*-smart [118] и RRT# [119] участки, потенциально являющиеся частью оптимального пути наименьшей стоимости, оптимизируются по всей длине, начиная от корня дерева.

В работе [120] описывается асимптотически оптимальный алгоритм FMT* (Fast Marching Tree), который строит маршрутную сеть на множестве случайных бесконфликтных конфигураций и одновременно поддерживает ее остовное дерево с корнем в начальной точке пути и минимальными значениями стоимости путей до его вершин.

Алгоритм LBT-RRT (Lower Bound Tree RRT) [121], также являющийся развитием RRT*, позволяет повысить эффективность поиска за счет смягчения требования асимптотической оптимальности. Для планирования движения в сложных стоимостных пространствах был разработан гибридный алгоритм TRRT*, использующий алгоритм имитации отжига [122]. В работах [123,124]

были предложены модификации RRT* и FMT*, использующие двунаправленный поиск.

10 Заключение

Таким образом, в работе представлен обзор задач и методов современной теории планирования движения. Рассмотрены основные факторы, определяющие особенности прикладных задач и влияющие на выбор применяемых математических методов. К ним отнесены характер постановки планирования движения (локальный или глобальный), геометрическое представление перемещаемого объекта (твердое тело или кинематическая конструкция), свойства окружения (статическое или динамическое), свойства конфигурационного пространства (равномерное распределение допустимых состояний или наличие узких областей), характер запросов планирования (однократный или многократный). Также представлены основные подходы к планированию движения, связанные с пространственной декомпозицией сцены, физическими аналогиями с потенциальными полями, маршрутными сетями и быстро растущими деревьями. Детально рассмотрены ключевые методы и алгоритмы, реализующие данные подходы, и аспекты их практического применения.

Предполагается, что обзор поможет в выборе методов с учетом особенностей решаемых прикладных задач и в их эффективной программной реализации и настройке. Ожидается также, что обзор послужит основой для систематизации и объектной концептуализации теории планирования движения, необходимой для создания единой программно-инструментальной среды разработки приложений.

Список литературы

[1]. Choset H., Lynch K., Seth H., Kantor G., Burgard W., Kavraki L.E., Thrun S. Principles of Robot Motion-Theory, Algorithms , and Implementation. MIT Press, 2005.
[2]. Geraerts R., Overmars M.H. Creating High-quality Paths for Motion Planning. Int. J. Rob. Res., vol. 26, № 8, 2007, pp. 845–863.
[3]. Karaman S., Frazzoli E. Sampling-based algorithms for optimal motion planning. Int. J. Robot., vol. 30, № 7, 2011, pp. 846–894
[4]. Laumond J.-P. Kineo CAM: A success story of motion planning algorithms. IEEE Robot. Autom. Mag., vol. 13, № 2, 2006, pp. 90–93.
[5]. Rockel S., Klimentjew D., Zhang L., Zhang J. An hyperreality imagination based reasoning and evaluation system (HIRES). Proc. IEEE Int. Conf. on Robotics and Automation, 2014. pp. 5705–5711.
[6]. Sucan I., Moll M., Kavraki L.E. The Open Motion Planning Library. IEEE Robot. Autom. Mag. vol. 19, № 4, 2012, pp. 72–82.
[7]. Diankov R. Automated Construction of Robotic Manipulation Programs. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010, 263 p.

[8]. Porta J.M., Ros L., Bohigas O., Manubens M., Rosales C., Jaillet L. The CUIK Suite: Motion Analysis of Closed-chin Multibody Systems. IEEE Robot. Autom. Mag., vol. 21, № 3, 2014, pp. 105–114.
[9]. Semenov V.A., Kazakov K.A., Zolotov V.A. Effective spatial reasoning in complex 4D modeling environments. eWork Ebus. Archit. Eng. Constr. eds. A.Mahdavi, B. Martens, R. Scherer, CRC Press. Taylor Fr. Group, London, UK. 2015, pp. 181–186.
[10]. Kazakov K.A., Semenov V.A., Zolotov V.A. Topological Mapping Complex 3D Environments Using Occupancy Octrees. 21st Int. Conf. Comput. Graph. Vision, Sept. 26-30, 2011, Moscow, Russ. 2011, pp. 111–114.
[11]. Semenov V.A., Kazakov K.A., Morozov S. V., Tarlapan O.A., Zolotov V.A., Dengenis T. 4D modeling of large industrial projects using spatio-temporal decomposition. eWork Ebus. Archit. Eng. Constr. eds. K. Menzel R. Scherer, CRC Press. Taylor Fr. Group,London, UK. 2010, pp. 89–95.
[12]. Brooks R.A., Lozano-Peres T. A Subdivision Algorithm in Configuration Space For Find Path with Rotation. IEEE Trans. Syst. Man. Cybern., vol. SMC-15, № 2, 1985, pp. 224–233.
[13]. D. Zhu and J.-C. Latombe. Constraint reformulation in a hierarchical path planner. In Proc. IEEE Int. Conf. on Robotics and Automation, 1990, pp. 1918-1923.
[14]. Hornung A., Wurm K.M., Bennewitz M., Stachniss C., Burgard W. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. Auton. Robots., vol. 34, № 3, 2013, pp. 189–206.
[15]. Semenov V.A., Kazakov K.A., Zolotov V.A. Global path planning in 4D environments using topological mapping. eWork Ebus. Archit. Eng. Constr. 2012, pp. 263–269.
[16]. Chen D.Z., Szczerba R.J., Uhran Jr J.J. Using Framed-Quadtrees to Find Conditional Shortest Paths in an Unknown 2-D Environment. Technical Report #95-2, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, Indiana, Jan. 1995, 33 p.
[17]. Chazelle B. Approximation and Decomposition of Shapes. Advances in Robotics 1: Algorithmic and Geometric Aspects of Robotics, 1987, pp. 145–185.
[18]. De Berg M., Cheong O., Van Kreveld M., Overmars M. Computational Geometry: Algorithms and Applications. 3rd Edition. Springer-Verlag, 2008, 386 p.
[19]. LaValle S.M. Planning Algorithms. Cambridge University Press, 2006, 844 p.
[20]. Canny J.F., Lin M.C. An opportunistic global path planner. Algorithmica, vol. 10, № 2-4, 1993, pp. 102–120.
[21]. Huang H.-P.H.H.-P., Chung S.-Y.C.S.-Y. Dynamic visibility graph for path planning. Proc. Int. Conf. Intell. Robot. Syst., vol. 3, 2004, pp. 2813–2818.
[22]. Kaluder H., Brezak M., Petrovic I. A visibility graph based method for path planning in dynamic environments. MIPRO, 2011, Proceedings of the 34th International Convention, Opatija, Croatia, 2011, pp. 717–721.
[23]. Aurenhammer F. Voronoi Diagrams – A Survey of a Fundamental Data Structure. ACM Comput. Surv., vol. 23, № 3, 1991, pp. 345–405.
[24]. Choset H. Sensor based motion planning: The hierarchical generalized voronoi graph. Ph.D. Thesis, California Institute of Technology, 1996, 201 p.
[25]. Russell S.J., Norvig P. Artificial Intelligence: A Modern Approach. Neurocomputing, vol. 9, № 2, 1995, pp. 215-218.
[26]. Bell S., An Overview of Optimal Graph Search Algorithms for Robot Path Planning in Dynamic or Uncertain Environments. Oklahoma Christian University, 2010, 9 p.

[27]. De Filippis L., Guglieri G. Advanced graph search algorithms for path planning of flight vehicles. Recent Adv. Aircr. Technol., 2012, pp. 159–192.
[28]. Zeng W., Church R.L. Finding shortest paths on real road networks: the case for A*. Int. J. Geogr. Inf. Sci., vol. 23, № 4, 2009, pp. 531–543.
[29]. Korf R.E. Depth-first iterative-deepening. An optimal admissible tree search. Artif. Intell., vol. 27, № 1, 1985, pp. 97–109.
[30]. Zhou R., Hansen E.A. Memory-Bounded {A*} Graph Search. The Florida AI Research Society Conference - FLAIRS, 2002, pp. 203–209.
[31]. Holte R., Perez M., Zimmer R., MacDonald A. Hierarchical A*: Searching abstraction hierarchies efficiently. Proceeding AAAI'96 Proceedings of the thirteenth national conference on Artificial intelligence – Volume 1, 1996, pp. 530–535.
[32]. Botea A., Muller M., Schaeffer J. Near optimal hierarchical path-finding. Journal of Game Development, vol. 1, issue 1, 2004, pp. 7-28.
[33]. Kring A., Champandard A., Samarin N. DHPA* and SHPA*: Efficient Hierarchical Pathfinding in Dynamic and Static Game Worlds. Proc. Sixth Artificial Intelligence and Interactive Digital Entertainment Conference, 2010, pp. 39–44.
[34]. Harabor D., Grastien A. Online Graph Pruning for Pathfinding On Grid Maps. AAAI Conf. Artif. Intell., 2011, pp. 1114–1119.
[35]. Koenig S., Likhachev M., Furcy D. Lifelong Planning A*. Artif. Intell., vol. 155, № 1-2, 2004, pp. 93–146.
[36]. Stentz A. The Focussed D* Algorithm for Real-Time Replanning. Proceeding IJCAI'95 Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2, 1995, pp. 1652-1659.
[37]. Koenig S., Likhachev M. D* Lite. Proceedings of the AAAI Conference of Artificial Intelligence (AAAI), 2002, pp. 476–483.
[38]. Koenig S., Likhachev M. Fast Replanning for Navigation in Unknown Terrain. IEEE Trans. Robot., vol. 21, issue 3, 2005, pp. 354-363..
[39]. Khatib O. Real time obstacle avoidance for manipulators and mobile robots. International Journal of Robotics and Research, vol. 5, № 1, 1986, pp. 90–98.
[40]. Barraquand J., Latombe J.C. A Monte-Carlo algorithm for path planning with many degrees of freedom. Proceedings 1990 IEEE International Conference on Robotics and Automation, vol.3, 1990, pp. 1712–1717.
[41]. Barraquand J., Latombe J.C. Robot motion planning: A distributed representation approach. International Journal of Robotics Research, vol. 10, issue 6, 1991. pp. 628 - 649
[42]. Kavraki L.E., Svestka P., Latombe J.C., Overmars M.H. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Trans. Robot. Autom., vol. 12, № 4, 1996, pp. 566–580.
[43]. Hsu D., Latombe J.C., Motwani R. Path Planning in Expansive Configuration Spaces. Proceedings 1997 IEEE International Conference on Robotics and Automation, vol. 3, 1997, pp. 2719–2726.
[44]. LaValle S.M., Kuffner J.J. Rapidly-exploring random trees: Progress and prospects. 2000 Workshop on the Algorithmic Foundations of Robotics, 2000, pp. 293–308.
[45]. Ferre E., Laumond J.-P. An iterative diffusion algorithm for part disassembly. Proceedings 2004 IEEE International Conference on Robotics and Automation, vol. 3, 2004, pp. 3149–3154.
[46]. Bayazit O.B., Lien J.-M., Amato N.M. Probabilistic Roadmap Motion Planning for Deformable Objects. Proceedings 2002 IEEE International Conference on Robotics and Automation, vol. 2, 2002, pp. 2126–2133.

[47]. Guibas L.J., Holleman C., Kavraki L.E. A Probabilistic Roadmap Planner for Flexible Objects with aWorkspace Medial-Axis-Based Sampling Approach. IEEE/RSJ Int. Conf. Intelligent Robot. Syst., 1999, pp. 254–260.
[48]. Berenson D., Srinivasa S., Kuffner J.J. Task Space Regions: A framework for pose-constrained manipulation planning. Int. J. Rob. Res., vol. 30, № 12, 2011, pp. 1435–1460.
[49]. Yakey J.H., LaValle S.M., Kavraki L.E. Randomized path planning for linkages with closed kinematic chains. IEEE Trans. Robot. Autom., vol. 17, № 6, 2001, pp. 951–958.
[50]. LaValle S.M., Kuffner J.J. Randomized Kinodynamic Planning. Int. J. Rob. Res., vol. 20, № 5, 2001, pp. 378–400.
[51]. Niederreiter H. Random number generation and Quasi-Monte Carlo methods. Society for Industrial and Applied Mathematics, 1992, 241 p.
[52]. Дмитриев К.А. От Монте-Карло к Квази Монте-Карло. Труды 12-ой международной конференции по компьютерной графике и машинному зрению ГрафиКон'2002, 2002, стр. 53-58
[53]. Соболь И.М. Многомерные квадратурные формулы и функции Хаара. М.: Главная редакция физико-математической литературы изд-ва «Наука», 1969, 288 стр.
[54]. Соболь И. М. Численные методы Монте-Карло. М.: Главная редакция физико-математической литературы изд-ва «Наука», 1973, 312 стр.
[55]. Khaksar W., Hong T.S., Khaksar M., Motlagh O. A low dispersion probabilistic roadmaps (LD-PRM) algorithm for fast and efficient sampling-based motion planning. Int. J. Adv. Robot. Syst.vol. 10, № 397, 2013, pp. 1-10.
[56]. LaValle S.M., Branicky M.S. On the Relationship Between Classical Grid Search and Probabilistic Roadmaps. In Algorithmic Foundations of Robotics V. Springer Tracts in Advanced Robotics, vol. 7, 2004, pp. 59-75
[57]. Zhang L., Kim Y.J., Manocha D. C-DIST: efficient distance computation for rigid and articulated models in configuration space. Proc. 2007 ACM Symp. Solid Phys. Model. - SPM ’07, 2007, pp. 159-169.
[58]. Lin M.C. Efficient Collision Detection for Animation and Robotics. Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, USA, 1993, 159 p.
[59]. Christer Ericson. Real-time Collision Detection. The Morgan Kaufmann Series in Interactive 3-D Technology. CRC Press, 2004, 632 p.
[60]. Andersen K. A., Bay C. A survey of algorithms for construction of optimal Heterogeneous Bounding Volume Hierarchies. Technical Report. Copenhague, Denmark: Department of Computer Science, University of Copenhagen, 2006, 32 p.
[61]. Золотов В.А., Семенов В.А. Современные методы поиска и индексации многомерных данных в приложениях моделирования больших динамических сцен. Труды ИСП РАН, том 24, 2013, стр. 381-416. DOI: 10.15514/ISPRAS-2013-24-17.
[62]. Золотов В.А., Семенов В.А. Перспективные схемы пространственно-временной индексации для визуального моделирования масштабных индустриальных проектов. Труды ИСП РАН, том 26, вып. 2, 2014, стр. 197-230. DOI: 10.15514/ISPRAS-2014-26(2)-9.
[63]. Золотов В.А., Петрищев К.С., Семенов В.А. Исследование методов пространственного индексирования динамических сцен на основе регулярных октодеревьев. Труды 25-й международной конференции GraphiCon2015, 2015, pp. 115-122.

[64]. Pungotra H. Collision Detection and Merging of Deformable B-spline Surfaces in Virtual Reality Environment. Ph. D. thesis, The Univeristy of Western Ontario, Canada, 2010, 180 p.
[65]. Sandqvist J., Collision detection using boundary representation, BREP. Master thesis, Umeå University, Sweden, 2015, 54 p.
[66]. Su C.J., Lin F., Ye L. New collision detection method for CSG-represented objects in virtual manufacturing. Computers in Industry, vol. 40, № 1, 1999, pp. 1–13.
[67]. Klein J., Zachmann G. Point cloud collision detection. Proc. Eurographics 2004, 2004, pp. 567–576.
[68]. Schwarzer F., Saha M., Latombe J.C. Exact Collision Checking of Robot Paths. In Algorithmic Foundations of Robotics V. Springer Tracts in Advanced Robotics, vol. 7, 2004, pp. 25–41.
[69]. Yershova A., LaValle S.M. Improving Motion Planning Alorithms by Efficient Nearest-Neighbor Searching. IEEE Transactions on Robotics, vol. 23, issue 1, 2007, pp. 151–157.
[70]. Yershova A., LaValle S.M. Planning for closed chains without inverse kinematics. Department of Computer Science, University of Illinois, Urbana, USA, 2007, 7 p. Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.7831&rep=rep1&type=pdf.
[71]. Brown R.A. Building a Balanced k-d Tree in O (kn log n) Time. J. Comput. Graph. Tech, vol. 4, № 1, 2015, pp. 50–68.
[72]. Ciaccia P., Patella M., Rabitti F., Zezula P. Indexing Metric Spaces with M-Tree. Proc. Atti del Quinto Convegno Nazionale SEBD, 1997, pp. 67–86.
[73]. Gipson B., Moll M., Kavraki L.E. Resolution Independent Density Estimation for motion planning in high-dimensional spaces. Proc. 2003 IEEE International Conference on Robotics and Automation, 2013, pp. 2437–2443.
[74]. Fredriksson K. Geometric Near-neighbor Access Tree (GNAT) revisited. arXiv:1605.05944v2, 2016, 7 p.
[75]. Yu C., Ooi B.C., Tan K.-L. Indexing the Distance: An Efficient Method to KNN Processing. Proceedings of the 27th International Conference on Very Large Data Bases, 2001, pp. 421-430.
[76]. Beygelzimer A., Kakade S., Langford J. Cover trees for nearest neighbor. ICML '06, Proc. 23rd Int. Conf. Mach. Learn., 2006, pp. 97–104.
[77]. Ichnowski J., Alterovitz R. Fast Nearest Neighbor Search in SE (3) for Sampling-Based Motion Planning. Algorithmic Foundations of Robotics XI, Volume 107 of the series Springer Tracts in Advanced Robotics, 2015, pp 197-214.
[78]. Amato N.M., Bayazit O.B., Dale L.K., Jones C., Vallejo D. Choosing good distance metrics and local planners for probabilistic roadmap methods. IEEE Trans. Robot. Autom., vol. 16, № 4, 2000, pp. 442–447.
[79]. Nissoux C., Simeon T., Laumond J.-P. Visibility based probabilistic roadmaps. 1999 IEEE/RSJ Int. Conf. Intell. Robot. Syst., vol. 3, 1999, pp. 1316–1321.
[80]. Amato N.M., Wu Y. A randomized roadmap method for path and manipulation planning. Proc. 1996 IEEE International Conference on Robotics and Automation, vol. 1, 1996, pp. 113–120.
[81]. Amato N.M., Bayazit O.B., Dale L.K., Jones C., Vallejo D. OBPRM: An Obstacle-Based PRM for 3D Workspaces. Proceeding WAFR '98 Proceedings of the third workshop on the algorithmic foundations of robotics on Robotics : the algorithmic perspective, 1998, pp. 155–168.

[82]. Yeh H.Y., Thomas S., Eppstein D., Amato N.M. UOBPRM: A uniformly distributed obstacle-based PRM. IEEE Int. Conf. Intell. Robot. Syst., 2012, pp. 2655–2662.
[83]. Hsu D., Jiang T., Reif J., Sun Z. The bridge test for sampling narrow passages with probabilistic roadmap planners. Proc. 2003 IEEE International Conference on Robotics and Automation, vol. 3, 2003, pp. 4420–4426.
[84]. Boor V., Overmars M.H., Stappen a. F. Van Der. The Gaussian sampling strategy for probabilistic roadmap planners. Proc. 1999 IEEE International Conference on Robotics and Automation, vol 2, 1999, pp. 1018–1023.
[85]. Lien J.-M., Thomas S., Amato N.M. A general framework for sampling on the medial axis of the free space. Proc. 2003 IEEE International Conference on Robotics and Automation, vol. 3, 2003, pp. 4439–4444.
[86]. Wilmarth S. a., Amato N.M., Stiller P.F. MAPRM: a probabilistic roadmap planner with sampling on the medial\naxis of the free space. Proc. 1999 IEEE International Conference on Robotics and Automation, vol 2, 1999, pp. 1024–1031.
[87]. Holleman C., Kavraki L.E. A framework for using the workspace medial axis in PRM planners. Proc. 2000 IEEE International Conference on Robotics and Automation, vol 2, 2000, pp. 1408–1413.
[88]. Nielsen C.L.L., Kavraki L.E. A two level fuzzy PRM for manipulation planning. 2000 IEEE/RSJ Int. Conf. Intell. Robot. Syst., vol. 3, 2000, pp. 1716–1721.
[89]. Bohlin R., Kavraki L.E. Path planning using lazy PRM. Proc. 2000 ICRA Millenn. Proc. 2000 IEEE International Conference on Robotics and Automation, vol 1, 2000, pp. 521–528.
[90]. Bohlin R., Kavraki L.E. A Randomized Approach to Robot Path Planning Based on Lazy Evaluation. In Handbook of Randomized Computing, volume I/II, Springer, 2001, pp. 221–253.
[91]. Leven P., Seth H. Toward Real-Time Path Planning in Changing Environments. Proceedings of the Fourth International Workshop on the Algorithmic Foundations of Robotics, 2000, pp. 363-376.
[92]. Nieuwenhuisen D., Van Den Berg J., Overmars M.H. Efficient path planning in changing environments. 2007 IEEE/RSJ Int. Conf. Intell. Robot. Syst., 2007, pp. 3295–3301.
[93]. Jaillet L., Simeon T. A PRM-based motion planner for dynamically changing environments. 2004 IEEE/RSJ Int. Conf. Intell. Robot. Syst., vol. 2, 2004, pp. 1606–1611.
[94]. Bessiere P., Ahuactzin J.M., Talbi E.-G., Mazer E. The Ariadne's Clew Algorithm: Global Planning With Local Methods. Proceeding of the Workshop on Algorithmic Foundations of Robotics, 1995, pp. 39-49.
[95]. Mazer E., Ahuactzin J.M., Bessiere P. The Ariadne ' s Clew Algorithm. Journal of Artifical Intelligence Research, vol. 9, 1998, pp. 295–316.
[96]. Carpin S., Pillonetto G. Motion planning using adaptive random walks. IEEE Trans. Robot., vol. 21, № 1, 2005, pp. 543–548.
[97]. Carpin S., Pillonetto G. Merging the adaptive random walks planner with the randomized potential field planner. Proc. Fifth Int. Work. Robot Motion Control, 2005, pp. 151–156.
[98]. Sanchez G., Latombe J.C. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In Robotics Research, Volume 6 of the series Springer Tracts in Advanced Robotics, Springer, 2003, pp 403-417.
[99]. Hsu D. Randomized Single-query Motion Planning in Expansive Spaces. Ph. D. thesis, Stanford University, USA, 2000, 118 p.
[100]. Sagan H. Space-Filling Curves. Springer-Verlag, New York, 1994, 193 p.

[101]. Kuffner J.J., LaValle S.M. RRT-connect: An efficient approach to single-query path planning. Proc. 2000 IEEE International Conference on Robotics and Automation, vol 2, 2000, pp. 995–1001.
[102]. Bekris K.E., Chen B.Y., Ladd A.M., Plaku E., Kavraki L.E. Multiple Query Motion Planning using Single Query Primitives. IEEE/RSJ Int. Conf. Intell. Robot. Syst., 2003, pp. 656–661.
[103]. Plaku E., Kavraki L.E. Distributed Sampling-Based Roadmap of Trees for Large-Scale Motion Planning. Proc. 2005 IEEE International Conference on Robotics and Automation, 2005, pp. 3879–3884.
[104]. Strandberg M. Robot Path Planning : An Object-Oriented Approach. Ph.D.thesis, Automatic and Control Departmentof, Signals, Sensors and Systems Royal Institute of Technology (KTH), Stockholm, Sweden, 2004, 244 p.
[105]. Bruce J.R., Veloso M. Real-time multi-robot motion planning with safe dynamics. Multi-Robot Systems. From Swarms to Intelligent Automata, Volume III. Proceedings from the 2005 International Workshop on Multi-Robot Systems, Springer, 2005, pp. 1–12.
[106]. Bruce J.R. Real-time motion planning and safe navigation in dynamic multi-robot environments. PhD. Thesis, Carnegie Mellon University, Pittsburgh, USA, 2006, 204 p.
[107]. Yershova A., Jaillet L., Siméon T., LaValle S.M. Dynamic-Domain RRTs: Efficient Exploration by Controlling the Sampling Domain. Proc. 2005 IEEE International Conference on Robotics and Automation, 2005, pp. 3856–3861.
[108]. Raveh B., Enosh A., Halperin D. A little more, a lot better: Improving path quality by a path-merging algorithm. IEEE Trans. Robot., vol. 27, № 2, 2011, pp. 365–371.
[109]. Urmson C., Simmons R. Approaches for heuristically biasing RRT growth. Proc. 2003 IEEE/RSJ Int. Conf. Intell. Robot. Syst., vol.2, 2003, pp. 1178–1183.
[110]. Jaillet L., Cortes J., Simeon T. Sampling-Based Path Planning on Costmaps Configuration-space. Ieee Trans. Robot., vol. 26, № 4, 2010, pp. 635–646.
[111]. Hauser K. Lazy collision checking in asymptotically-optimal motion planning. Proc. 2015 IEEE International Conference on Robotics and Automation, 2015, pp. 2951–2957.
[112]. Luo J., Hauser K. An Empirical Study of Optimal Motion Planning. IEEE/RSJ Conf. Intell. Robot. Syst., 2014, pp. 1761–1768.
[113]. Marble J.D., Bekris K.E. Asymptotically Near-Optimal is Good Enough for Motion Planning. 15th Int. Symp. Robot. Res., 2011, pp. 419-436.
[114]. Marble J.D., Bekris K.E. Computing spanners of asymptotically optimal probabilistic roadmaps. IEEE/RSJ Int. Conf. Intell. Robot. Syst., 2011, pp. 4292–4298.
[115]. Marble J.D., Bekris K.E. Towards small asymptotically near-optimal roadmaps. Proc. 2012 IEEE International Conference on Robotics and Automation, 2012, pp. 2557–2562.
[116]. Dobson A., Bekris K.E. Improving Sparse Roadmap Spanners. Proc. 2013 IEEE International Conference on Robotics and Automation, 2013, pp. 4106–4111.
[117]. Dobson A., Bekris K.E. Sparse roadmap spanners for asymptotically near-optimal motion planning. Int. J. Rob. Res., vol. 33, № 1, 2014, pp. 18–47.
[118]. Nasir J., Islam F., Malik U., Ayaz Y., Hasan O., Khan M., Muhammad M.S. RRT*-SMART: A rapid convergence implementation of RRT*. Int. J. Adv. Robot. Syst., vol. 10, 2013, pp. 1-12.
[119]. Arslan O., Tsiotras P. Use of relaxation methods in sampling-based algorithms for optimal motion planning. Proc. 2013 IEEE International Conference on Robotics and Automation, 2013, pp. 2421–2428.

[120]. Janson L., Pavone M. Fast Marching Trees : a Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions – Extended Version, arXiv:1306.3532v4, 2013, pp. 1–60.
[121]. Salzman O., Halperin D. Asymptotically near-optimal RRT for fast, high-quality, motion planning. Proc. 2014 IEEE International Conference on Robotics and Automation, 2014, pp. 4680–4685.
[122]. Devaurs D., Simeon T., Cortes J. Optimal Path Planning in Complex Cost Spaces with Sampling-Based Algorithms. IEEE Trans. Autom. Sci. Eng., vol. 13, № 2, 2016, pp. 415–424.
[123]. Klemm S., Oberlander J., Hermann A., Roennau A., Schamm T., Zollner J.M., Dillmann R. RRT∗-Connect: Faster, asymptotically optimal motion planning. 2015 IEEE Int. Conf. Robot. Biomimetics, 2015, pp. 1670–1677.
[124]. Starek J.A., Gomez J. V, Schmerling E., Janson L., Moreno Luis, Pavone M. An Asymptotically-Optimal Sampling-Based Algorithm for Bi-directional Motion Planning. IEEE/RSJ Int. Conf. Intell. Robot. Syst., 2015, pp. 2072 – 2078.

An overview of modern methods for motion planning

[1]K.A. Kazakov <kazakov@ispras.ru>
[1,2]V.A. Semenov <sem@ispras.ru>
[1]Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2]Moscow Institute of Physics and Technology (State University),
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia

Abstract. Currently there is growing interest in motion planning problems that play a key role in automation of technologically difficult processes in mechanical, power and transport engineering, medicine, construction and creation of new products and services. This interest particularly relates to the increasing role of computer simulation and such disciplines as complex planning of industrial projects, realistic 3D animation, robotic surgery, automotive products assembly and organization of movement of transport streams. This paper is devoted to the overview and analysis of modern motion planning methods.

Keywords: motion planning, path planning, roadmaps, collision detection.

DOI: 10.15514/ISPRAS-2016-28(4)-14

For citation: Kazakov K.A., Semenov V.A. An overview of modern methods for motion planning. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016, pp. 241-294 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-14

References

[1]. Choset H., Lynch K., Seth H., Kantor G., Burgard W., Kavraki L.E., Thrun S. Principles of Robot Motion-Theory, Algorithms , and Implementation. MIT Press, 2005.

[2]. Geraerts R., Overmars M.H. Creating High-quality Paths for Motion Planning. Int. J. Rob. Res., vol. 26, № 8, 2007, pp. 845–863.
[3]. Karaman S., Frazzoli E. Sampling-based algorithms for optimal motion planning. Int. J. Robot., vol. 30, № 7, 2011, pp. 846–894
[4]. Laumond J.-P. Kineo CAM: A success story of motion planning algorithms. IEEE Robot. Autom. Mag., vol. 13, № 2, 2006, pp. 90–93.
[5]. Rockel S., Klimentjew D., Zhang L., Zhang J. An hyperreality imagination based reasoning and evaluation system (HIRES). Proc. IEEE Int. Conf. on Robotics and Automation, 2014. pp. 5705–5711.
[6]. Sucan I., Moll M., Kavraki L.E. The Open Motion Planning Library. IEEE Robot. Autom. Mag. vol. 19, № 4, 2012, pp. 72–82.
[7]. Diankov R. Automated Construction of Robotic Manipulation Programs. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010, 263 p.
[8]. Porta J.M., Ros L., Bohigas O., Manubens M., Rosales C., Jaillet L. The CUIK Suite: Motion Analysis of Closed-chin Multibody Systems. IEEE Robot. Autom. Mag., vol. 21, № 3, 2014, pp. 105–114.
[9]. Semenov V.A., Kazakov K.A., Zolotov V.A. Effective spatial reasoning in complex 4D modeling environments. eWork Ebus. Archit. Eng. Constr. eds. A.Mahdavi, B. Martens, R. Scherer, CRC Press. Taylor Fr. Group, London, UK. 2015, pp. 181–186.
[10]. Kazakov K.A., Semenov V.A., Zolotov V.A. Topological Mapping Complex 3D Environments Using Occupancy Octrees. 21st Int. Conf. Comput. Graph. Vision, Sept. 26-30, 2011, Moscow, Russ. 2011, pp. 111–114.
[11]. Semenov V.A., Kazakov K.A., Morozov S. V., Tarlapan O.A., Zolotov V.A., Dengenis T. 4D modeling of large industrial projects using spatio-temporal decomposition. eWork Ebus. Archit. Eng. Constr. eds. K. Menzel R. Scherer, CRC Press. Taylor Fr. Group,London, UK. 2010, pp. 89–95.
[12]. Brooks R.A., Lozano-Peres T. A Subdivision Algorithm in Configuration Space For Find Path with Rotation. IEEE Trans. Syst. Man. Cybern., vol. SMC-15, № 2, 1985, pp. 224–233.
[13]. D. Zhu and J.-C. Latombe. Constraint reformulation in a hierarchical path planner. In Proc. IEEE Int. Conf. on Robotics and Automation, 1990, pp. 1918-1923.
[14]. Hornung A., Wurm K.M., Bennewitz M., Stachniss C., Burgard W. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. Auton. Robots., vol. 34, № 3, 2013, pp. 189–206.
[15]. Semenov V.A., Kazakov K.A., Zolotov V.A. Global path planning in 4D environments using topological mapping. eWork Ebus. Archit. Eng. Constr. 2012, pp. 263–269.
[16]. Chen D.Z., Szczerba R.J., Uhran Jr J.J. Using Framed-Quadtrees to Find Conditional Shortest Paths in an Unknown 2-D Environment. Technical Report #95-2, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, Indiana, Jan. 1995, 33 p.
[17]. Chazelle B. Approximation and Decomposition of Shapes. Advances in Robotics 1: Algorithmic and Geometric Aspects of Robotics, 1987, pp. 145–185.
[18]. De Berg M., Cheong O., Van Kreveld M., Overmars M. Computational Geometry: Algorithms and Applications. 3rd Edition. Springer-Verlag, 2008, 386 p.
[19]. LaValle S.M. Planning Algorithms. Cambridge University Press, 2006, 844 p.
[20]. Canny J.F., Lin M.C. An opportunistic global path planner. Algorithmica, vol. 10, № 2-4, 1993, pp. 102–120.

[21]. Huang H.-P.H.H.-P., Chung S.-Y.C.S.-Y. Dynamic visibility graph for path planning. Proc. Int. Conf. Intell. Robot. Syst., vol. 3, 2004, pp. 2813–2818.
[22]. Kaluder H., Brezak M., Petrovic I. A visibility graph based method for path planning in dynamic environments. MIPRO, 2011, Proceedings of the 34th International Convention, Opatija, Croatia, 2011, pp. 717–721.
[23]. Aurenhammer F. Voronoi Diagrams – A Survey of a Fundamental Data Structure. ACM Comput. Surv., vol. 23, № 3, 1991, pp. 345–405.
[24]. Choset H. Sensor based motion planning: The hierarchical generalized voronoi graph. Ph.D. Thesis, California Institute of Technology, 1996, 201 p.
[25]. Russell S.J., Norvig P. Artificial Intelligence: A Modern Approach. Neurocomputing, vol. 9, № 2, 1995, pp. 215-218.
[26]. Bell S., An Overview of Optimal Graph Search Algorithms for Robot Path Planning in Dynamic or Uncertain Environments. Oklahoma Christian University, 2010, 9 p.
[27]. De Filippis L., Guglieri G. Advanced graph search algorithms for path planning of flight vehicles. Recent Adv. Aircr. Technol., 2012, pp. 159–192.
[28]. Zeng W., Church R.L. Finding shortest paths on real road networks: the case for A*. Int. J. Geogr. Inf. Sci., vol. 23, № 4, 2009, pp. 531–543.
[29]. Korf R.E. Depth-first iterative-deepening. An optimal admissible tree search. Artif. Intell., vol. 27, № 1, 1985, pp. 97–109.
[30]. Zhou R., Hansen E.A. Memory-Bounded {A*} Graph Search. The Florida AI Research Society Conference - FLAIRS, 2002, pp. 203–209.
[31]. Holte R., Perez M., Zimmer R., MacDonald A. Hierarchical A*: Searching abstraction hierarchies efficiently. Proceeding AAAI'96 Proceedings of the thirteenth national conference on Artificial intelligence – Volume 1, 1996, pp. 530–535.
[32]. Botea A., Muller M., Schaeffer J. Near optimal hierarchical path-finding. Journal of Game Development, vol. 1, issue 1, 2004, pp. 7-28.
[33]. Kring A., Champandard A., Samarin N. DHPA* and SHPA*: Efficient Hierarchical Pathfinding in Dynamic and Static Game Worlds. Proc. Sixth Artificial Intelligence and Interactive Digital Entertainment Conference, 2010, pp. 39–44.
[34]. Harabor D., Grastien A. Online Graph Pruning for Pathfinding On Grid Maps. AAAI Conf. Artif. Intell., 2011, pp. 1114–1119.
[35]. Koenig S., Likhachev M., Furcy D. Lifelong Planning A*. Artif. Intell., vol. 155, № 1-2, 2004, pp. 93–146.
[36]. Stentz A. The Focussed D* Algorithm for Real-Time Replanning. Proceeding IJCAI'95 Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2, 1995, pp. 1652-1659.
[37]. Koenig S., Likhachev M. D* Lite. Proceedings of the AAAI Conference of Artificial Intelligence (AAAI), 2002, pp. 476–483.
[38]. Koenig S., Likhachev M. Fast Replanning for Navigation in Unknown Terrain. IEEE Trans. Robot., vol. 21, issue 3, 2005, pp. 354-363..
[39]. Khatib O. Real time obstacle avoidance for manipulators and mobile robots. International Journal of Robotics and Research, vol. 5, № 1, 1986, pp. 90–98.
[40]. Barraquand J., Latombe J.C. A Monte-Carlo algorithm for path planning with many degrees of freedom. Proceedings 1990 IEEE International Conference on Robotics and Automation, vol.3, 1990, pp. 1712–1717.
[41]. Barraquand J., Latombe J.C. Robot motion planning: A distributed representation approach. International Journal of Robotics Research, vol. 10, issue 6, 1991. pp. 628 - 649

[42]. Kavraki L.E., Svestka P., Latombe J.C., Overmars M.H. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Trans. Robot. Autom., vol. 12, № 4, 1996, pp. 566–580.
[43]. Hsu D., Latombe J.C., Motwani R. Path Planning in Expansive Configuration Spaces. Proceedings 1997 IEEE International Conference on Robotics and Automation, vol. 3, 1997, pp. 2719–2726.
[44]. LaValle S.M., Kuffner J.J. Rapidly-exploring random trees: Progress and prospects. 2000 Workshop on the Algorithmic Foundations of Robotics, 2000, pp. 293–308.
[45]. Ferre E., Laumond J.-P. An iterative diffusion algorithm for part disassembly. Proceedings 2004 IEEE International Conference on Robotics and Automation, vol. 3, 2004, pp. 3149–3154.
[46]. Bayazit O.B., Lien J.-M., Amato N.M. Probabilistic Roadmap Motion Planning for Deformable Objects. Proceedings 2002 IEEE International Conference on Robotics and Automation, vol. 2, 2002, pp. 2126–2133.
[47]. Guibas L.J., Holleman C., Kavraki L.E. A Probabilistic Roadmap Planner for Flexible Objects with aWorkspace Medial-Axis-Based Sampling Approach. IEEE/RSJ Int. Conf. Intelligent Robot. Syst., 1999, pp. 254–260.
[48]. Berenson D., Srinivasa S., Kuffner J.J. Task Space Regions: A framework for pose-constrained manipulation planning. Int. J. Rob. Res., vol. 30, № 12, 2011, pp. 1435–1460.
[49]. Yakey J.H., LaValle S.M., Kavraki L.E. Randomized path planning for linkages with closed kinematic chains. IEEE Trans. Robot. Autom., vol. 17, № 6, 2001, pp. 951–958.
[50]. LaValle S.M., Kuffner J.J. Randomized Kinodynamic Planning. Int. J. Rob. Res., vol. 20, № 5, 2001, pp. 378–400.
[51]. Niederreiter H. Random number generation and Quasi-Monte Carlo methods. Society for Industrial and Applied Mathematics, 1992, 241 p.
[52]. Dmitriev K.A. From Monte Carlo to Monte Carlo Quasi. International Conference GraphiCon2002, 2002, pp. 53-59 (in Russian).
[53]. Sobol' I.M. Multidimensional quadrature formulas and Haar functions. M.: Home edition of Physical and Mathematical literature, Publishing house "Science", 1969, 288 p. (in Russian).
[54]. Sobol' I.M. Numerical Monte Carlo methods. M.: Home edition of Physical and Mathematical literature, Publishing house "Science", 1973, 312 p. (in Russian).
[55]. Khaksar W., Hong T.S., Khaksar M., Motlagh O. A low dispersion probabilistic roadmaps (LD-PRM) algorithm for fast and efficient sampling-based motion planning. Int. J. Adv. Robot. Syst.vol. 10, № 397, 2013, pp. 1-10.
[56]. LaValle S.M., Branicky M.S. On the Relationship Between Classical Grid Search and Probabilistic Roadmaps. In Algorithmic Foundations of Robotics V. Springer Tracts in Advanced Robotics, vol. 7, 2004, pp. 59-75
[57]. Zhang L., Kim Y.J., Manocha D. C-DIST: efficient distance computation for rigid and articulated models in configuration space. Proc. 2007 ACM Symp. Solid Phys. Model. - SPM ’07, 2007, pp. 159-169.
[58]. Lin M.C. Efficient Collision Detection for Animation and Robotics. Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, USA, 1993, 159 p.
[59]. Christer Ericson. Real-time Collision Detection. The Morgan Kaufmann Series in Interactive 3-D Technology. CRC Press, 2004, 632 p.

[60]. Andersen K. A., Bay C. A survey of algorithms for construction of optimal Heterogeneous Bounding Volume Hierarchies. Technical Report. Copenhague, Denmark: Department of Computer Science, University of Copenhagen, 2006, 32 p.
[61]. Zolotov V.A., Semenov V.A. Advanced indexing methods for large multidimensional data in complex dynamic scenes. Trudy ISP RAN/Proc. ISP RAS, vol. 24, 2013, pp. 381-416 (in Russian). DOI: 10.15514/ISPRAS-2013-24-17.
[62]. Zolotov V.A., Semenov V.A. [Effectie spatio-temporal indexing methods for visual modeling of large industrial projects]. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 2, 2014, pp. 175-196 (in Russian). DOI: 10.15514/ISPRAS-2014-26(2)-9.
[63]. Zolotov V.A., S. P.K., Semenov V.A. Octree-based approach to spatial indexing of complex dynamic scenes. International Conference GraphiCon2015, Russia, 2015, pp. 115-122 (in Russian).
[64]. Pungotra H. Collision Detection and Merging of Deformable B-spline Surfaces in Virtual Reality Environment. Ph. D. thesis, The Univeristy of Western Ontario, Canada, 2010, 180 p.
[65]. Sandqvist J., Collision detection using boundary representation, BREP. Master thesis, Umeå University, Sweden, 2015, 54 p.
[66]. Su C.J., Lin F., Ye L. New collision detection method for CSG-represented objects in virtual manufacturing. Computers in Industry, vol. 40, № 1, 1999, pp. 1–13.
[67]. Klein J., Zachmann G. Point cloud collision detection. Proc. Eurographics 2004, 2004, pp. 567–576.
[68]. Schwarzer F., Saha M., Latombe J.C. Exact Collision Checking of Robot Paths. In Algorithmic Foundations of Robotics V. Springer Tracts in Advanced Robotics, vol. 7, 2004, pp. 25–41.
[69]. Yershova A., LaValle S.M. Improving Motion Planning Alorithms by Efficient Nearest-Neighbor Searching. IEEE Transactions on Robotics, vol. 23, issue 1, 2007, pp. 151–157.
[70]. Yershova A., LaValle S.M. Planning for closed chains without inverse kinematics. Department of Computer Science, University of Illinois, Urbana, USA, 2007, 7 p. Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.7831&rep=rep1&type=pdf.
[71]. Brown R.A. Building a Balanced k-d Tree in O (kn log n) Time. J. Comput. Graph. Tech, vol. 4, № 1, 2015, pp. 50–68.
[72]. Ciaccia P., Patella M., Rabitti F., Zezula P. Indexing Metric Spaces with M-Tree. Proc. Atti del Quinto Convegno Nazionale SEBD, 1997, pp. 67–86.
[73]. Gipson B., Moll M., Kavraki L.E. Resolution Independent Density Estimation for motion planning in high-dimensional spaces. Proc. 2003 IEEE International Conference on Robotics and Automation, 2013, pp. 2437–2443.
[74]. Fredriksson K. Geometric Near-neighbor Access Tree (GNAT) revisited. arXiv:1605.05944v2, 2016, 7 p.
[75]. Yu C., Ooi B.C., Tan K.-L. Indexing the Distance: An Efficient Method to KNN Processing. Proceedings of the 27th International Conference on Very Large Data Bases, 2001, pp. 421-430.
[76]. Beygelzimer A., Kakade S., Langford J. Cover trees for nearest neighbor. ICML '06, Proc. 23rd Int. Conf. Mach. Learn., 2006, pp. 97–104.
[77]. Ichnowski J., Alterovitz R. Fast Nearest Neighbor Search in SE (3) for Sampling-Based Motion Planning. Algorithmic Foundations of Robotics XI, Volume 107 of the series Springer Tracts in Advanced Robotics, 2015, pp 197-214.

[78]. Amato N.M., Bayazit O.B., Dale L.K., Jones C., Vallejo D. Choosing good distance metrics and local planners for probabilistic roadmap methods. IEEE Trans. Robot. Autom., vol. 16, № 4, 2000, pp. 442–447.
[79]. Nissoux C., Simeon T., Laumond J.-P. Visibility based probabilistic roadmaps. 1999 IEEE/RSJ Int. Conf. Intell. Robot. Syst., vol. 3, 1999, pp. 1316–1321.
[80]. Amato N.M., Wu Y. A randomized roadmap method for path and manipulation planning. Proc. 1996 IEEE International Conference on Robotics and Automation, vol. 1, 1996, pp. 113–120.
[81]. Amato N.M., Bayazit O.B., Dale L.K., Jones C., Vallejo D. OBPRM: An Obstacle-Based PRM for 3D Workspaces. Proceeding WAFR '98 Proceedings of the third workshop on the algorithmic foundations of robotics on Robotics : the algorithmic perspective, 1998, pp. 155–168.
[82]. Yeh H.Y., Thomas S., Eppstein D., Amato N.M. UOBPRM: A uniformly distributed obstacle-based PRM. IEEE Int. Conf. Intell. Robot. Syst., 2012, pp. 2655–2662.
[83]. Hsu D., Jiang T., Reif J., Sun Z. The bridge test for sampling narrow passages with probabilistic roadmap planners. Proc. 2003 IEEE International Conference on Robotics and Automation, vol. 3, 2003, pp. 4420–4426.
[84]. Boor V., Overmars M.H., Stappen a. F. Van Der. The Gaussian sampling strategy for probabilistic roadmap planners. Proc. 1999 IEEE International Conference on Robotics and Automation, vol 2, 1999, pp. 1018–1023.
[85]. Lien J.-M., Thomas S., Amato N.M. A general framework for sampling on the medial axis of the free space. Proc. 2003 IEEE International Conference on Robotics and Automation, vol. 3, 2003, pp. 4439–4444.
[86]. Wilmarth S. a., Amato N.M., Stiller P.F. MAPRM: a probabilistic roadmap planner with sampling on the medial\naxis of the free space. Proc. 1999 IEEE International Conference on Robotics and Automation, vol 2, 1999, pp. 1024–1031.
[87]. Holleman C., Kavraki L.E. A framework for using the workspace medial axis in PRM planners. Proc. 2000 IEEE International Conference on Robotics and Automation, vol 2, 2000, pp. 1408–1413.
[88]. Nielsen C.L.L., Kavraki L.E. A two level fuzzy PRM for manipulation planning. 2000 IEEE/RSJ Int. Conf. Intell. Robot. Syst., vol. 3, 2000, pp. 1716–1721.
[89]. Bohlin R., Kavraki L.E. Path planning using lazy PRM. Proc. 2000 ICRA Millenn. Proc. 2000 IEEE International Conference on Robotics and Automation, vol 1, 2000, pp. 521–528.
[90]. Bohlin R., Kavraki L.E. A Randomized Approach to Robot Path Planning Based on Lazy Evaluation. In Handbook of Randomized Computing, volume I/II, Springer, 2001, pp. 221–253.
[91]. Leven P., Seth H. Toward Real-Time Path Planning in Changing Environments. Proceedings of the Fourth International Workshop on the Algorithmic Foundations of Robotics, 2000, pp. 363-376.
[92]. Nieuwenhuisen D., Van Den Berg J., Overmars M.H. Efficient path planning in changing environments. 2007 IEEE/RSJ Int. Conf. Intell. Robot. Syst., 2007, pp. 3295–3301.
[93]. Jaillet L., Simeon T. A PRM-based motion planner for dynamically changing environments. 2004 IEEE/RSJ Int. Conf. Intell. Robot. Syst., vol. 2, 2004, pp. 1606–1611.
[94]. Bessiere P., Ahuactzin J.M., Talbi E.-G., Mazer E. The Ariadne's Clew Algorithm: Global Planning With Local Methods. Proceeding of the Workshop on Algorithmic Foundations of Robotics, 1995, pp. 39-49.

[95]. Mazer E., Ahuactzin J.M., Bessiere P. The Ariadne ' s Clew Algorithm. Journal of Artifical Intelligence Research, vol. 9, 1998, pp. 295–316.
[96]. Carpin S., Pillonetto G. Motion planning using adaptive random walks. IEEE Trans. Robot., vol. 21, № 1, 2005, pp. 543–548.
[97]. Carpin S., Pillonetto G. Merging the adaptive random walks planner with the randomized potential field planner. Proc. Fifth Int. Work. Robot Motion Control, 2005, pp. 151–156.
[98]. Sanchez G., Latombe J.C. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In Robotics Research, Volume 6 of the series Springer Tracts in Advanced Robotics, Springer, 2003, pp 403-417.
[99]. Hsu D. Randomized Single-query Motion Planning in Expansive Spaces. Ph. D. thesis, Stanford University, USA, 2000, 118 p.
[100]. Sagan H. Space-Filling Curves. Springer-Verlag, New York, 1994, 193 p.
[101]. Kuffner J.J., LaValle S.M. RRT-connect: An efficient approach to single-query path planning. Proc. 2000 IEEE International Conference on Robotics and Automation, vol 2, 2000, pp. 995–1001.
[102]. Bekris K.E., Chen B.Y., Ladd A.M., Plaku E., Kavraki L.E. Multiple Query Motion Planning using Single Query Primitives. IEEE/RSJ Int. Conf. Intell. Robot. Syst., 2003, pp. 656–661.
[103]. Plaku E., Kavraki L.E. Distributed Sampling-Based Roadmap of Trees for Large-Scale Motion Planning. Proc. 2005 IEEE International Conference on Robotics and Automation, 2005, pp. 3879–3884.
[104]. Strandberg M. Robot Path Planning : An Object-Oriented Approach. Ph.D.thesis, Automatic and Control Departmentof, Signals, Sensors and Systems Royal Institute of Technology (KTH), Stockholm, Sweden, 2004, 244 p.
[105]. Bruce J.R., Veloso M. Real-time multi-robot motion planning with safe dynamics. Multi-Robot Systems. From Swarms to Intelligent Automata, Volume III. Proceedings from the 2005 International Workshop on Multi-Robot Systems, Springer, 2005, pp. 1–12.
[106]. Bruce J.R. Real-time motion planning and safe navigation in dynamic multi-robot environments. PhD. Thesis, Carnegie Mellon University, Pittsburgh, USA, 2006, 204 p.
[107]. Yershova A., Jaillet L., Siméon T., LaValle S.M. Dynamic-Domain RRTs: Efficient Exploration by Controlling the Sampling Domain. Proc. 2005 IEEE International Conference on Robotics and Automation, 2005, pp. 3856–3861.
[108]. Raveh B., Enosh A., Halperin D. A little more, a lot better: Improving path quality by a path-merging algorithm. IEEE Trans. Robot., vol. 27, № 2, 2011, pp. 365–371.
[109]. Urmson C., Simmons R. Approaches for heuristically biasing RRT growth. Proc. 2003 IEEE/RSJ Int. Conf. Intell. Robot. Syst., vol.2, 2003, pp. 1178–1183.
[110]. Jaillet L., Cortes J., Simeon T. Sampling-Based Path Planning on Costmaps Configuration-space. Ieee Trans. Robot., vol. 26, № 4, 2010, pp. 635–646.
[111]. Hauser K. Lazy collision checking in asymptotically-optimal motion planning. Proc. 2015 IEEE International Conference on Robotics and Automation, 2015, pp. 2951–2957.
[112]. Luo J., Hauser K. An Empirical Study of Optimal Motion Planning. IEEE/RSJ Conf. Intell. Robot. Syst., 2014, pp. 1761–1768.
[113]. Marble J.D., Bekris K.E. Asymptotically Near-Optimal is Good Enough for Motion Planning. 15th Int. Symp. Robot. Res., 2011, pp. 419-436.
[114]. Marble J.D., Bekris K.E. Computing spanners of asymptotically optimal probabilistic roadmaps. IEEE/RSJ Int. Conf. Intell. Robot. Syst., 2011, pp. 4292–4298.
[115]. Marble J.D., Bekris K.E. Towards small asymptotically near-optimal roadmaps. Proc. 2012 IEEE International Conference on Robotics and Automation, 2012, pp. 2557–2562.

[116]. Dobson A., Bekris K.E. Improving Sparse Roadmap Spanners. Proc. 2013 IEEE International Conference on Robotics and Automation, 2013, pp. 4106–4111.
[117]. Dobson A., Bekris K.E. Sparse roadmap spanners for asymptotically near-optimal motion planning. Int. J. Rob. Res., vol. 33, № 1, 2014, pp. 18–47.
[118]. Nasir J., Islam F., Malik U., Ayaz Y., Hasan O., Khan M., Muhammad M.S. RRT*-SMART: A rapid convergence implementation of RRT*. Int. J. Adv. Robot. Syst., vol. 10, 2013, pp. 1-12.
[119]. Arslan O., Tsiotras P. Use of relaxation methods in sampling-based algorithms for optimal motion planning. Proc. 2013 IEEE International Conference on Robotics and Automation, 2013, pp. 2421–2428.
[120]. Janson L., Pavone M. Fast Marching Trees : a Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions – Extended Version, arXiv:1306.3532v4, 2013, pp. 1–60.
[121]. Salzman O., Halperin D. Asymptotically near-optimal RRT for fast, high-quality, motion planning. Proc. 2014 IEEE International Conference on Robotics and Automation, 2014, pp. 4680–4685.
[122]. Devaurs D., Simeon T., Cortes J. Optimal Path Planning in Complex Cost Spaces with Sampling-Based Algorithms. IEEE Trans. Autom. Sci. Eng., vol. 13, № 2, 2016, pp. 415–424.
[123]. Klemm S., Oberlander J., Hermann A., Roennau A., Schamm T., Zollner J.M., Dillmann R. RRT∗-Connect: Faster, asymptotically optimal motion planning. 2015 IEEE Int. Conf. Robot. Biomimetics, 2015, pp. 1670–1677.
[124]. Starek J.A., Gomez J. V, Schmerling E., Janson L., Moreno Luis, Pavone M. An Asymptotically-Optimal Sampling-Based Algorithm for Bi-directional Motion Planning. IEEE/RSJ Int. Conf. Intell. Robot. Syst., 2015, pp. 2072 - 2078.