ИСП
Институт Системного Программирования
им. В.П. Иванникова
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 29, выпуск 5

Volume 29, issue 5

Москва 2017

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

Table of Contents

The Study into Cross-Site Request Forgery Attacks within the Framework of Analysis of Software Vulnerabilities

[1] *A.V. Barabanov <ab@cnpo.ru>*
[1] *A.I. Lavrov <mail@cnpo.ru>*
[2] *A.S. Markov <a.markov@bmstu.ru>*
[1] *I.A. Polotnyanschikov <mail@cnpo.ru>*
[2] *V.L. Tsirlov <v.tsirlov@bmstu.ru>*
[1] *NPO Echelon, Elektrozavodskaya street, 24, Moscow, 107023, Russia*
[2] *Bauman MSTU, 2 Baumanskaya street, 5, Moscow,105005 Russia*

Abstract. Nowadays, web applications are one of the most popular types of target of evaluation within the framework of the information security certification. The relevance of the study of web applications vulnerabilities during information security certification is due to the fact that web technologies are actively used while producing modern information systems, including information systems critical from the information security point of view, and on the other hand carrying out basic attacks on such information systems do not require violators of high technical competence, since data on typical vulnerabilities and attacks, including the attacking tools are heavily represented in publicly available sources of information, and the information systems themselves are usually available from public communication networks. The paper presents the results of a study of the security of web applications that are target of evaluation within the framework of certification for information security requirements against cross-site requests forgery attacks. The results of systematization and generalization of information about the cross-site requests forgery attacks and security controls used by web application developers are presented. The results of experimental studies of 10 web applications that have passed certification tests against information security requirements are presented. The results of experimental studies have shown that most developers do not pay enough attention to protection from cross-site request forgery attack - 7 out of 10 web applications tested have been vulnerable to this type of attack. Based on the results of processing the results of experimental studies, the distribution of security controls used in web applications and identified vulnerabilities by programming languages were obtained. Recommendations regarding the protection of web applications against cross-site request forgery attack for developers planning to certify their software are formulated.

Keywords: information security; software security; analysis of vulnerabilities; web-application; CSRF-attack.

DOI: 10.15514/ISPRAS-2017-29(5)-1

For citation: Barabanov A.V., Lavrov A.I., Markov A.S., Polotnyanschikov I.A., Tsirlov V.L The study into cross-site request forgery attacks within the framework of analysis of software vulnerabilities. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 7-18. DOI: 10.15514/ISPRAS-2017-29(5)-1

1. Introduction

Software created with the use of web-technologies is currently one of the main components in automated control system (ACS) design. The designed ACS are, as a rule, multi-user and can be found on public domain networks (for instance, Internet), which increases the risk of their successful attack. Various procedures (such as certification, independent security audit) are currently used to lower probability of successful attack. They are aimed at identifying vulnerabilities in the software used to design ACS [1, 2].

Software vulnerabilities are analyzed during certification for compliance with the requirements to the protection profiles approved by FSTEC of Russia (Federal Service for Technology and Export Control), which clearly includes requirements of AVA_VAN assurance family “Vulnerability analysis”, and during testing for compliance with the requirements of the technical specifications and classic governing documents of FSTEC of Russia. The procedure for vulnerability analysis recommended by FSTEC of Russia consists in the joint use of approaches specified in the Common Methodology for Information Technology Security Evaluation and ISO/IEC TR 20004 [3]. It should be noted that more specific instructions for the test laboratories (for instance, standard penetration tests) have not yet been developed, which makes this procedure non-determined [4].

The experience of analysis into vulnerabilities of web-applications within the framework of the accredited test laboratory showed that Cross-Site Request Forgery attack, hereinafter – CSRF-attack is currently the most successful attack against targets of evaluation. The main attention of the developers of web-applications, as a rule, is concentrated on implementing measures protecting against attacks like SQL-injections or Cross-site scripting. The situation is aggravated by the fact that measures protecting against CSRF-attacks are still being actively studied, and best practices have not been rigidly registered yet [2, 6].

The goal of this work consisted in developing guidelines for the developers of web-applications, who are planning to certify their solutions as to the information security requirements. The work solves the following tasks to achieve the set goal:

a) Classification and summary of information about CSRF-attacks and measures of protection against them;

b) Consolidation of information about vulnerabilities of web-applications identified within the framework of work of the accredited test laboratories.

2. The results of classification and summary of information about CSRF-attacks and relevant security measures

A hacker performing a CSRF-attack makes the web-browser used by the legal user, who has been authenticated in "security measures against " the attacked web-application, send HTTP-request, which is going to be identified by the application as a request received from a legal user, to the web-application.

A possible consequence from a successful CSRF-attack implementation is running of an arbitrary code in the web-application in the name of authenticated user. Thus, the main causes of CSRF-attacks are vulnerabilities in web-applications related to wrong implementation of algorithm of HTTP-request authorization. Success of CSRF-attack is determined by the following factors [7, 8]:

- The browser automatically applies authentication data of the user (for instance, session cookie-files), when sending HTTP-request to the web-application;
- Web-application uses the obtained authentication data to authorize the action required for performance by HTTP-request.

It should be noted that despite difficulties in implementation, there are cases of successful CSRF-attacks of ‘Login’ and ‘Logout’ type on web-applications [1, 9, 10]. The probability of successful ‘stored’ CSRF-attack is higher, because a malicious code is stored on the side of the attacked web-application, and the hacker does not have to make the user (for instance, using methods of social engineering) go to a special resource with a malicious code.

Implementation of the security measures on the client’s side [11-16], represented by plugins/extensions of the browser or additional software (proxy), has significant drawbacks [8] and is currently only of academic interest.

There are suggestion on implementing security measures directly with the browser source code, for instance, using ‘samesite’ properties of the cookie-files, but currently these measures are experimental and are implemented only in certain browsers. Integrated measures (measures implemented jointly by the software code on the client- and the server-sides), as a rule, implement a certain information control policy [6, 17], which contain critical information (for instance, authentication data), between the browser and the web-server. It should be noted that effective implementation of this type of security measures is possible by making changes in the browser source code. Moreover, essential limitations of these security measures are well-known, which does not allow their use as a sole measure of protection.

The most popular security measures against CSRF-attacks are tokens (synchronic tokens or generated using HMAC cryptographic function) that are generated and checked on the web-application side. This security measure is implemented, as a rule, by the web-application itself or the framework. It should be noted that the majority of the most popular frameworks (such as, Ruby on Rails, ASP.NET,

Django) implement this measure, which somewhat decreases the workload for the developer of a certain web-application and reduces the number of errors related to implementation of the security algorithm by the developer of the web-application.

The main distinctive feature of the token-based security measures is in the token storage method:

- Generated token may be stored on the web-application side (it is associated with the user session) and it shall be compared with the token received from the web-browser;
- Generated token may be stored on the web-browser side (for instance, in the cookie); when the web-application receives a request from the web-browser, the web-application compares the values of tokens in the cookie and the HTTP-request body.

It should be noted that this measure of the web-application security is used correctly, if it is designed and implemented in a way that HTTP-requests of GET type do not change the server state, and are used only for request of the necessary information. AJAX-requests may be protected with tokens inserted in HTTP-header, or custom HTTP-headers (during implementation of this security measure the web-application only checks availability of the heading in the received request).

The leading specialists in the web-application security recommend using the defense in depth principle, when implementing security measures. Thus, specialists of OWASP community recommend implementing security of the web-application by combining two types of the security measures –HTTP-headers verification and tokens.

In some cases, the developers use three or more security measures for critical information systems (for instance, online banking systems). For example, it can be a combination of tokens, verification of HTTP-header and security measures that require actions from the end user, who performs a critical operation (entry of one-time code/ password).

3. Methods and results of the study

The study into the security level of the web-application was carried out in the accredited test laboratory of NPO Echelon (study period: January – November 2016). Brief information about the web-applications that participate in the study is represented in Table 1.

Table I. Brief information about the study objects

Software identifier	Programming language	Type of developer	Level of measures for secure software development implementation (maturity level)
Software No. 1	PHP	Russian	2

Software No. 2	Java	Foreign	5
Software No. 3	PHP	Russian	1
Software No. 4	Java	Foreign	5
Software No. 5	C#	Russian	4
Software No. 6	Java	Russian	1
Software No. 7	C#	Russian	1
Software No. 8	PHP	Russian	1
Software No. 9	Ruby	Russian	3
Software No. 10	Ruby	Russian	3

Level of measures for secure software development implementation (maturity level) was assessed by the expert method with account of the scope of measures implemented by the developer of measures from the basic set of measures for developing secure software suggested in the National Standard GOST R 56939-2016 Information Protection. Secure Software Development. General Requirements. [4, 18]: 1 - not one measure is implemented, 2 - less than 20% of measures is implemented, 3 – from 20% to 40% of measures is implemented, 4 - from 40% to 60% of measures is implemented, 5 - from 60% to 80% of measures is implemented, 6 - over 80% of measures is implemented.

Vulnerabilities were analysed using standard tests developed with account of recommendations and CAPEC resource. Below is the general sequence of the performed tests.

1) Analysis of parts of web-applications (pages), which allow changing the state of the web-application (creating/ changing/ deleting user accounts, protected information, other information etc.).

2) Study of the requests to the identified parts of web-applications: transmission of the requests from the web-browser to the web-application with further interception and analysis of the request structure. The expert analyses the intercepted request and defines the type of security measure against CSRF-attack on a specific page.

3) Generating a mock HTTP-request, which is saved as an HTML-file on the local computer and is opened in the web-browser, provided that there is a session authenticated by the target of evaluation (web-application).

4) If the analysis of intercepted request (cl. 2) revealed security measures against CSRF-attacks, the following actions shall be additionally taken:

a) when tokens are used as a security measure:

- analysis of URL for a presence of token in a plain text;
- sending a request without a token;
- sending a request with an altered token;

- sending a request using one token for various user accounts;
- an attempt to guess /select a token;

b) when using verification of the HTTP-headers as a security measure:

- sending a request with altered HTTP Referer (originally a misspelling of «referrer»)/Origin fields;
- sending a request without HTTP Referrer/Origin fields.

The tests were performed using the following software: BurpSuite software, Scaner-VS software. The average time spent on testing of one web-application by one expert of the test laboratory is 8 hours.

The results of the study are specified below.

1) CSRF-attacks were successful in 70% of cases – 7 out of 10 analysed web-applications turned out to be vulnerable.

2) The majority of CSRF-attacks were successful in relation to web-applications developed in Russia. It should be noted that the only CSRF-attack that was successful in relation to the foreign web-application was that of "Logout" type, and the experts of the test laboratory failed to develop an attack vector that implements information security threat. Only one web-application initially did not have any security measures against CSRF-attacks. The other vulnerable web-applications had security measures based on verification of HTTP-headers or token (Figure 1).

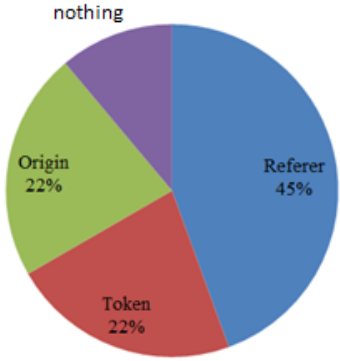


Fig. 1. Distribution of protection measures used in vulnerable web-applications

3) It has been established that web-applications written in PHP have a few more vulnerabilities that results in successful CSRF-attacks (Figure 2) [20].

4) The developers upgraded vulnerable web-applications using security measures based on tokens in all cases.

5) In the majority of cases the upgraded web-application and web-applications, where the vulnerability has not been identified, used a combination of several security measures against CSRF-attacks.

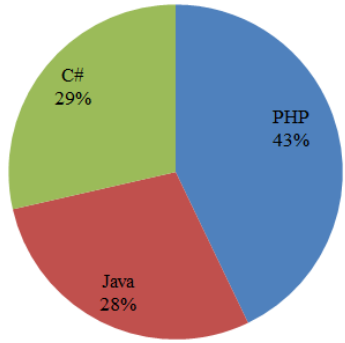


Fig. 2. Distribution of identified vulnerabilities as to the programming language

6) The average time required for the web-application developer to correct vulnerability is 3 weeks.

7) One of the results of the study was a deduced empirical rule, in accordance with which the number of vulnerabilities identified in the software is in inverse proportion to the maturity level of the secure software development processes implemented by the developer.

4. Recommendations to developers on increasing the security level of web-applications

Based on the results of the study the following recommendations were provided for the developers of web-applications that are planning to hold certification tests as to information safety requirements.

1) It is advisable that the developers implement measures for secure software development in the software lifecycle processes. At the very least, it is recommended to implement measures related to testing penetration of web-application prior to their submission to the test laboratory. To minimize time for such testing, the developers should generate sets of standard tests, which may be developed with account of guidelines represented in the works [17, 19]. The developers are advised against limiting their tests to the standard test only, and are recommended to run additional tests aimed at performing CSRF-attacks, like ‘Login’ and ‘Logout’, and verify that the selected security measure is correctly implemented.

2) The developers are recommended using the defense in depth principle – combine two or more security measures (as a rule, verification of token and HTTP-headers), when implementing security measures against CSRF-attacks in the web-application.

3) When implementing security measures against CSRF-attacks in the web-application, the developers are, first of all, recommended to use security measures

that are already implemented in the operational environment, for instance, frameworks.

5. Conclusions

This work consisted in the study into security of web-applications, which are the test targets within the framework of certification as to information security requirements, against cross-site request forgery attacks. The result showed that the majority of the developers (around 70%) do not pay due attention to implementing security measures against such attacks. Resulting from the study, we defined recommendations for the developers, the main of them being recommendations on the use of defense in depth principle and the use of token-based security measures that had already been implemented by the framework developers. We deduced empirical rule, in accordance with which the number of vulnerabilities identified in the software is in inverse proportion to the maturity level of the secure software development processes implemented by the developer. Further studies are intended into the issues of the web-application protection against SQL-injection attacks and cross-site scripting attack and defining general guidelines for the developers of web-applications, who are planning certification.

References

[1]. H. Selim, S. Tayeb, Y. Kim, J. Zhan, and M. Pirouz. Vulnerability Analysis of Iframe Attacks on Websites. In Proceedings of the The 3rd Multidisciplinary International Social Networks Conference on SocialInformatics 2016, Data Science 2016 (MISNC, SI, DS 2016). ACM, New York, NY, USA, Article 45, pp. 1-6, August 2016. DOI: 10.1145/2955129.2955180.

[2]. W. Du, K. Jayaraman, X. Tan, T. Luo, and S. Chapin. Position paper: why are there so many vulnerabilities in web applications? In Proceedings of the 2011 New Security Paradigms Workshop (NSPW '11). ACM, New York, NY, USA, pp. 83-94. 2011. DOI: 10.1145/2073276.2073285.

[3]. A. Barabanov, A. Markov, A. Fadin, V. Tsirlov, I. Shakhalov. Synthesis of Secure Software Development Controls. In Proceedings of the 8th International Conference on Security of Information and Networks (Sochi, Russia, September 8-10, 2015). SIN '15. ACM, New York, NY, USA, pp. 93-97. 2015. DOI: 10.1145/2799979.2799998.

[4]. A.V. Barabanov, A.S. Markov, V.L. Tsirlov. Methodological Framework for Analysis and Synthesis of a Set of Secure Software Development Controls. Journal of Theoretical and Applied Information Technology. 2016. V. 88. No 1, pp. 77-88.

[5]. N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm) , pp. 1–10, September 2006.

[6]. A. Czeskis, A. Moshchuk, T. Kohno, and H.J. Wang. Lightweight server support for browser-based CSRF protection. In Proceedings of the 22nd international conference on World Wide Web (WWW '13). ACM, New York, NY, USA, 2013, pp. 273-284. DOI: 10.1145/2488388.2488413.

[7]. K. Jayaraman, P. G. Talaga, G. Lewandowski, S.J. Chapin, and M. Hafiz. Modeling user interactions for (fun and) profit: preventing request forgery attacks on web applications. In Proceedings of the 16th Conference on Pattern Languages of Programs (PLoP '09). ACM, New York, NY, USA, Article 16, pp. 1-9. August 2009. DOI: 10.1145/1943226.1943246.

[8]. A. Barth, C. Jackson, and J.C. Mitchell. Robust defenses for cross-site request forgery. In Proceedings of the 15th ACM conference on Computer and communications security (CCS '08). ACM, New York, NY, USA, pp. 75-88. October 2008. DOI: 10.1145/1455770.1455782.

[9]. M. Zhou, P. Bisht, and V.N. Venkatakrishnan. Strengthening XSRF defenses for legacy web applications using whitebox analysis and transformation. In Proceedings of the 6th international conference on Information systems security (ICISS'10), pp. 96-110. 2010.

[10]. E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2015), pp. 239-260, June 2015. DOI: 10.1007/978-3-319-20550-2_13

[11]. H. Shahriar and M. Zulkernine. Client-Side Detection of Cross-Site Request Forgery Attacks. In Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10). IEEE Computer Society, Washington, DC, USA, pp. 358-367. November 2010. DOI: 10.1109/ISSRE.2010.12.

[12]. P.D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: transparent client-side mitigation of malicious cross-domain requests. In Proceedings of the Second international conference on Engineering Secure Software and Systems (ESSoS'10), pp. 18-34. 2010. DOI: 10.1007/978-3-642-11747-3_2.

[13]. R. Pelizzi and R. Sekar. A server- and browser-transparent CSRF defense for web 2.0 applications. In Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11). ACM, New York, NY, USA, pp. 257-266. December 2011. DOI: 10.1145/2076732.2076768.

[14]. L. Xing, Y. Zhang, and S. Chen. A client-based and server-enhanced defense mechanism for cross-site request forgery. In Proceedings of the 13th international conference on Recent advances in intrusion detection (RAID'10), pp. 484-485. 2010.

[15]. N. Gelernter and A. Herzberg. Cross-Site Search Attacks. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). ACM, New York, NY, USA, pp. 1394-1405. October 2015. DOI: 10.1145/2810103.2813688.

[16]. E. Z. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp. Toward principled browser security. In Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems (HotOS'13). USENIX Association, Berkeley, CA, USA, pp. 17-17. 2013.

[17]. W. Maes, T. Heyman, L. Desmet, and W. Joosen. Browser protection against cross-site request forgery. In Proceedings of the first ACM workshop on Secure execution of untrusted code (SecuCode '09). ACM, New York, NY, USA, pp. 3-10. November 2009. DOI: 10.1145/1655077.1655081.

[18]. A. Barabanov, A. Markov, V. Tsirlov. Procedure for substantiated development of measures to design secure software for automated process control systems. In Proceedings of the International Siberian Conference on Control and Communications, SIBCON 2016, IEEE, 1-4. June 2016. DOI: 10.1109/SIBCON.2016.7491660.

[19]. X. Li and Y.Xue. A survey on server-side approaches to securing web applications. ACM Comput. Surv., 46, 4, Article 54 (March 2014), 29 pages. April 2014. DOI: 10.1145/2541315

[20]. A.S. Markov, V.L. Tsirlov. Experience in identifying vulnerabilities in foreign software products. Voprosy kiberbezopasnosti [Cybersecurity Issues]. 2013. No 1(1), pp. 42-48. (In Russian).

Исследование атак типа «Cross-Site Request Forgery» в рамках проведения анализа уязвимостей веб-приложений

[1] *А.В. Барабанов < ab@cnpo.ru>*
[1] *А.И. Лавров < mail@cnpo.ru>*
[2] *А.С. Марков < a.markov@bmstu.ru>*
[1] *И.А. Полотнянщиков < mail@cnpo.ru>*
[2] *В.Л. Цирлов <v.tsirlov@bmstu.ru>*
[1] *НПО «Эшелон», 107023, Россия, г. Москва, ул. Электрозаводская, д.24*
[2] *МГТУ им. Н.Э. Баумана,*
105005, Россия, г. Москва, 2-я Бауманская ул., д. 5, стр. 1

Аннотация. Веб-приложения являются одним из наиболее распространенных типов объектов исследования в рамках работы системы сертификации средств защиты информации. Актуальность исследования уязвимостей в веб-приложениях в рамках сертификации по требованиям безопасности информации обусловлена тем, что веб-технологии, с одной стороны, активно используются при реализации современных информационных систем, в том числе критичных с точки зрения информационной безопасности, а, с другой стороны, проведение базовых атак на подобные информационные системы не требуют от нарушителей высокой технической компетентности, поскольку данные о типовых уязвимостях и атаках, включая инструментальные средства проведения атак, в большом объеме представлены в общедоступных источниках информации, а сами информационные системы, как правило, доступны из сетей связи общего пользования. В работе представлены результаты исследования защищённости веб-приложений, являющихся объектами испытаний в рамках сертификации по требованиям безопасности информации, от атак типа «межсайтовая подделка запросов». Приведены результаты систематизации и обобщения сведений об атаке типа «межсайтовая подделка запросов» и мерах защиты, используемых разработчиками веб-приложений. Представлены результаты экспериментальных исследований 10 веб-приложений, которые проходили сертификационные испытания по требованиям безопасности информации. Результаты экспериментальных исследований показали, что большинство разработчиков не уделяют должного внимания защите от межсайтовой подделки запросов – 7 из 10 исследованных веб-приложений оказали уязвимыми к данному типу атаки. По результатам обработки результатов экспериментальных исследований получены распределения мер защиты, используемых в веб-приложениях, и выявленных уязвимостей по языкам программирования. Сформулированы рекомендации в части

защиты веб-приложений от межсайтовой подделки запросов для разработчиков, планирующих проведение сертификации своего программного обеспечения.

Ключевые слова: информационная безопасность; безопасное программное обеспечение; анализ уязвимостей; веб-приложение; межсайтовая подделка запроса.

DOI: 10.15514/ISPRAS-2017-29(5)-1

Для цитирования: Барабанов А.В., Лавров А.И., Марков А.С., Полотнянщиков И.А., Цирлов В.Л. Исследование атак типа «Cross-Site Request Forgery» в рамках проведения анализа уязвимостей веб-приложений. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 7-18 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(5)-1

Список литературы

[1]. H. Selim, S. Tayeb, Y. Kim, J. Zhan, and M. Pirouz. Vulnerability Analysis of Iframe Attacks on Websites. In Proceedings of the The 3rd Multidisciplinary International Social Networks Conference on SocialInformatics 2016, Data Science 2016 (MISNC, SI, DS 2016). ACM, New York, NY, USA, Article 45, pp. 1-6, August 2016. DOI: 10.1145/2955129.2955180.

[2]. W. Du, K. Jayaraman, X. Tan, T. Luo, and S. Chapin. Position paper: why are there so many vulnerabilities in web applications? In Proceedings of the 2011 New Security Paradigms Workshop (NSPW '11). ACM, New York, NY, USA, pp. 83-94. 2011. DOI: 10.1145/2073276.2073285.

[3]. A. Barabanov, A. Markov, A. Fadin, V. Tsirlov, I. Shakhalov. Synthesis of Secure Software Development Controls. In Proceedings of the 8th International Conference on Security of Information and Networks (Sochi, Russia, September 8-10, 2015). SIN '15. ACM, New York, NY, USA, pp. 93-97. 2015. DOI: 10.1145/2799979.2799998.

[4]. A.V. Barabanov, A.S. Markov, V.L. Tsirlov. Methodological Framework for Analysis and Synthesis of a Set of Secure Software Development Controls. Journal of Theoretical and Applied Information Technology. 2016. V. 88. No 1, pp. 77-88.

[5]. N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm) , pp. 1–10, September 2006.

[6]. A. Czeskis, A. Moshchuk, T. Kohno, and H.J. Wang. Lightweight server support for browser-based CSRF protection. In Proceedings of the 22nd international conference on World Wide Web (WWW '13). ACM, New York, NY, USA, 2013, pp. 273-284. DOI: 10.1145/2488388.2488413.

[7]. K. Jayaraman, P. G. Talaga, G. Lewandowski, S.J. Chapin, and M. Hafiz. Modeling user interactions for (fun and) profit: preventing request forgery attacks on web applications. In Proceedings of the 16th Conference on Pattern Languages of Programs (PLoP '09). ACM, New York, NY, USA, Article 16, pp. 1-9. August 2009. DOI: 10.1145/1943226.1943246.

[8]. A. Barth, C. Jackson, and J.C. Mitchell. Robust defenses for cross-site request forgery. In Proceedings of the 15th ACM conference on Computer and communications security (CCS '08). ACM, New York, NY, USA, pp. 75-88. October 2008. DOI: 10.1145/1455770.1455782.

[9]. M. Zhou, P. Bisht, and V.N. Venkatakrishnan. Strengthening XSRF defenses for legacy web applications using whitebox analysis and transformation. In Proceedings of the 6th international conference on Information systems security (ICISS'10), pp. 96-110. 2010.

[10]. E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2015), pp. 239-260, June 2015. DOI: 10.1007/978-3-319-20550-2_13

[11]. H. Shahriar and M. Zulkernine. Client-Side Detection of Cross-Site Request Forgery Attacks. In Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10). IEEE Computer Society, Washington, DC, USA, pp. 358-367. November 2010. DOI: 10.1109/ISSRE.2010.12.

[12]. P.D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: transparent client-side mitigation of malicious cross-domain requests. In Proceedings of the Second international conference on Engineering Secure Software and Systems (ESSoS'10), pp. 18-34. 2010. DOI: 10.1007/978-3-642-11747-3_2.

[13]. R. Pelizzi and R. Sekar. A server- and browser-transparent CSRF defense for web 2.0 applications. In Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11). ACM, New York, NY, USA, pp. 257-266. December 2011. DOI: 10.1145/2076732.2076768.

[14]. L. Xing, Y. Zhang, and S. Chen. A client-based and server-enhanced defense mechanism for cross-site request forgery. In Proceedings of the 13th international conference on Recent advances in intrusion detection (RAID'10), pp. 484-485. 2010.

[15]. N. Gelernter and A. Herzberg. Cross-Site Search Attacks. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). ACM, New York, NY, USA, pp. 1394-1405. October 2015. DOI: 10.1145/2810103.2813688.

[16]. E. Z. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp. Toward principled browser security. In Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems (HotOS'13). USENIX Association, Berkeley, CA, USA, pp. 17-17. 2013.

[17]. W. Maes, T. Heyman, L. Desmet, and W. Joosen. Browser protection against cross-site request forgery. In Proceedings of the first ACM workshop on Secure execution of untrusted code (SecuCode '09). ACM, New York, NY, USA, pp. 3-10. November 2009. DOI: 10.1145/1655077.1655081.

[18]. A. Barabanov, A. Markov, V. Tsirlov. Procedure for substantiated development of measures to design secure software for automated process control systems. In Proceedings of the International Siberian Conference on Control and Communications, SIBCON 2016, IEEE, 1-4. June 2016. DOI: 10.1109/SIBCON.2016.7491660.

[19]. X. Li and Y.Xue. A survey on server-side approaches to securing web applications. ACM Comput. Surv., 46, 4, Article 54 (March 2014), 29 pages. April 2014. DOI: 10.1145/2541315

[20]. Марков. А.С., Цирлов В.Л. Опыт выявления уязвимостей в зарубежных программных продуктах. Вопросы кибербезопасности, 2013, № 1 (1), стр. 42-48

A Modified Scrum Story Points Estimation Method Based on Fuzzy Logic Approach

S. A. Semenkovich <sofya-semenkovich@yandex.ru>
O. I. Kolekonova <okolekonova@gmail.com>
K. Y. Degtiarev <kdegtiarev@hse.ru>
National Research University Higher School of Economics (HSE),
3, Kochnovsky Proezd, Moscow, 125319, Russian Federation

Abstract. Several known methods allow to estimate the overall effort(s) to be used up for the software development. The approach based on story points is preferable and quite common in the context of Scrum agile development methodology. However, it might be rather challenging for people, who are new to this methodology or to a specific Scrum team to estimate the amount of work with story points. The proposed approach involves estimation of features on the basis of linguistic terms that are both habitual and clear for everyone. The presented fuzzy inference system (Mamdani's model) makes it possible to calculate story points using people's opinions expressed as sentences in natural language – the study shows empirically that beginners to Scrum methodology consider the proposed approach to be more convenient and easier in use than the 'plain' story points estimation. Also, four groups of people with different levels of qualification in Scrum were asked to estimate several features of a certain project using the developed approach and common story points approach to prove the relevance of the approach – it was shown that the results of basic story points estimation for Scrum experts differ slightly from the results revealed by proposed approach, while for Scrum beginners such difference is significant. To the opinion of authors, the proposed approach may allow to adapt to Scrum more smoothly, with better understanding of what is implied by story points, grasping the general idea and learning faster their use in practice. The experimental study conducted as a part of the research has shown results approaching the estimations provided by Scrum experts who have been working in real projects and making use of story points for several years. Continuation of the present work can be associated with intensive studies of more complicated methods of aggregation of the experts' opinions, analysis of alternative representation forms of confidence degrees in estimates provided as well as the development of plugin for JIRA tracking system.

Keywords: fuzzy logic; Scrum; story points; expert estimations; aggregation of opinions; fuzzy inference system; Likert scale

DOI: 10.15514/ISPRAS-2017-29(5)-2

For citation: Semenkovich S.A., Kolekonova O.I., Degtiarev K.Y. A Modified Scrum Story Points Estimation Method Based on Fuzzy Logic Approach. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017, pp. 19-38. DOI: 10.15514/ISPRAS-2017-29(5)-2

1. Introduction

Many software systems relate to large-scaled and rather complex products that embrace, in particular, numerous factors to monitor and control at the development stage. Without a doubt, software development is a multifold process that essentially depends on tangled human activities, thus requiring effective management and planning [1]. Software development effort estimation acts as a key constituent of decision-making support during the process of such planning and further management. In short, effort can be defined in the context of combination «man-time» and expressed as the time (number of units) needed for a man (team's member) to complete a given task [1, 2]. Nowadays, we may address a relatively long list of recognized estimation methods aimed at evaluating efforts needed to be spent in the software development process. In fact, many efforts to categorize such methods are originating from the publications by Barry Boehm on software cost modeling and engineering economics in the early eighties of the previous century. We cannot talk about «the best» from all conceivable standpoints classification, but in rough outline such methods can be divided into three aggregative categories, namely: methods based on expert subjective estimates and views (non-model based methods), formal estimation methods that are grounded on specific or generic models, and combined (or, composite) methods built upon joint use of analysis and processing of available from different sources data along with expert estimates [3]. Amongst others, the first category takes in such approaches as planning poker (also known as Scrum poker) and Wideband Delphi, two similar methods where the provided estimations are based on judgments and expressed opinions of project's stakeholders [1]. In formal estimation models (e.g. Constructive Cost Model (COCOMO), COCOMO II as a generalization of COCOMO, weighted micro function points (WMFP), SLIM, use case modeling, story points) formulas and/or results derived from earlier implemented projects are used.

In the present paper, we consider the method of estimation with story points in the context of Scrum, an agile flexible framework to manage the process of software development. The main goal of Scrum is to deliver new software capability (features) every 1-2 weeks (the duration can be extended), each new version includes the most important features for Product Owner, thus allowing to inspect and adapt product to current conditions. The main Scrum characteristic of the estimation process is that Product Owner defines priorities for the features because the product should be maintained in a tested/integrated state every Sprint (i.e. fixed number of days team works together to produce beforehand coordinated changes in the product), so the work should be broken down to pieces/stages [4]. In case of proper compliance with other Agile principles, the release deadline cannot be missed by the team; if the features were evaluated incorrectly by some reasons, skipping over the less important tasks can be the only noticeable disadvantage as compared to waterfall or pseudo-Agile teams' experience.

In contrast with other approaches, Scrum is concerned with two main factors that are important in estimating development efforts. Firstly, the responsibility for the

product falls on the shoulders of the whole team rather than individuals. It means that there are no gradations like «my work» and «your work». The framework attracts attention to cumulative effort(s) per Product Backlog' Item rather than individual effort(s) per feature. Secondly, the tasks are estimated in a relative manner, i.e. they are assessed (compared to each other) in terms of relative units, but not absolute ones. Thus, story points may be employed as such units of measure to express an estimate of the overall effort required to fully implement a product backlog item or any other piece of work [5]. As it is noticed by Joshua Kerievsky [6], "… Many say that story points make us better at estimating because we're estimating the size of work, rather than the time it takes to complete it; … in 2005, one of our customers found story points to be so confusing that he renamed them NUTs (Nebulous Units of Time)".

Such witty testimonial inherently expresses the attitude of newcomers to Scrum development methodology towards story points, their 'fear' of commonly used phrases and statements: «number of points per sprint», or «the estimate in story points is better than estimate in hours», etc. There are many helpful and well-composed electronic and printed sources dedicated to Scrum's set of principles and practices aimed at developing complex systems – books and articles by J. Sutherland, C. Sims, A. Stellman, guides, reports, tutorials on Scrum and other Agile methodologies by AgileRussia.ru, Scrum Alliance®, training courses from ScrumTrek, Scrum.org, LuxSoft, to name a few. Even cursory glance at results of Google search gives cause for being not fully confident indeed in the conception of various word-combinations related to story points enquiries, e.g. «they are cheaper than hours», «relative unit of measure», «estimate of effort», and the like.

In brief, story points are founded on "a short description of a set of features called user stories"; each such story will have a set of story points [7]. When we estimate features with story points, we assign a point value to each item. The raw values we assign are unimportant (we can talk about unit of measure that team's members agreed on), what matters are the relative values. A story that is assigned a value of 2 should require twice as much effort as a story that is assigned a value of 1, and it also constitutes two thirds of a story that is estimated at the level of 3 story points. Because story points represent the effort(s) to develop a story, a team's estimate must cover every aspect that can affect the effort. In general, they bring together as a single whole the amount of work to do, the complexity of the work, any risk or uncertainty in doing the work.

Our research proposes to simplify the process of estimating features with the help of story points. For most people it is rather confusing or even difficult to combine three aforesaid components into one in their mind and give an approximate resultant value. Instead of evaluating the features with story points, we assume that each member of the Scrum team (e.g. expert) provides his/her opinion regarding two factors, namely, these are the amount of work to do and its complexity. Besides, the experts should also specify the level (or, degrees) of their confidence in both such

estimates. Experts operate with preset collection of linguistic terms expressed as words or phrases of the natural language. These verbal units are converted after that to proper fuzzy sets used in further processing. The latter provides application of fuzzy inference system (FIS) for each expert's estimations and aggregation of the results obtained into one outcome. What are the reasons to resort to the help of fuzzy approach? Well, we can partly refer to [8] saying that "many fuzzy categories described linguistically appear to be more informative than precise descriptions".

On top of that, a short survey was also conducted with the aim to figure out the opinions of four different groups of people on proposed approach. The core of this activity is the comparison of story points obtained in "experimental" manner and regular story points estimation.

The rest of the paper is organized as follows: section 2 presents basic definitions, terms (type-1 fuzzy set, linguistic variable, inference system, defuzzification, aggregation of estimates) that are used in the subsequent parts of the paper. The proposed approach to obtain story point-based estimates on the basis of defined input variables of Mamdani's fuzzy inference system (FIS) is discussed and visualized in section 3. The results of conducted experiment (empirical study) with several groups of people having different practical skills relative to use of story points estimations are discussed in the section 4. Concluding remarks are drawn in section 5.

2. Basic definitions and general comments

In clear majority of cases humans express their opinions and judgments using statements of natural language; many things that are thus heard or said are vague to a variable degree. According to Stanford Encyclopedia of Philosophy, a term «is vague to the extent that it has borderline cases», and the latter acquires special significance in relation to the vagueness that has to be modeled in adequate way for the case under consideration. In general, such task appears simple enough only at the first glance, and one of practical approaches, at least, from perception-based point of view, relates to fuzzy logic (FL) methodology. It provides ample means to model the perceived meaning of words/phrases conveying the experts opinions (estimates) in a graded fashion. Following seminal paper "Fuzzy Sets" by L.Zadeh [9], the concept of fuzzy set constitutes a class of objects with continuum membership grades.

Definition 1. Let U be a set of elements (objects) that are denoted generically as x (U={x}); fuzzy set $A \subseteq U$ is a set of ordered pairs $\{(x, \mu_A(x))\}$, where mapping $\mu_A : x \to [0,1]$ is a (type-1) membership function of a fuzzy set A. Value $\mu_A(x)$ is a degree (grade) of membership of x in the set A.

In many situations the shape of membership function can be set by a specialist (expert, domain engineer); such manual tuning of function's parameters turns out to be sufficient at the initial stages of model's development and processing. Thus, piecewise linear functions are often chosen due to their usability, expressive power

in grasping thoroughly both the knowledge and human's perception of situation, as well as computational efficiency.

Definition 2. Trapezoidal membership function [10] is defined by a 4-tuple (a_1, a_2, a_3, a_4) of its parameters in the following way:

$$\mu_A(x) = \begin{cases} 0, \ x \in (-\infty, a_1) \\ (x - a_1)/(a_2 - a_1), \ x \in [a_1, a_2] \\ 1, \ x \in [a_1, a_2] \\ (a_4 - x)/(a_4 - a_3), \ x \in [a_3, a_4] \\ 0, \ x \in (a_4, +\infty) \end{cases} \quad (1)$$

Normalized trapezoidal (and triangular with values $a_2 = a_3$) functions having height $h = \max(\mu_A(x)) = 1, \ \forall x \in U \subset \mathbb{R}^1$, often describe values in the form «close to *b*», «around *b*», where *b* is either a crisp real number $b_{val} \in \mathbb{R}^1$, or the interval $\left[b_{val}^{(1)}, b_{val}^{(2)}\right] \subset \mathbb{R}^1$.

Definition 3. A linguistic variable is characterized by a parameter vector (or, 5-tuple) $\left\langle L_v, T(L_v), U, R_{syn}, R_{sem} \right\rangle$, where parameter L_v is the name of the variable (e.g. $L_v \equiv$"complexity of work"), $T(L_v)$ is the set formed by labels of variable's L_v linguistic values $l_1, .., l_n$ (term-set of L_v; e.g. 'easy', 'normal', 'difficult', etc.). These names are generated using syntactic rule R_{syn}, whereas the meaning $R_{sem}(l_i)$ is associated with each value l_i, $i = \overline{1, n}$, from $T(L_v)$ by means of semantic rule R_{sem}; $R_{sem}(l_i)$ is a fuzzy set (respective membership function) defined on a universe of discourse U. Linguistic modifiers (they are also called hedges) 'very', 'more or less' and the like, together with logical connectives 'and', 'or' and negation 'not' are treated as special type operators that modify the primordial meaning of primary values (terms) $l_1, .., l_n$. It results in altered shape of membership functions representing $l_1^{mod}, .., l_n^{mod}$ [11].

Definition 4. The fuzzy inference is a process of deriving conclusion from given premises and system's inputs (or, given fact), for which compositional rule of inference (CRI) serves as a core. CRI can be viewed as a generalization of modus ponens argument scheme (the mode that affirms). The premises are represented as a set of If-Then rules forming knowledge base Ω, e.g. If x is A_i Then y is B_i, $i = \overline{1, m}$, as a basic case ($A_i \to B_i$, i.e. A_i implies B_i) – potentially, such rules may have more complex appearance. Mamdani-type fuzzy inference system (FIS) proposed and evolved by E.H. Mamdani and S. Assilian in 1975 owing to the examination of

fuzzy logic controller can be expressed as $B'(y) = \bigcup_{x \in U_1} A'(x) \wedge R(x, y)$, where relation $R(x, y)$ is calculated as follows: $R(x, y) = \bigcup_{i=1}^{m} A_i(x) \wedge B_i(y)$, where A_i and B_i are (type-1) fuzzy sets, $A_i \subset U_1$, $B_i \subset U_2$.

The knowledge base Ω represented as a set of If-Then rules constitutes rather convenient and transparent form to express individual expert conceptions of phenomenon under study as well as perceptions of a group of specialists. On the whole, model Ω is a handy tool to discuss hypotheses (under potential tuning up rules and initially set parameters of fuzzy sets, if needed) and to make final decisions.

The process of representing initial data (e.g. linguistic values) as membership functions is called fuzzification; most of applications require to perform at final stages the opposite translation from fuzzy functional forms to crisp values; the latter act as representatives of corresponding fuzzy sets. This is achieved through defuzzification procedures, and one of commonly utilized method is called Center Of Area (COA). It stipulates calculation of the resultant value res^* by way of

$$res^* = \frac{\int_U x \cdot \mu(x) dx}{\int_U \mu(x) dx} \tag{2}$$

The intersection (*AND*) and union (*OR*) operations that are used in computational schemes with fuzzy sets are expressed as functions called t-norms $T(\cdot)$ and s-norms $S(\cdot)$, accordingly [12]. Different types of $T(\cdot)$ and $S(\cdot)$ are presented and discussed at length in the literature (e.g. [13]) – without loss of generality, in the paper we use standard *min* and *max* operators:

$$\mu_{A \cap B}(x) = T(\mu_A(x), \mu_B(x)) = min(\mu_A(x), \mu_B(x)) \tag{3}$$

$$\mu_{A \cup B}(x) = S(\mu_A(x), \mu_B(x)) = max(\mu_A(x), \mu_B(x)) \tag{4}$$

It is worth noting that story points are crisp numbers, because they appear to be the most convenient and easy "units" to compare and interpret by Scrum team members as compared to, for instance, numeric intervals. Thus, crisp numbers are associated with story points, which help to rank features in compliance with efforts required to implement them. As it was mentioned before, the valuable source of information are expert judgments (estimations), and once all such estimations are obtained, they should be aggregated to form conjoint opinion. Such activity can be performed by a dedicated person called *analyst*. With this aim in mind, two methods of aggregation are used in the paper.

The first method of aggregation is applied when all estimations elicited from Scrum team members (experts) are different, with one minimum and one maximum

denoting left and rights extremities in the resultant sequence. For example, if it is of a form 10, 25, 46, 34, 30, 47, 28, simple expression allows to calculate the aggregated estimate:

$$e_{\text{agr}} = \left(\sum_{\text{i} \neq \text{i}_{\min}, \text{i}_{\max}} e_{\text{i}} - e_{\min} - e_{\max} \right) \Big/ \left(\text{n}_e - 2 \right) \quad (5)$$

where e_{agr} is the aggregated estimate, $e_{\min}$ and $e_{\max}$ are minimum and maximum values among obtained estimations, respectively, n_e is the total number of values in the sequence; summation goes over all estimations excluding $e_{\min}$ and $e_{\max}$.

The second aggregation method (weighted arithmetic mean) can be used in situation of appearance of recurring experts' estimations as in the case of values 10, 25, 10, 34, 25, 47, 28; such outcomes (with repetitions) are rather practicable, so they should be addressed reasonably enough. If R_e is the most recurring estimate (conditional mean) observed in the numeric sequence, then e_{agr} can be obtained as follows:

$$e_{\text{agr}} = R_e - \left(\sum_{\text{i}} \left(\left(e_{\text{i}} - R_e \right) \cdot f_{\text{i}} \right) \right) \Big/ \text{n}_e \quad (6)$$

where f_{i} is the frequency of e_{i} occurrence in the row of estimations provided.

All prepared comments allow to proceed to approach that may assist people who are new to Scrum methodology (or, they are newcomers to a specific Scrum team) and who do not fully understand how they can estimate the amount of work to do on the base of story points. The central idea of such approach relates to a natural course, i.e. story points seem brittle and a bit confusing – fine, try in that case to estimate how much certain part of work will take making good use of terms you are familiar with. The aforesaid definitions simplify the perception of the following material, and they should not be considered as an extra "difficulty" to tackle on top of Scrum methodology itself; «*such overload is a bit too thick!*» – the reader may exclaim. We think, in no way, as long as all necessary (not very complex) calculations can be done by analysts; in other respects, interviewing and grasping the verbal statements expressing the results (what is said) in pretty understandable form are natural and plain day-to-day human activities.

3. Expert opinions and levels of confidence – modified Likert scale and fuzzy approach

Suppose that through talks and consultations with experts, the analyst collected the opinions (estimations) of several experts on certain feature expressing how much work, reasoning from their understanding and perception, they'll have to do to implement this feature, complexity of the work and their level of confidence about each of these estimations. After fuzzification of verbal data obtained and applying

fuzzy rules, the aggregated result is converted to story points; the latter can be used at subsequent stages in any project management system.

As it was already mentioned earlier, the expert puts his/her opinion concerning complexity, amount of work as well as degree of confidence in estimation expressed in linguistic forms (statements) [14]. For example, the expert may say the following: «*I'm quite sure that this feature will be difficult to implement, besides I must do a large amount of work to implement this feature, however, I'm not very sure about it*». From this sentence, we can pick out the following pairs of linguistic terms, namely: 'difficult' → 'quite sure' and 'large' → 'not very sure'. With such estimations in mind (and their formal representation by way of fuzzy sets), we'll be able to proceed to the construction of corresponding fuzzy rules [15].

Table 1. Parameters of trapezoidal membership functions representing values of term-sets

The amount of work (set T(*A*)**)**	**The complexity of work (set** T(*C*)**)**	**The overall effort (set** T(*E*)**)**
value 'very small' (1,1,5,20)	value 'very easy' (1,1,5,20)	value 'tiny' (1,1,5,20)
value 'small' (5,15,30,40)	value 'easy' (5,15,30,40)	value 'little' (5,15,30,40)
value 'medium' (25,40,60,75)	value 'normal' (25,40,60,75)	value 'average' (25,40,60,75)
value 'large' (60,70,85,95)	value 'difficult' (60,70,85,95)	value 'big' (60,70,85,95)
value 'very large' (80,95,100,100)	value 'very difficult' (80,95,100,100)	value 'huge' (80,95,100,100)

It is commonly advised to use the interval [1,100] to represent story points estimations, so we direct our attention to the same extreme points 1 and 100 to define the universe U to specify fuzzy sets [4]. The amount of work to do, the complexity of the work and the degrees of confidence are considered as system's input variables, whereas the overall (combined) effort is taken as an output variable. Thus, the following linguistic variables $L_v^{(i)}$ denoted as *A*, *C* and *E* and their values (labels of linguistic terms) are considered [11]: $L_v^{(1)}=A \equiv$ "amount of work to do", $L_v^{(2)}=C \equiv$ "complexity of work" (Fig. 1), $L_v^{(3)}=E \equiv$ "the overall (combined) effort" (Fig. 2), where

T(*A*) = { 'very small', 'small', 'medium', 'large', 'very large' },
T(*C*) = { 'very easy', 'easy', 'normal', 'difficult', 'very difficult' },
T(*E*) = { 'tiny', 'little', 'average', 'big', 'huge' }.

Table 2. Correspondence between levels of confidence and their values

Level of confidence (linguistic term)	Value
'not sure at all'	0.05
'almost not sure'	0.15
'not very sure'	0.35
'more or less sure'	0.5
'sure'	0.65
'quite sure'	0.8
'definitely sure'	0.95
'extremely sure'	1

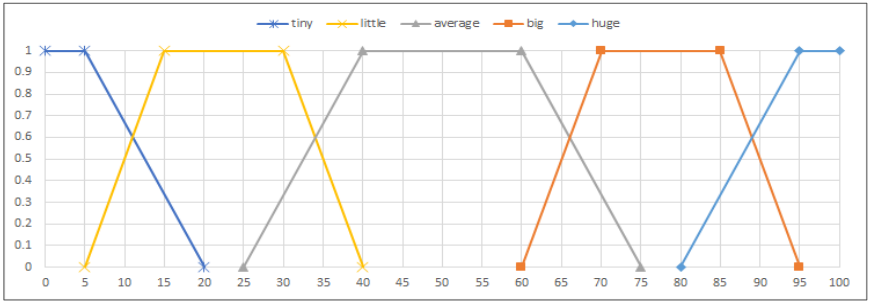


Fig. 1. Linguistic variable C = "complexity of work".

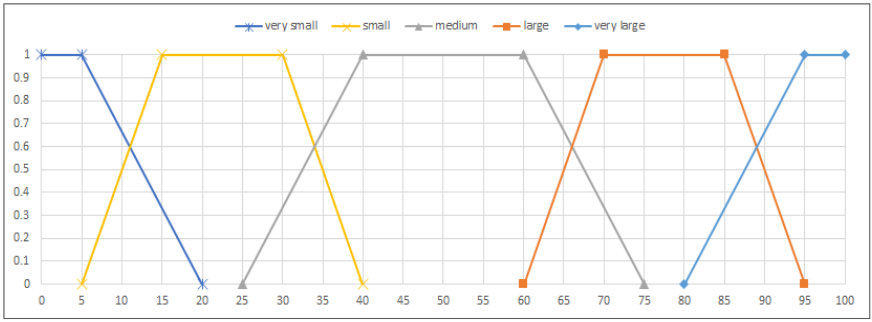


Fig. 2. Linguistic variable E = "the overall (combined) effort".

After consultations with experts, the analyst (and his group) defines the parameters of trapezoidal membership functions (1) to represent formally values of term-sets T(*A*), T(*C*) and T(*E*) fuzzy sets as shown in Table 1. For example, if expert says something like «*... this feature is hard to implement, but I must do small amount of work*», we select primary linguistic values 'small' from the set T(*A*) and 'difficult' – from T(*C*). The parameters of membership functions (Table 1) were chosen

empirically, although slight alterations of values within certain bounds ($\pm\varepsilon_i$, $i = \overline{1,k}$, k is the number of deliberate assortments of such deviations on all terms of sets T(·)) turn out to be allowable. Such "mobility" of value ranges may bring to the advisability to consider further on type-2 interval fuzzy sets – unlike type-1 sets, they enable to express the uncertainty about the membership grades of elements on the domain considered.

Table 3. The accordance of the amount of work and the complexity of work to overall effort.

A \ C	**very easy**	**easy**	**normal**	**difficult**	**very difficult**
very small	tiny	tiny	little	average	average
small	tiny	little	little	average	average
medium	little	little	average	big	big
large	average	average	big	big	huge
very large	average	average	big	huge	huge

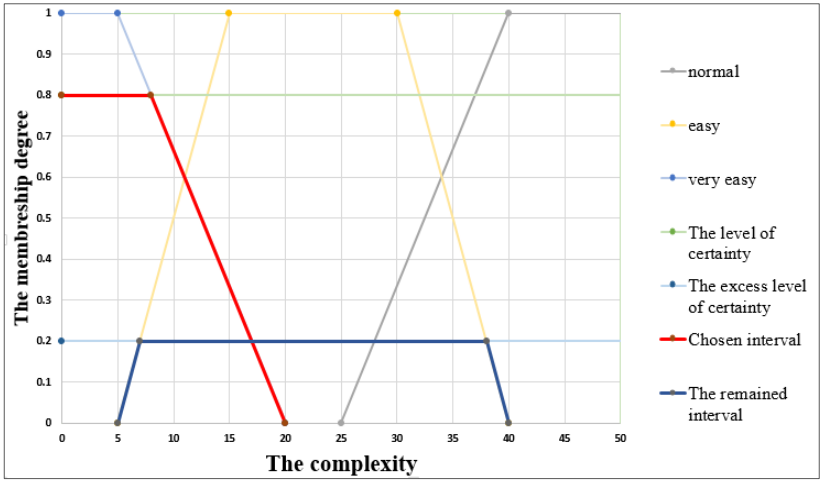


Fig. 3. The distribution of confidence levels (fuzzification stage).

The next step is to relate the level of confidence to fuzzy set being thought about. The ideas and views concerning Likert scale (psychometric response scale suggested by American sociologist Rensis Likert in 1932) allow to come out with relatively simple scheme to use in aforesaid task. Following Qing Li, the level of agreement (LA) as an estimate within the range [0,1] can be associated with the membership degree (as an option, terms 'strongly agree', 'agree', 'neither agree, nor disagree', 'disagree' and 'strongly disagree' can be in use) [16]. The sum of LA for all

options is equal to 1. In the case considered, the option provided by an expert and the level of agreement is experts' levels of confidence are shown in Table 2. However, if expert's level of confidence is not 'extremely sure', we are facing with the excess of LA. Thus, it can be suggested to distribute emergent excess between the nearest neighbors of the option selected by the expert. If there are two nearest neighbors, they both will get half of the excess observed; if there is only one nearest neighbor, it will get the whole amount of excess. In the paper, we use crisp numbers to represent level of confidence' values as the starting point of our approach. These values are based on the results of survey – opinions of approximately 50 people concerning the correspondence between linguistic values (labels) of confidence level and their actual mapped numbers were first elicited and averaged afterwards (see Table 2).

For example, if expert says that his/her level of confidence can be expressed as 'quite sure' (i.e. expert explains that «*... I'm quite sure that…*»), and the feature under consideration is very easy to implement, we choose fuzzy number representing term 'very easy' and define degree of membership as being equal to 0.8 – it is the value of choice. Thus, the excess level of confidence comes to 0.2, and it is handed over to the nearest neighbor of the term 'very easy', which is 'easy'. This distribution of confidence levels is shown graphically in Fig. 3.

Based on the information and knowledge elicited from experts, we may design a set of fuzzy rules (fuzzy rule-base). The amount of work to be done and the complexity of work act as input variables, and their combination result in the value of the overall effort. In general, these rules reflect the perceptions of experts, their feelings and conclusions drawn regarding situation given. For instance, a "typical" question may look as follows: «*How much will it take in the sense of overall effort to accomplish a 'very easy' task that needs just 'medium' amount of work to be done*». The short version of the rule-base is represented in Tab. 3, whereas the full set is provided below (rules Ri, $i = \overline{1,5}$). From the very outset, there were 25 rules (one rule for each combination of the amount of work (*A*) and the complexity of work (*C*)). Later, they were combined on the base of resulting value of overall effort, and only five rules R1,…,R5 were retained.

- rule R1:
IF amount is 'very small' ***AND*** complexity is 'very easy' ***OR***
amount is 'very small' ***AND*** complexity is 'easy' ***OR***
amount is 'small' ***AND*** complexity is 'very easy',
THEN effort is 'tiny'

- rule R2:
IF amount is 'very small' ***AND*** complexity is 'normal' ***OR***
amount is 'small' ***AND*** complexity is 'easy' ***OR***
amount is 'small' ***AND*** complexity is 'normal' ***OR***
amount is 'medium' ***AND*** complexity is 'very easy' ***OR***
amount is 'medium' ***AND*** complexity is 'easy',

THEN effort is 'little'

- rule R3:
IF amount is 'very small' ***AND*** complexity is 'difficult' ***OR***
amount is 'very small' ***AND*** complexity is 'very difficult' ***OR***
amount is 'small' ***AND*** complexity is 'difficult' ***OR***
amount is 'small' ***AND*** complexity is 'very difficult' ***OR***
amount is 'medium' ***AND*** complexity is 'normal' ***OR***
amount is 'large' ***AND*** complexity is 'very easy' ***OR***
amount is 'large' ***AND*** complexity is 'easy' ***OR***
amount is 'very large' ***AND*** complexity is 'very easy' ***OR***
amount is 'very large' ***AND*** complexity is 'easy',
THEN effort is 'average'

- rule R4:
IF amount is 'medium' ***AND*** complexity is 'difficult' ***OR***
amount is 'medium' ***AND*** complexity is 'very difficult' ***OR***
amount is 'large' ***AND*** complexity is 'normal' ***OR***
amount is 'large' ***AND*** complexity is 'difficult' ***OR***
amount is 'very large' ***AND*** complexity is 'normal',
THEN effort is 'big'

- rule R5:
IF amount is 'large' ***AND*** complexity is 'very difficult' ***OR***
amount is 'very large' ***AND*** complexity is 'difficult' ***OR***
amount is 'very large' ***AND*** complexity is 'very difficult',
THEN effort is 'huge'.

Let's consider the following expert's verdict: «*Well, I am quite sure that this {feature} is easy to implement; to tell the truth, I'm also more or less sure that it requires a large amount of work to do*». From this statement, we can extract the following pairs of linguistic terms: 'easy' → 'quite sure' and 'large' → 'more or less sure'. Membership degrees in use are summarized in Tables 4 and 5 (elements of T(*A*) and T(*C*) – five terms in each case):

Table 4. Membership degrees of the complexity C values (example).

The complexity of work (set T(*C*)**)**	**Membership degree**
value 'very easy'	0.1
value 'easy'	0.8
value 'normal'	0.1
value 'difficult'	0
value 'very difficult'	0

Table 5. Membership degrees of the amount of work A values (example).

The amount of work (set T(A))	**Membership degree**
value 'very small'	0
value 'small'	0
value 'medium'	0.25
value 'large'	0.5
value 'very large'	0.25

In this case, fuzzy rules R2, R3 and R4 will give non-zero resultant value. As already stated above, Mamdani inference system (FIS) is used in the experiments – it allows to obtain an output in the form of fuzzy set. Rules R2, R3 and R4 "fire", thus ensuring non-zero results; in compliance with (3) and (4), we arrive at the following:

R2: $max\big(min(0.1,0.25),min(0.8,0.25)\big)=0.25$ – membership degree that corresponds to the term 'little' (element of T(E)),

R3: $max=0.5$ – membership degree that corresponds to the term 'average' (element of T(E)),

R4: $max\big(min(0.1,0.5),min(0.1,0.25)\big)=0.1$ (label of the term 'big' as the element of T(E)).

COA (Center Of Area) method (2) is applied to obtain crisp result. According to equation (2), the output value equals to approx. 45 story points as shown in Fig. 4.

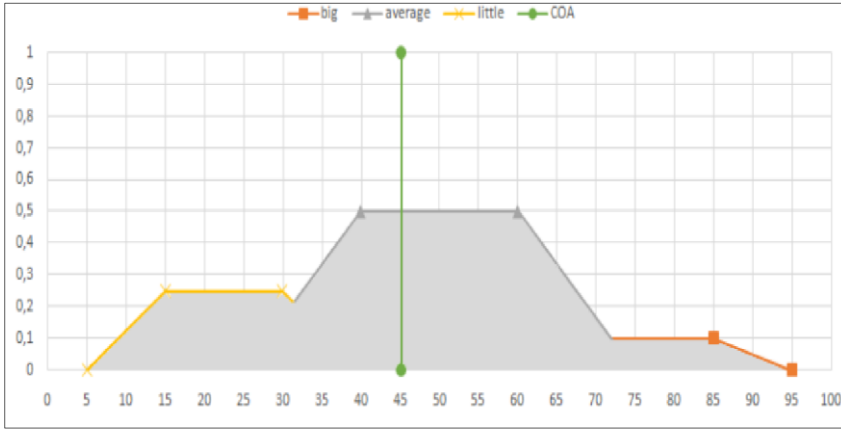


Fig. 4. The result of defuzzification (COA, approx. 45 points).

In Scrum story points estimation' approach the experts often aggregate their opinions using the method of planning poker. It relies on collective judgments (several rounds may become necessary until experts make an agreement) and tries

to avoid "pointless haggling over small differences" by compelling to use estimation value from a set of sharply defined distinct values [17]. All participants (they can also be called estimators) secretly write down their estimations in story points on preprepared cards, and then all cards are laid on the table at one time. If all participants select the same value, this value becomes the feature estimation. If not, each expert one after another explains his/her reasons in showing preference for specific value provided, especially when the choice is fixed upon the highest and the lowest estimators in the set. Afterwards, the process is reiterated, i.e. experts vote again, planning poker goes on. It continues until estimators arrive at the agreement.
As regards the aggregation procedure, two approaches mentioned earlier are used. The exact way to calculate the aggregated opinion based on estimations expressed is chosen according to simple rule: if some of them (estimations) are repeated, the equation (6) is used; otherwise, the equation (5) is preferred.

4. Results of experiment – different groups of potential users. Does the proposed method work?

For the sake of completeness, we have asked several groups of people about their views regarding proposed method (its details were discussed with persons concerned in advance). **Group 1** consisted of those people who have worked with story points for a long time. Those delegates who worked with story points before for relatively short-term period formed **group 2**, while those who know what story points are, but have never used them earlier found themselves in the **3rd group**. Finally, people who never even heard of story points fell into **group 4**. As a result, opinions stated below were emphasized (single form of statements are cited for convenience):

(1) **group 1**: «*... I personally consider story points to be the most effective and quite fast way of evaluating features. I make almost no mistakes in estimating features now, and I can adapt myself in new projects in a short time. Your approach is not useful for me now, though I think it might be helpful at the beginning of (my) career*»,

(2) **group 2**: «*... As for me, it took about two months to fully understood the concept of story points, but even now I sometimes make mistakes while estimating features in terms of story points. Today I believe that estimating in story points is more convenient than estimating in hours or some other units. I'm quite experienced member of the currently ongoing project, and I don't need your approach now, though I could still use it, if I have to get the feel of some new project later on*»,

(3) **group 3**: «*... I have heard that story points exist, and that they are used in project estimation, though I have no experience of participating in real projects, where story points were adopted. I think that your approach is better for me right now than story points in their "pure" appearance as I understand it more clearly as compared to story points per se*»,

(4) **group 4**: «*... Oh, I have no idea what are these "story points" are, so obviously, I better prefer to give my opinion on how much work I will have to do to implement the feature, or how difficult this work seems to me*».

Afterwards, we gave people a description of the project (Android App "VR Quest in city" and its features planned for implementation) was introduced to people who took part in the interview session. They were asked to estimate these features both (A) in terms of "plain" story points and (B) using proposed approach.

Table 6. The results of the conducted experiment.

Feature name		**gr. 1**	**gr. 2**	**gr. 3**	**gr. 4**
Create a login form	story points	30	28	40	45
	our approach	35	37	30	28
Find a quest with specific parameters	story points	27	30	55	60
	our approach	29	24	27	30
Save/load a quest	story points	20	25	35	45
	our approach	18	22	20	19
Begin a quest walkthrough	story points	15	18	25	40
	our approach	16	14	15	17
Buy quests in local currency	story points	50	48	75	85
	our approach	52	55	50	45

As shown in Table 6 and Fig. 5-6 (data obtained for groups 1,4 only are visualized), the results of basic story points estimation for the group 1 (participants in this group know how to estimate features in story points), differ not appreciably from the results revealed by proposed approach. It can be treated as initial piece of empirical evidence of the fact that our method is relevant enough and can be used for feature estimation and further elaboration. Moreover, results in both groups 3 and 4 (members of these groups have never used story points before) are substantially different in case of our method as compared with basic story points estimation' approach. This can be attributed to the marked fact that people do not really understand what story points are in the context of non-using them earlier. This is an extra argument in favor of potential utility of the proposed method for those people who are new to Scrum.

Taking story points estimates as landmarks, the Root Mean Square Error is growing steadily from 2.76 for group 1 to 28.26 for group 4 (for groups 3 and 4 the error values are equal to 6.18 and 19.15, correspondingly). The MAD measure, i.e. the size of deviation in units of landmarks from values calculated with the help of

proposed approach ((A) and (B) estimates, Table 6), is progressing from 2.4 (group 1) to 27.2

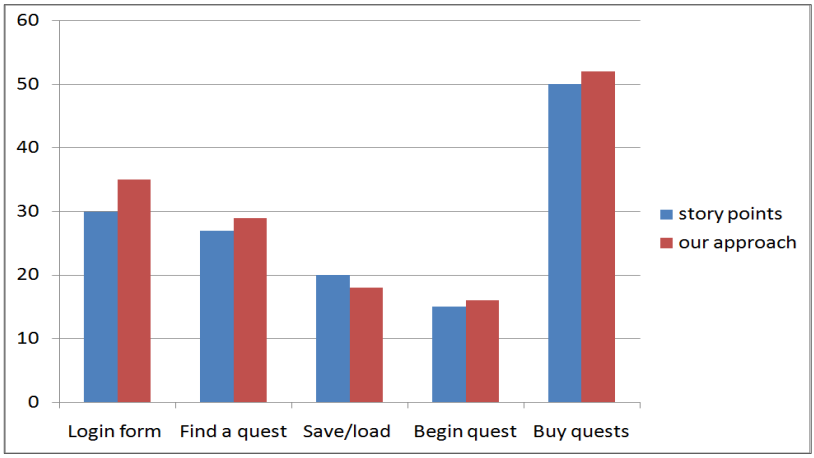


Fig. 5. The results of the conducted experiment as applied to group 1.

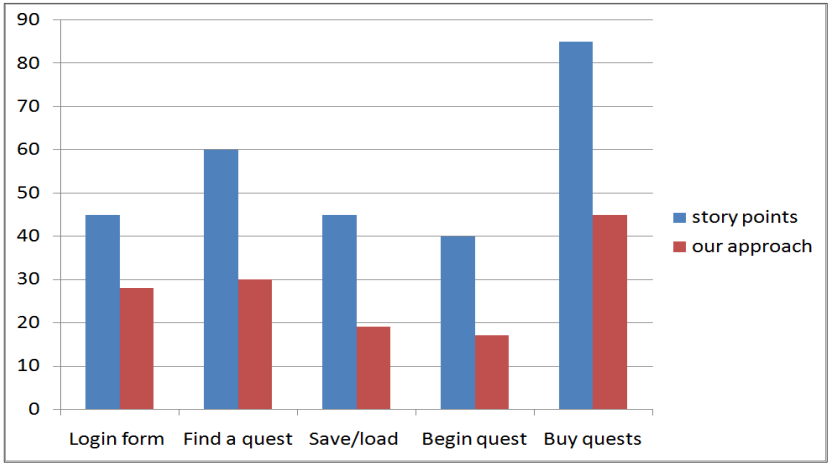


Fig. 6. The results of the conducted experiment as applied to group 4.

(group 4), while values of 5.8 and 17.6 stand for groups 3 and 4, accordingly. These error values show certain tendency of drawing groups 1 and 2 together along with more perceptible isolation (or, distancing) of «groups 3 and 4» bundle from the practical standpoint of both perception and acceptance of story point-based estimation approach. However, even against a background of such observation, group 3 reveals some positive "detachment" toward group 4. In aggregate, we may conclude that the proposed method has rather tangible effect (in decreasing sequence) on group 1, group 2 and group 3 just "touching" the latter in passing. To

the opinion of authors, it can be treated as encouraging sign that is incentive to continue research in this direction.

5. Conclusion

A novel approach that relates to feature estimation in terms of story points was presented in the paper. The natural idea behind the approach reflects the fact that people may estimate their perception (ideas) concerning the *complexity* of implementation of certain product's feature to be and the *amount of work* to be done to develop this feature. Besides, they can also specify the level of their confidence (or, confidence degree) in evaluation provided. Fuzzy inference scheme lays both solid and transparent groundwork for converting aforesaid input information (data) to the number of story points that can be utilized in the software project management (SPM) at a later stage.

To the opinion of authors, this approach allows people to adapt to Scrum more smoothly, with better understanding of what is implied by story points, grasping the general idea and learning faster their use in practice. The experimental study of the proposed method has shown results approaching the estimations provided by Scrum experts who have been working in real projects and making use of story points for several years. According to survey conducted, such approach can be successfully applied by Scrum newbies, since it is more convenient for people who just make up with story points estimations.

It must be noted that full awareness of strong and weak points of the proposed approach reasoning from one example (project) cannot be realized entirely. Therefore, a sequel of empirical studies and active cooperation with Scrum teams may result in enhancement of the approach. One thing is just to mention that the method seems both promising and handy, but it's quite another matter to make it applicable in practice because of convenience and clearness, at least, as a part of induction stage of the "immersion" to Scrum. Transparent and well perceptible ideas of fuzzy logic are very much to the point here.

Further steps can be associated with intensive studies of more complicated methods of aggregation of the experts' opinions – in particular, they may consider the level (or, weight) of professional qualification of domain experts drawn into project activity. Currently a program's prototype to support (implement) the approach discussed in the paper is under development. The present-day agenda also covers the development of plugin for JIRA tracking system. It is also worth mentioning that certain refinements and changes of the proposed approach can be done at the theoretical level either – some of them are visible enough at present. For instance, the confidence degree values can be represented as intervals, i.e. a form of uncertainty/vagueness expression at the lowest level of comprehension. Such intervals may come about as an effect of possible discord concerning the choice of crisp values shown in the Tab. 2. For the time being, these values may be treated as rough aggregated estimates underlying the computational steps of the discussed approach. Besides, the transition from intervals to type-1 fuzzy sets is also an

explicable option to consider. Fuzzy set can be decomposed into a series of nested crisp intervals (so-called α-cuts of a fuzzy set), and this fact can be effectively used in algorithms. Without confining ourselves to just modeling linguistic terms that stand for confidence levels in use, type-2 fuzzy sets and systems are also regarded as "right" candidates for expansion research efforts in a given problem.

References

[1]. Trendowicz A., 2013. Software Cost Estimation, Benchmarking, and Risk Assessment: The Software Decision-Makers' Guide to Predictable Software Development, Springer-Verlag

[2]. Živadinović J., Medić Z., Maksimović D., et al., 2011. Methods of Effort Estimation in Software Engineering. Proc. Int. Symposium Engineering Management and Competitiveness (EMC), 417–422.

[3]. Briand L.C., Wieczorek I. Resource Estimation in Software Engineering. Int. Software Engineering Research Network, TR ISERN 00-05, web-resource: https://pdfs.semanticscholar.org/943d/a2bb363c06319218ee204622bb10f816490f.pdf (access date 24.02.2017)

[4]. Shivangi S., Umesh K., 2016. Review of Various Software Cost Estimation Techniques. International Journal of Computer Applications, vol. 141, 31–34.

[5]. Colomo-Palacios R. González-Carrasco I., et al., 2012. Resyster: A Hybrid Recommender System for Scrum Team Roles based on Fuzzy and Rough Sets. Int. Journal Appl. Math. Comput. Science, 2012, Vol. 22, No. 4, 801–816.

[6]. Industrial Logic site: Stop Using Story Points, Kerievsky J. (blog), 2012, web-resource: https://www.industriallogic.com/blog/stop-using-story-points/ (access date 24.02.2017)

[7]. Pries K.H., Quigley J., 2010. Scrum Project Management, CRC Press

[8]. Aliev R.A., Aliyev R.R., 2001. Soft Computing and Its Applications, World Scientific

[9]. Zadeh L.A., 1965. Fuzzy Sets, Information and Control, #8, 338–353.

[10]. Bingyi K., Daijun W., Li Y., Deng Y., 2012. A Method of Converting Z-Number to Classical Fuzzy Number. Journal of Information & Computational Science, 9, #3, 703–709.

[11]. Zadeh L.A., 1975. The Concept of a Linguistic Variable and Its Application to Approximate Reasoning - I. Information Sciences, vol. 8, no. 3, 199–249.

[12]. Fuzzy Logic Fundamentals, Pearson Education, 2001, Ch.3, 61–99, web-resource: http://ptgmedia.pearsoncmg.com/images/0135705991/samplechapter/0135705991.pdf (access date 21.03.2017)

[13]. Klir G.J., Bo Yuan., 1995. Fuzzy Sets and Fuzzy Logic: Theory and Applications, 1st ed., Prentice Hall

[14]. Zadeh L.A., 1996. Fuzzy logic = Computing with Words. IEEE Trans. Fuzzy Systems, vol. 4, no. 2, 103–111.

[15]. Zadeh L.A., 1992. Fuzzy Logic and the Calculus of Fuzzy If-Then Rules. Proc. 22nd Intl. Symp. on Multiple-Valued Logic, Los Alamitos, CA: IEEE Computer Society Press, 480–480.

[16]. Quing L., 2013. A Novel Likert Scale Based on Fuzzy Sets Theory. Expert Systems with Applications, vol. 40, #5, 1609–1618.

[17]. Meyer B., 2014. Agile! The Good, the Hype and the Ugly, Springer Int.

Семенкович С.А., Колеконова О.И., Дегтярев К.Ю.. Модифицированный метод оценки Story Points в методологии разработки Scrum, основанный на теории нечеткой логики. *Труды ИСП РАН*, том 29, вып. 5, 2017, стр. 19-38

Модифицированный метод оценки Story Points в методологии разработки Scrum, основанный на теории нечеткой логики

С.А. Семенкович <sofya-semenkovich@yandex.ru>
О.И. Колеконова <okolekonova@gmail.com>
К.Ю. Дегтярев <kdegtiarev@hse.ru>
Национальный исследовательский Университет «Высшая Школа Экономики»,
125319, Москва, Кочновский проезд, д. 3, Российская Федерация

Аннотация. Существует несколько известных методов, позволяющих оценить усилия, которые придется потратить на разработку программного обеспечения. В популярной на сегодняшний день методологии гибкой разработки Scrum для этих целей широко используется подход, основанный на story points. Однако, их использование для оценки объема работы может быть затруднительным для тех людей, которые только начинают знакомство с методологией Scrum или впервые попадают в новую Scrum-команду. Описанный в статье подход предлагает использовать оценку трудозатрат на разработку конкретной части программного продукта на основе привычных и понятных для всех фраз естественного языка. Предложенная система нечеткого вывода (модель Мамдани) позволяет преобразовывать мнения людей, выраженные в виде предложений на естественном языке, в число story points – проведенные исследования эмпирически показывают, что те, кто делает первые шаги в методологии Scrum, считают такой подход более удобным и простым, по сравнению с обычным методом оценивания в story points. Также, с целью выяснения, может ли разработанный подход использоваться при работе над реальными проектами, был проведен дополнительный эксперимент, в котором приняли участие четыре группы людей с различными уровнями квалификации в Scrum-разработке. Представителям этих групп было дано задание оценить трудозатраты на разработку отдельных частей некоторого проекта с использованием предложенного подхода и обычных story points единицах. Оценки группы экспертов в области Scrum оказались примерно одинаковы для обоих подходов, в то время как оценки ′новичков′ в методологии сильно отличались при применении двух разных методов. По мнению авторов, предложенный подход может дать возможность более плавного вхождения в методологию Scrum, лучшего понимания природы story points и более быстрой выработке навыков работы с ними на практике. Отдельного внимания заслуживает вопрос изучения разных форм агрегации мнений экспертов, анализ альтернативных подходов к представлению степеней уверенности экспертных оценок и возможная разработка плагина для системы отслеживания ошибок JIRA. Всё это может составить предмет развития данной темы.

Ключевые слова: нечеткая логика; Scrum; story points; экспертные оценки; агрегация мнений; система нечеткого вывода; шкала Лайкерта

DOI: 10.15514/ISPRAS-2017-29(5)-2

Для цитирования: Семенкович С.А., Колеконова О.И., Дегтярев К.Ю. Модифицированный метод оценки Story Points в методологии разработки Scrum, основанный на теории нечеткой логики. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 19-38 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(5)-2

Список литературы

[1]. Trendowicz A., 2013. Software Cost Estimation, Benchmarking, and Risk Assessment: The Software Decision-Makers' Guide to Predictable Software Development, Springer-Verlag

[2]. Živadinović J., Medić Z., Maksimović D., et al., 2011. Methods of Effort Estimation in Software Engineering. Proc. Int. Symposium Engineering Management and Competitiveness (EMC), 417–422.

[3]. Briand L.C., Wieczorek I. Resource Estimation in Software Engineering. Int. Software Engineering Research Network, TR ISERN 00-05, web-resource: https://pdfs.semanticscholar.org/943d/a2bb363c06319218ee204622bb10f816490f.pdf (access date 24.02.2017)

[4]. Shivangi S., Umesh K., 2016. Review of Various Software Cost Estimation Techniques. International Journal of Computer Applications, vol. 141, 31–34.

[5]. Colomo-Palacios R. González-Carrasco I., et al., 2012. Resyster: A Hybrid Recommender System for Scrum Team Roles based on Fuzzy and Rough Sets. Int. Journal Appl. Math. Comput. Science, 2012, Vol. 22, No. 4, 801–816.

[6]. Industrial Logic site: Stop Using Story Points, Kerievsky J. (blog), 2012, web-resource: https://www.industriallogic.com/blog/stop-using-story-points/ (access date 24.02.2017)

[7]. Pries K.H., Quigley J., 2010. Scrum Project Management, CRC Press

[8]. Aliev R.A., Aliyev R.R., 2001. Soft Computing and Its Applications, World Scientific

[9]. Zadeh L.A., 1965. Fuzzy Sets, Information and Control, #8, 338–353.

[10]. Bingyi K., Daijun W., Li Y., Deng Y., 2012. A Method of Converting Z-Number to Classical Fuzzy Number. Journal of Information & Computational Science, 9, #3, 703–709.

[11]. Zadeh L.A., 1975. The Concept of a Linguistic Variable and Its Application to Approximate Reasoning - I. Information Sciences, vol. 8, no. 3, 199–249.

[12]. Fuzzy Logic Fundamentals, Pearson Education, 2001, Ch.3, 61–99, web-resource: http://ptgmedia.pearsoncmg.com/images/0135705991/samplechapter/0135705991.pdf (access date 21.03.2017)

[13]. Klir G.J., Bo Yuan., 1995. Fuzzy Sets and Fuzzy Logic: Theory and Applications, 1st ed., Prentice Hall

[14]. Zadeh L.A., 1996. Fuzzy logic = Computing with Words. IEEE Trans. Fuzzy Systems, vol. 4, no. 2, 103–111.

[15]. Zadeh L.A., 1992. Fuzzy Logic and the Calculus of Fuzzy If-Then Rules. Proc. 22nd Intl. Symp. on Multiple-Valued Logic, Los Alamitos, CA: IEEE Computer Society Press, 480–480.

[16]. Quing L., 2013. A Novel Likert Scale Based on Fuzzy Sets Theory. Expert Systems with Applications, vol. 40, #5, 1609–1618.

[17]. Meyer B., 2014. Agile! The Good, the Hype and the Ugly, Springer Int.

Модифицированные коды с суммированием взвешенных переходов в системах функционального контроля комбинационных схем

[1] *Д.В. Ефанов <TrES-4b@yandex.ru>*
В.В. Сапожников <port.at.pgups@gmail.com>
Вл.В. Сапожников <at.pgups@gmail.com>
[1] *Петербургский государственный университет путей сообщения Императора Александра I,*
190031, Россия, г. Санкт-Петербург, Московский пр., д. 9

Аннотация. Предложен способ построения модифицированных кодов с суммированием взвешенных переходов между разрядами в информационных векторах, занимающими соседние позиции в информационных векторах. Новые коды с суммированием имеют такое же количество контрольных разрядов, как и классические коды Бергера, однако обнаруживают большее количество ошибок в информационных векторах. Модифицированные коды с суммированием взвешенных переходов по сравнению с кодами Бергера также имеют улучшенные характеристики обнаружения ошибок в области малой кратности. Кроме того, для некоторых значений длин информационных векторов могут быть построены коды с обнаружением любых двукратных и любых трехкратных ошибок. Авторами разработан способ синтеза систем функционального контроля комбинационных схем, основанный на анализе топологии объекта диагностирования с выделением групп контролепригодных выходов с учетом свойств обнаружения ошибок модифицированными кодами с суммированием взвешенных переходов. Сформирован алгоритм синтеза системы функционального контроля.

Ключевые слова: система функционального контроля; комбинационная схема; код Бергера; код с суммированием взвешенных переходов; обнаружение двукратных ошибок; обнаружение трехкратных ошибок.

DOI: 10.15514/ISPRAS-2017-29(5)-3

Для цитирования: Сапожников В.В., Сапожников Вл.В., Ефанов Д.В. Модифицированные коды с суммированием взвешенных переходов в системах функционального контроля комбинационных схем. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 39-60. DOI: 10.15514/ISPRAS-2017-29(5)-3

1. Введение

При построении надежных дискретных систем часто используются методы обнаружения отказов в процессе их функционирования, в частности, самопроверяемые схемы встроенного контроля (системы функционального контроля) [1 – 4]. Методы синтеза систем функционального контроля базируются на применении помехоустойчивого кодирования и, непосредственно, кодов, ориентированных на обнаружение ошибок. К таким кодам относятся равномерные блочные коды, включающие в себя большой класс кодов с суммированием (кодов Бергера и их модификаций [5 – 8]), а также неразделимые равновесные коды [9, 10].

Универсальным подходом при синтезе систем функционального контроля является использование разделимых кодов с суммированием, или (m,k)-кодов (m и k – длины информационных и контрольных векторов) [11 – 14]. Свойства (m,k)-кодов по обнаружению ошибок в информационных векторах определяют характеристики обнаружения ошибок на выходах объектов диагностирования в системах функционального контроля. Сложность же функций, описывающих разряды контрольных векторов (m,k)-кодов, напрямую связана со сложностью контрольного оборудования в системе диагностирования.

При синтезе системы диагностирования решается задача наилучшего покрытия неисправностей в объекте диагностирования. Наиболее популярной является классическая модель одиночных константных неисправностей выходов внутренних логических элементов (stuck-at fault) [1 – 4]. В системах функционального контроля реальных дискретных устройств должно быть обеспечено обнаружение любых неисправностей из заданного класса. Для решения этой задачи с учетом наименьших аппаратурных затрат используют свойства обнаружения ошибок (m,k)-кодами, а также различные виды функциональной зависимости между выходами объекта диагностирования [15 – 20].

2. Модифицированные коды с суммированием взвешенных переходов

Разделимые блочные коды для систем функционального контроля строятся по различным правилам, предполагающим операции суммирования как единичных информационных разрядов, так и взвешенных разрядов или переходов между разрядами, занимающими соседние позиции в информационных векторах [2, 5 – 8, 21].

Исследования способов модификации классических кодов с суммированием показали, что класс кодов с эффективным обнаружением ошибок в информационных векторах, в том числе, в области ошибок малой кратностью, может быть получен с использованием следующего способа построения.

Алгоритм 1. *Последовательность построения модифицированного кода с суммированием взвешенных переходов:*

1. Переходам между разрядами, занимающими соседние позиции в информационных векторах, приписываются весовые коэффициенты из натурального ряда чисел, начиная с перехода между младшими разрядами: $[w_{m,m-1}, w_{m-1,m-2}, \ldots, w_2, w_1] = [m-1, m-2, \ldots, 2, 1]$.
2. Устанавливается значение модуля $M = 2^{\lceil \log_2(m+1) \rceil - 1}$.
3. Подсчитывается сумма весовых коэффициентов активных переходов:

$$W = \sum_{i=1}^{i=m-1} w_{i+1,i}\left(f_{i+1} \oplus f_i\right), \quad (1)$$

где f_i – значение i-го разряда в информационном векторе.

4. Определяется наименьший неотрицательный вычет числа W по модулю M: $W_M = W(\mathrm{mod} M)$.
5. Подсчитывается специальный поправочный коэффициент α, как сумма по модулю два значений заранее выбранных информационных разрядов.
6. Вычисляется модифицированный вес информационного вектора:

$$V = W_M \left(\mathrm{mod}\, M\right) + \alpha M. \quad (2)$$

7. Число V представляется в двоичном виде и записывается в разряды контрольного вектора.

Получаемый по алгоритму 1 код обозначим как *RWT*(*m*,*k*)-код, отдельно указывая формулу подсчета поправочного коэффициента α. Следует отметить, что в [22] приведен способ построения одного из *RWT*(*m*,*k*)-кодов, в котором предполагается, что существует только единственный способ модификации: путем вычисления поправочного коэффициента по формуле $\alpha = f_{m-k} \oplus \ldots \oplus f_{m-1} \oplus f_m$.

Действия алгоритма 1 иллюстрируются в табл. 1 при построении *RWT*(4,3)-кода с поправочным коэффициентом, вычисляемым по формуле $\alpha = f_3 \oplus f_4$.

При построении *RWT*(4,3)-кода с $\alpha = f_3 \oplus f_4$ информационные векторы, имеющие равноудаленное расположение от центра таблицы задания кода (см. табл. 1), имеют одно и то же значение модифицированного веса V. Кроме того, в первой половине таблицы, благодаря поправочному коэффициенту, присутствуют по одному разу все веса из множества $V \in \{0,1,\ldots,7\}$. Это обуславливает равномерное распределение всех информационных векторов между всеми контрольными векторами (табл. 2) и, что важнее, наличие в каждой контрольной группе информационных векторов с кодовым расстоянием $d=m$. Таким образом, в классе необнаруживаемых рассматриваемым кодом ошибок находятся только четырехкратные ошибки в

информационных векторах. Ошибки же с кратностями $d \leq 3$ данным кодом обнаруживаются.

Табл. 1. Кодовые слова RWT(4,3)-кода с $\alpha = f_3 \oplus f_4$

Table 1. Code word of RWT(4,3)-code with $\alpha = f_3 \oplus f_4$

Десятичный эквивалент информационного вектора	Значения разрядов информационного вектора				W	W(mod4)	α	V	Значения разрядов контрольного вектора		
	f_4	f_3	f_2	f_1					g_3	g_2	g_1
	Весовые коэффициенты переходов										
	$w_{4,3}$=3		$w_{3,2}$=2	$w_{2,1}$=1							
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	0	1	0	0	1
2	0	0	1	0	3	3	0	3	0	1	1
3	0	0	1	1	2	2	0	2	0	1	0
4	0	1	0	0	5	1	1	5	1	0	1
5	0	1	0	1	6	2	1	6	1	1	0
6	0	1	1	0	4	0	1	4	1	0	0
7	0	1	1	1	3	3	1	7	1	1	1
8	1	0	0	0	3	3	1	7	1	1	1
9	1	0	0	1	4	0	1	4	1	0	0
10	1	0	1	0	6	2	1	6	1	1	0
11	1	0	1	1	5	1	1	5	1	0	1
12	1	1	0	0	2	2	0	2	0	1	0
13	1	1	0	1	3	3	0	3	0	1	1
14	1	1	1	0	1	1	0	1	0	0	1
15	1	1	1	1	0	0	0	0	0	0	0

Табл. 2. Распределение информационных векторов RWT(4,3)-кода на контрольные группы

Table 2. Data vectors of RWT(4,3)-code distribution on check groups

V							
0	1	2	3	4	5	6	7
Контрольные векторы							
000	001	010	011	100	101	110	111
Информационные векторы							

0000	0001	0011	0010	0110	0100	0101	0111
1111	1110	1100	1101	1001	1011	1010	1000

Исследования показывают, что характеристики обнаружения ошибок модифицированными кодами с суммированием взвешенных переходов напрямую определяются способом вычисления поправочного коэффициента α. Например, в табл. 3 даются распределения необнаруживаемых ошибок по видам и кратностям для семейства *RWT*(4,3)-кодов с всевозможными способами подсчета поправочного коэффициента. В каждой клетке таблицы указаны общее количество необнаруживаемых ошибок, а также три числа через косую черту – число монотонных, симметричных и асимметричных необнаруживаемых ошибок.

Табл. 3. Характеристики обнаружения ошибок различными RWT(4,3)-кодами
Table 3. Error-detection characteristics by different RWT(4,3)-code

Формула поправочного коэффициента α	Общее количество необнаруживаемых ошибок	Распределение необнаруживаемых ошибок по кратностям *d*			
		1	2	3	4
$\alpha = 0$, $\alpha = f_1 \oplus f_2 \oplus f_3 \oplus f_4$	48 18 / 22 / 8	0	32 16 / 16 / 0	0	16 2 / 6 / 8
$\alpha = f_1 \oplus f_3$, $\alpha = f_2 \oplus f_3$, $\alpha = f_1 \oplus f_4$, $\alpha = f_2 \oplus f_4$	32 10 / 14 / 8	0	16 8 / 8 / 0	0	16 2 / 6 / 8
$\alpha = f_1$, $\alpha = f_2$, $\alpha = f_3$, $\alpha = f_4$, $\alpha = f_1 \oplus f_2 \oplus f_3$, $\alpha = f_1 \oplus f_2 \oplus f_4$, $\alpha = f_1 \oplus f_3 \oplus f_4$, $\alpha = f_2 \oplus f_3 \oplus f_4$	16 8 / 8 / 0	0	16 8 / 8 / 0	0	0
$\alpha = f_1 \oplus f_2$, $\alpha = f_3 \oplus f_4$	16 2 / 6 / 8	0	0	0	16 2 / 6 / 8

В табл. 4 приводятся рассчитанные значения долей необнаруживаемых ошибок кратностью *d* от общего количества ошибок данной кратностью (величин β_d, %) для различных кодов с суммированием с длиной информационного вектора *m*=4. В таблице представлены данные для семейства *RWT*(4,3)-кодов с различными способами подсчета поправочного

коэффициента α, а также величины для классического кода Бергера (*S*(4,3)-кода) [23], модифицированного кода Бергера (*RS*(4,3)-кода) [24], а также модульных кодов с суммированием взвешенных разрядов (*WSM*(4,3)-кода) и переходов (*WTM*(4,3)-кода) [25]. Некоторые из разработанных *RWT*(4,3)-кодов имеют улучшенные характеристики обнаружения ошибок по сравнению с известными кодами с суммированием.

Табл. 4. Обнаружение ошибок кодами с суммированием при m=4
Table 4. Error-detection by sum codes with m=4

Код	Формула поправочного коэффициента α	β_d, %				
		1	2	3	4	1 ÷ 4
Модифицированный код с суммированием взвешенных переходов						
RWT(4,3)	$\alpha = 0$, $\alpha = f_1 \oplus f_2 \oplus f_3 \oplus f_4$	0	33,333	0	100	20
RWT(4,3)	$\alpha = f_1 \oplus f_3$, $\alpha = f_2 \oplus f_3$, $\alpha = f_1 \oplus f_4$, $\alpha = f_2 \oplus f_4$	0	16,667	0	100	13,333
RWT(4,3)	$\alpha = f_1$, $\alpha = f_2$, $\alpha = f_3$, $\alpha = f_4$, $\alpha = f_1 \oplus f_2 \oplus f_3$, $\alpha = f_1 \oplus f_2 \oplus f_4$, $\alpha = f_1 \oplus f_3 \oplus f_4$, $\alpha = f_2 \oplus f_3 \oplus f_4$	0	16,667	0	0	6,667
RWT(4,3)	$\alpha = f_1 \oplus f_2$, $\alpha = f_3 \oplus f_4$	0	0	0	100	6,667
Код Бергера						
S(4,3)	–	0	50	0	37,5	22,5
Другие модифицированные коды с суммированием						
RS(4,3)	$\alpha = f_1$, $\alpha = f_2$, $\alpha = f_3$, $\alpha = f_4$, $\alpha = f_1 \oplus f_2 \oplus f_3$, $\alpha = f_1 \oplus f_2 \oplus f_4$, $\alpha = f_1 \oplus f_3 \oplus f_4$, $\alpha = f_2 \oplus f_3 \oplus f_4$	0	25	0	0	10
RS(4,3)	$\alpha = f_1 \oplus f_2$, $\alpha = f_3 \oplus f_4$ $\alpha = f_1 \oplus f_3$, $\alpha = f_2 \oplus f_3$, $\alpha = f_1 \oplus f_4$, $\alpha = f_2 \oplus f_4$	0	16,667	0	50	10
WSM(4,3)	–	0	0	18,75	25	6,667
WTM(4,3)	–	0	8,333	0	100	10

Среди всех *RWT*(4,3)-кодов выделяются два кода, поправочные коэффициенты которых вычисляются по формулам $\alpha = f_1 \oplus f_2$ и $\alpha = f_3 \oplus f_4$. Данными кодами обнаруживаются любые ошибки с кратностями $d \leq 3$. При этом генераторы данных кодов имеют простые структуры (рис. 1).

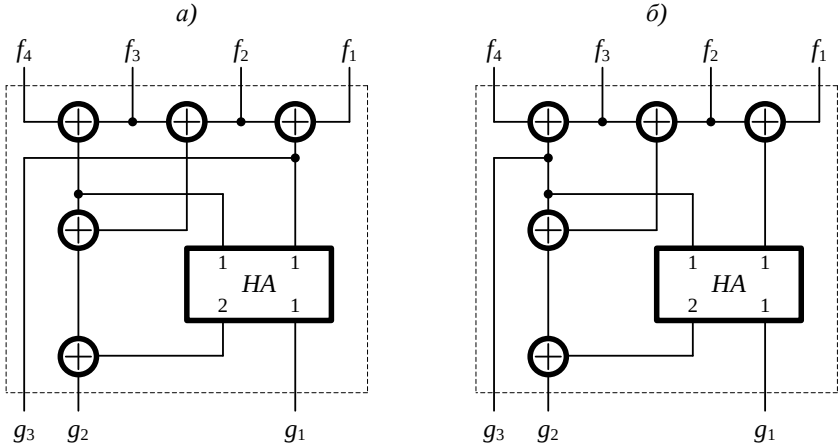


Рис. 1. Схемы генераторов *RWT*(4,3)-кодов: а) с $\alpha = f_1 \oplus f_2$; б) с $\alpha = f_3 \oplus f_4$

Fig. 1. RWT(4,3)-code generators circuits: a) with $\alpha = f_1 \oplus f_2$; b) with $\alpha = f_3 \oplus f_4$

3. Характеристики обнаружения ошибок модифицированными кодами с суммированием взвешенных переходов

К сожалению, с увеличением длины информационного вектора свойство обнаружения *RWT*(*m*,*k*)-кодами любых трехкратных ошибок в информационных векторах теряется. Однако модифицированные коды с суммированием взвешенных переходов остаются по характеристикам обнаружения ошибок различных кратностей более эффективными, чем рассмотренные выше коды с суммированием (см. табл. 4).

Исследования *RWT*(*m*,*k*)-кодов при различных длинах информационных векторов показали следующие важные особенности. Различные способы вычисления поправочного коэффициента α дают различные по своим характеристикам *RWT*(*m*,*k*)-коды. Помехоустойчивые *RWT*(*m*,*k*)-коды могут быть построены при любых сочетаниях информационных разрядов в сумме поправочного коэффициента α для любых значений длин информационных векторов за исключением $m = 2^q + 1,\ q = 1,2,...$ Для построения помехоустойчивых *RWT*(*m*,*k*)-кодов при $m = 2^q + 1,\ q = 1,2,...$, необходимо, что в сумме поправочного коэффициента α присутствовал старший

информационный разряд. Такая особенность модифицированных взвешенных кодов связана с тем, что при значении длины информационного вектора $m = 2^q + 1$ значение модуля $M = 2^q$ и старший разряд f_m, переход между которым и разрядом f_{m-1}, имеет весовой коэффициент, равный числу M, не контролируется разрядами контрольного вектора [26]. Для наделения $RWT(m,k)$-кода свойством помехоустойчивости в рассмотренном случае требуется контроль старшего разряда в сумме поправочного коэффициента.

Следует также подчеркнуть, что $RWT(m,k)$-коды, для которых поправочный коэффициент вычисляется по формулам $\alpha = 0$ и $\alpha = f_1 \oplus f_2 \oplus ... \oplus f_{m-1} \oplus f_m$, не эффективно используют старший контрольный разряд, ввиду чего не обнаруживают большее количество ошибок, чем все остальные модифицированных коды с суммированием взвешенных переходов.

Дальнейшие исследования показали, что число кодов с различными характеристиками обнаружения ошибок гораздо меньше числа способов подсчета поправочного коэффициента. Например, в табл. 5 приведены характеристики обнаружения ошибок по кратностям всеми $RWT(m,k)$-кодами при m=5 ÷ 7, а в табл. 6 – $RWT(m,k)$-кодами при m=8. При бóльших значениях длин информационных векторов число разнообразных $RWT(m,k)$-кодов является существенным. При этом для коэффициента α для сокращения записи использовано следующее обозначение: коэффициент α представлен в виде десятичного числа-эквивалента формулы подсчета (например, число 57 представляется в двоичном виде как $<f_7\ f_6\ f_5\ f_4\ f_3\ f_2\ f_1>$ = <0111001>, что означает использование при вычислении поправочного коэффициента формулы $\alpha = f_1 \oplus f_4 \oplus f_5 \oplus f_6$).

Модифицированным кодам с суммированием взвешенных переходов присуща следующая закономерность. При четных значениях m $RWT(m,k)$-коды обнаруживают любые ошибки нечетной кратностью в информационных векторах. Это свойство характерно также любому коду с суммированием единичных информационных разрядов [8].

Отдельно следует упомянуть о модифицированных взвешенных кодах с обнаружением любых двукратных ошибок в информационных векторах. Такие коды строятся при m=4, 5 и 8. Из табл. 3 и 5 следует, что существует по два способа построения $RWT(m,k)$-кодов с обнаружением любых двукратных ошибок при значениях m=4 и m=5. При m=8 таких способов 16. Укажем десятичные эквиваленты поправочных коэффициентов для данных $RWT(m,k)$-кодов: (135, 165, 210, 240), (143, 173, 218, 248), (147, 177, 198, 228), (155, 185, 206, 236). В скобках выделены способы вычисления поправочных коэффициентов α с абсолютно идентичными характеристиками обнаружения ошибок в информационных векторах.

При этом, что является важным для задач технической диагностики, некоторые способы вычисления поправочного коэффициента α дают возможность построения таких *RWT*(*m*,*k*)-кодов, которые имеют минимально возможное общее количество необнаруживаемых ошибок при заданных значениях *m* и *k* [24].

Табл. 5. Обнаружение ошибок RWT(5,3) и RWT(6,3)
Table 5. Error-detection by RWT(5,3) and RWT(6,3) codes

Способ вычисления поправочного коэффициента α	Число необнаруживаемых ошибок кратностью *d* и значения величин β_d						
	2	3	4	5	6	7	2 ÷ 7
m=5							
19, 28	0	64	32	0			96
	0%	20%	20%	0%			9,68%
17, 18, 20, 23, 24, 27, 29, 30	32	32	0	32			96
	10%	10%	0%	100%			9,68%
21, 22, 25, 26	32	32	32	0			96
	10%	10%	20%	0%			9,68%
16, 31	64	0	32	0			96
	20%	0%	20%	0%			9,68%
m=6							
19, 28, 35, 44	128	0	320	0	0		448
	13,33%	0%	33,33%	0%	0%		11,11%
3, 12, 15, 17, 18, 20, 23, 24, 27, 29, 30, 33, 34, 36, 39, 40, 43, 45, 46, 48, 51, 60	192	0	192	0	64		448
	20%	0%	20%	0%	100%		11,11%
7, 11, 13, 14, 21, 22, 25, 26, 37, 38, 41, 42, 49, 50, 52, 56	192	0	256	0	0		448
	20%	0%	26,67%	0%	0%		11,11%
16, 31, 32, 47	256	0	192	0	0		448
	26,67%	0%	20%	0%	0%		11,11%
5, 6, 9, 10, 53, 54, 57, 58	256	0	256	0	64		576
	26,67%	0%	26,67%	0%	100%		14,29%
1, 2, 4, 8, 55, 59, 61, 62	320	0	128	0	0		448
	33,33%	0%	13,33%	0%	0%		11,11%
0, 63	448	0	448	0	64		960
	46,67%	0%	46,67%	0%	100%		23,81%

Что касается распределений ошибок по видам для *RWT*(*m*,*k*)-кодов, то оно также разнообразно для различных способов подсчета поправочного коэффициента. Например, на рис. 2 представлено распределение ошибок по видам для *RWT*(8,4)-кодов (по оси абсцисс расположены десятичные

эквиваленты формул подсчета поправочного коэффициента α, а по оси ординат – количество необнаруживаемых ошибок данного вида). Распределения ошибок по видам симметричны относительно центральных значений десятичных эквивалентов поправочных коэффициентов при четных значениях m, при нечетных значениях данная закономерность нарушается. Для малых значений длин информационных векторов $m \leq 6$ более распространенными оказывают симметричные необнаруживаемые ошибки. При бóльших значениях m наименьшим является количество монотонных, затем симметричных и асимметричных ошибок в классе необнаруживаемых *RWT*(m,k)-кодами ошибок.

Табл. 6. Обнаружение ошибок RWT(7,3)-кодами

Table 6. Error-detection by RWT(7,3)-codes

Способ вычисления поправочного коэффициента α	Число необнаруживаемых ошибок кратностью d и значения величин β_d						
	2	3	4	5	6	7	2 ÷ 7
7, 11, 13, 14, 19, 28, 35, 44, 49, 50, 52, 56, 67, 76, 79, 112, 115, 124	256	512	640	512	0	0	1920
	9,52%	11,43%	14,29%	19,05%	0%	0%	11,81%
3, 12, 15, 48, 51, 60, 71, 75, 77, 78, 83, 92, 99, 108, 113, 114, 116, 120	256	640	640	256	0	128	1920
	9,52%	14,29%	14,29%	9,52%	0%	100%	11,81%
21, 22, 25, 26, 37, 38, 41, 42	384	384	768	384	0	0	1920
	14,29%	8,57%	17,14%	14,29%	0%	0%	11,81%
5, 6 , 9, 10, 17, 18, 20, 23, 24, 27, 29, 30, 33, 34, 36, 39, 40, 43, 45, 46, 53, 54, 57, 58	384	512	512	384	0	128	1920
	14,29%	11,43%	11,43%	14,29%	0%	100%	11,81%
65, 66, 68, 72, 80, 95, 96, 111, 119, 123, 125, 126	512	384	384	512	0	128	1920
	19,05%	8,57%	8,57%	19,05%	0%	100%	11,81%
69, 70, 73, 74, 81, 82, 84, 87, 88, 91, 93, 94, 97, 98, 100, 103, 104, 107, 109, 110, 117, 118, 121, 122	384	640	512	384	0	0	1920
	14,29%	14,29%	11,43%	14,29%	0%	0%	11,81%
85, 86, 89, 90, 101, 102, 105, 106	384	768	768	384	0	128	2432
	14,29%	17,14%	17,14%	14,29%	0%	100%	14,96%
1, 2, 4, 8, 16, 31, 32, 47, 55, 59, 61, 62	512	768	384	256	0	0	1920
	19,05%	17,14%	8,57%	9,52%	0%	0%	11,81%
64, 127	768	0	1152	0	0	0	1920
	28,57%	0%	25,71%	0%	0%	0%	11,81%
0, 63	768	1152	1152	768	0	128	3968
	28,57%	25,71%	25,71%	28,57%	0%	100%	24,41%

Подобные графики (см. рис. 2) позволяют определить диапазоны разбросов значений чисел необнаруживаемых кодами ошибок различных видов для целого семейства *RWT*(*m*,*k*)-кодов заданной длины.

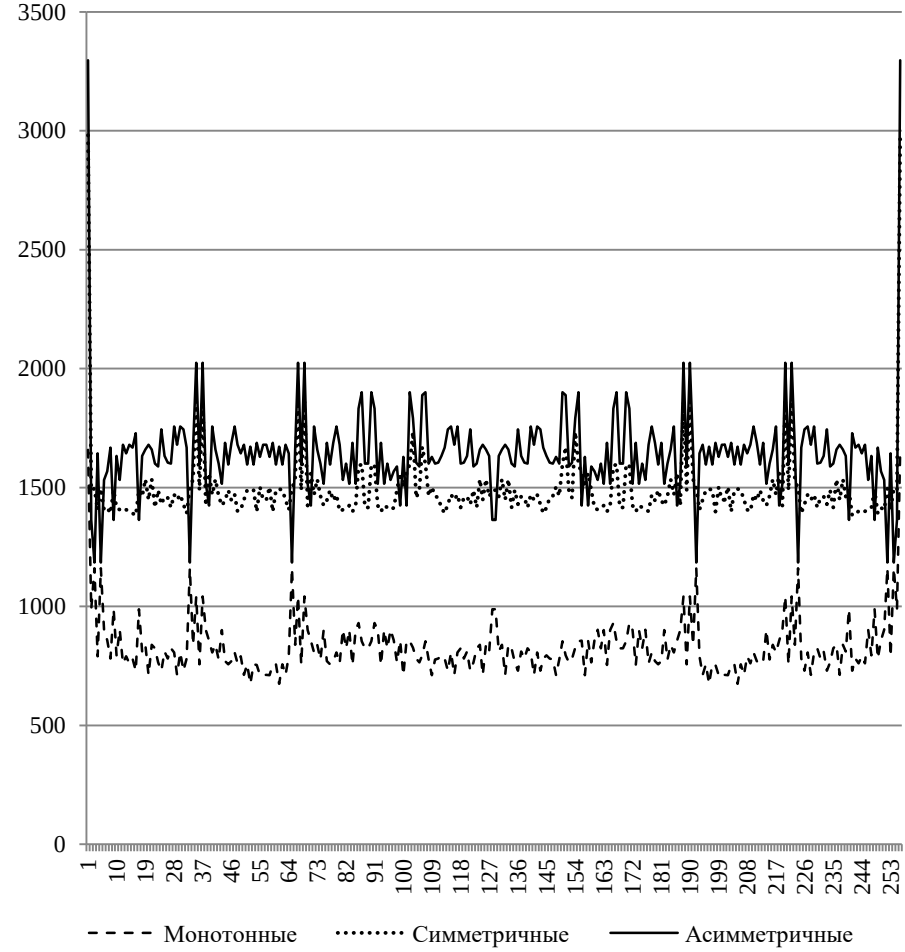


Рис. 2. Распределение необнаруживаемых ошибок по видам для различных RWT(8,4)-кодов

Fig. 2. Distribution of undetectable errors by types for different RWT(8,4)-codes

4. Понятие об r-независимых группах выходов комбинационных схем

RWT(*m*,*k*)-коды могут быть использованы при синтезе систем функционального контроля комбинационных логических схем. В особенности это касается кодов с обнаружением любых ошибок с кратностями $d \leq 3$ – *RWT*(4,3)-кодов с $\alpha = f_1 \oplus f_2$ и $\alpha = f_3 \oplus f_4$.

RWT(4,3)-коды могут быть эффективно использованы при организации контроля комбинационных схем по группам, содержащим по 4 выхода. Каждая такая группа должна удовлетворять одному условию – выходы группы не должны допускать одновременного искажения значений, то есть обладать независимостью относительно четырехкратных искажений. Сформулируем условия поиска логических элементов G_t комбинационных схем допускающих четырехкратные искажения.

Известны следующие положения [2, 11].

Определение 1. Подмножество выходов комбинационной схемы $\{f_{i_1}, f_{i_2},..., f_{i_q}\}$ ($\{i_1, i_2,..., i_q\} \in \{1,2,...,n\}$, n – число выходов схемы) является группой независимых выходов (образует H^1-группу), если неисправность выхода любого логического элемента схемы искажает значения только одного выхода группы.

Теорема 1. Множество выходов комбинационной схемы $\{f_{i_1}, f_{i_2},..., f_{i_q}\}$ ($i_1, i_2,..., i_q \in \{1,2,...,n\}$, n – число выходов схемы) образует H^1-группу, если для каждой пары выходов $\{f_{i_a}, f_{i_c}\}$ ($a, b \in \{1,2,...,q\}$) и для каждого логического элемента G_t выполняется условие:

$$\frac{\partial f_{i_a}}{\partial y_t} \frac{\partial f_{i_b}}{\partial y_t} = 0, \qquad (3)$$

где f_{i_a} и f_{i_b} – функции, реализуемые на соответствующих выходах логической схемы;

y_t – функция, реализуемая на выходе элемента G_t.

Используя понятие независимых выходов, введем понятие H^r-группы выходов.

Определение 2. Множество выходов схемы $\{f_{i_1}, f_{i_2},..., f_{i_q}\}$ ($i_1, i_2,..., i_q \in \{1,2,...,n\}$) является r-независимой группой выходов (H^r-группой), если неисправность выхода любого логического элемента схемы искажает значения не более r выходов группы.

Теорема 2. Множество выходов логической схемы $\{f_{i_1}, f_{i_2},..., f_{i_q}\}$ ($q \geq r+1$) образует H^r-группу, если для каждого из C_q^{r+1} подмножеств из $r+1$ выходов и для каждого элемента логической схемы выполняется условие:

$$\frac{\partial f_{i_1}}{\partial y_t}\frac{\partial f_{i_2}}{\partial y_t}\cdot...\cdot\frac{\partial f_{i_{r+1}}}{\partial y_t}=0. \qquad (4)$$

Доказательство. Справедливость данного положения следует из того, что каждая производная в левой части выражения (4) определяет те входные наборы, на которых неисправность элемента с выходом y_t проявляется на соответствующем выходе, а произведение всех четырех производных определяет те входные наборы, на которых одновременно искажаются все четыре выхода. Если для каждого элемента условие (4) выполняется, значит, на множестве выходов ни при каких условиях не будут возникать искажения с кратностью $d \geq r+1$. **Доказательство завершено.**

При организации контроля логических устройств по группам r-независимых выходов требуется определить конкретное значение числа r. Это делается с учетом свойств обнаружения ошибок выбранным на этапе проектирования системы функционального контроля кодом: $r=d_{\min}-1$, где $d_{\min}$ – минимальное расстояние Хэмминга для кодовых слов выбранного кода. Например, при использовании *RWT*(4,3)-кодов с $\alpha = f_1 \oplus f_2$ и $\alpha = f_3 \oplus f_4$ $d_{\min}=4$, а значит, контроль схем может быть осуществлен на основе выделения H^3-групп выходов.

Для примера рассмотрим схемы, изображенные на рис. 3.

Четырехкратные искажения может вызывать только элемент G_1. Определим, является ли каждая из схем схемой с H^3-группой выходов. Для этого выпишем функции, реализуемые на выходах схем. Для схемы на рис. 3, а) имеем:

$f_1 = y_1$, $f_2 = y_1 \vee \overline{x_1 \vee x_3}$, $f_3 = y_1 \vee x_1\overline{x_2}x_3$ и $f_4 = \overline{y_1\overline{x_2 \vee x_3}}$.

Вычислим булевы производные для каждого из выходов:

$$\frac{\partial f_1}{\partial y_1} = (y_1 = 0) \oplus (y_1 = 1) = 1;$$

$$\frac{\partial f_2}{\partial y_1} = (y_1 = 0) \vee \overline{x_1 \vee x_3} \oplus (y_1 = 1) \vee \overline{x_1 \vee x_3} = x_1 \vee x_3;$$

$$\frac{\partial f_3}{\partial y_1} = (y_1 = 0) \vee x_1\overline{x_2}x_3 \oplus (y_1 = 1) \vee x_1\overline{x_2}x_3 = \overline{x_1} \vee x_2 \vee \overline{x_3};$$

$$\frac{\partial f_4}{\partial y_1} = \overline{(y_1 = 0)\overline{x_2 \vee x_3}} \oplus \overline{(y_1 = 1)\overline{x_2 \vee x_3}} = \overline{x_2}\,\overline{x_3};$$

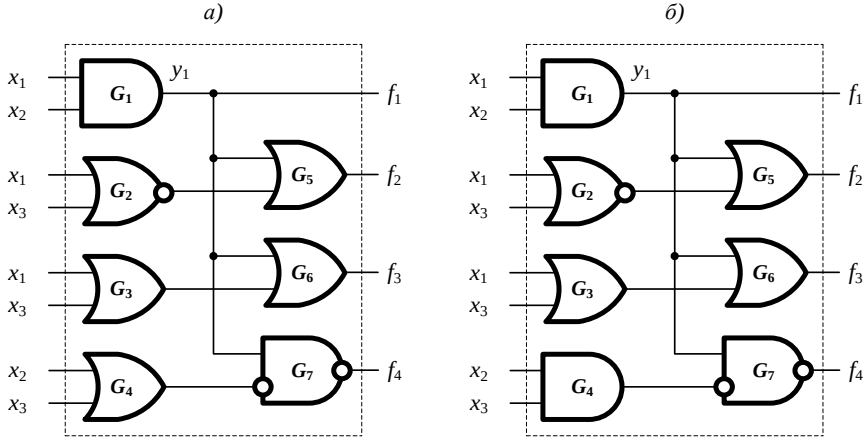


Рис. 3. Комбинационные схемы: а) схема, выходы которой образуют 3-независимую группу; б) схема, выходы которой не образуют 3-независимую группу
Fig 3. Combinational circuits: a) circuit with 3-independence outputs group; b) circuit without 3-independence outputs group

$$\frac{\partial f_1}{\partial y_1}\frac{\partial f_2}{\partial y_1}\frac{\partial f_3}{\partial y_1}\frac{\partial f_4}{\partial y_1} = 1 \cdot (x_1 \vee x_3)\left(\overline{x_1} \vee x_2 \vee \overline{x_3}\right)\left(\overline{x_2}\,\overline{x_3}\right) = 0.$$

Выводом из последнего выражения является то, что схема на рис. 3, а) является схемой с 3-независимыми выходами.

Для схемы на рис. 3, б) отличается только функция f_4:

$$f_4 = \overline{y_1 \overline{x_2 x_3}}.$$

Вычислим производную для данной функции по переменной y_1:

$$\frac{\partial f_4}{\partial y_1} = \overline{(y_1 = 0)\overline{x_2 x_3}} \oplus \overline{(y_1 = 1)\overline{x_2 x_3}} = \overline{x_2} \vee \overline{x_3};$$

$$\frac{\partial f_1}{\partial y_1}\frac{\partial f_2}{\partial y_1}\frac{\partial f_3}{\partial y_1}\frac{\partial f_4}{\partial y_1} = 1 \cdot (x_1 \vee x_3)\left(\overline{x_1} \vee x_2 \vee \overline{x_3}\right)\left(\overline{x_2} \vee \overline{x_3}\right) = \overline{x_1}\,\overline{x_2} x_3 \vee x_1 \overline{x_3}.$$

Схема на рис. 3, б) не является 3-независимой.

5. Алгоритм синтеза системы функционального контроля на основе выделения 3-независимых групп выходов

Сформируем алгоритм построения системы функционального контроля на основе выделения H^3-групп на множестве выходов контролируемой комбинационной схемы и контроле каждой из групп на основе *RWT*(4,3)-кода.

Алгоритм 2. *Последовательность построения системы функционального контроля:*

1. Определяется множество Q^4 логических элементов G_i, связанных путями с четырьмя и более выходами схемы.
2. Для каждого логического элемента G_i из подмножества Q^4 определяется подмножество выходов схемы $G_i(f)=\{f_{i_1}, f_{i_2},..., f_{i_q}\}$, с которыми он связан.
3. Определяется подмножество $G^0(f)$ путем объединения всех подмножеств, полученных в п.2.
4. Рассматривается каждое подмножество $G_i^4(f)\in G^0(f)$, содержащее три выхода. Для него проверяется условие теоремы 2 относительно элементов $G_i \in Q^4$.
5. По результатам выполнения п.4 составляется множество W_z групп выходов $G_i^4(f)$, которые не являются H^3-группами.
6. Решается задача покрытия выходов схемы подмножествами $G_i^4(f)$ 3-независимых групп выходов с учетом минимального количества таких групп.
7. Каждая 3-независимая группа выходов схемы, полученная в п.6, контролируется на основе *RWT*(4,3)-кода (с $\alpha = f_1 \oplus f_2$, либо с $\alpha = f_3 \oplus f_4$) – строятся отдельные подсхемы контроля каждой группы выходов.
8. Если какие-либо выходы схемы не вошли в покрытие выходов подмножествами $G_i^4(f)$, для них реализуется подсхема контроля по методу дублирования.
9. Выходы подсхем контроля объединяются на входах самопроверяемого компаратора, реализуемого на основе стандартных модулей сжатия парафазных сигналов *TRC* [27].

Следует отметить, что при реализации шагов алгоритма 2 могут, в конечном итоге, остаться H^2-группы и H^1-группы выходов. Они также могут контролироваться на основе *RWT*(4,3)-кодов, однако эффективнее может оказаться применение для контроля H^2-групп кодов с обнаружением любых

одно- и двукратных ошибок [28 – 31] и для контроля H^1-групп выходов кодов паритета [32, 33].

6. Заключение

Предложенный в статье класс модифицированных взвешенных кодов с суммированием переходов между разрядами, занимающими соседние позиции в информационных векторах, можно эффективно использовать в задачах технической диагностики дискретных систем. При этом при значении m=4 получены коды с обнаружением любых ошибок кратностью $d \leq 3$. Используя последние и выделяя во множестве выходов комбинационных схем группы 3-независимых выходов, можно строить системы функционального контроля с обнаружением любых одиночных константных неисправностей во внутренней структуре контролируемых объектов.

Список литературы

[1]. McCluskey E.J. Logic Design Principles: With Emphasis on Testable Semicustom Circuits. N.J.: Prentice Hall PTR, 1986, 549 p.

[2]. Согомонян Е.С., Слабаков Е.В. Самопроверяемые устройства и отказоустойчивые системы. М.: Радио и связь, 1989, 207 с.

[3]. Goessel M., Graf S. Error Detection Circuits. – London: McGraw-Hill, 1994, 261 p.

[4]. Mitra S., McCluskey E.J. Which Concurrent Error Detection Scheme to Choose? Proceedings of International Test Conference, 2000, USA, Atlantic City, NJ, 03-05 October 2000, pp. 985-994, doi: 10.1109/TEST.2000.894311.

[5]. Berger J.M. A Note on Error Detection Codes for Asymmetric Channels. Information and Control, 1961, Vol. 4, Issue 1, pp. 68-73, doi: 10.1016/S0019-9958(61)80037-5.

[6]. Piestrak S.J. Design of Self-Testing Checkers for Unidirectional Error Detecting Codes. Wrocław: Oficyna Wydawnicza Politechniki Wrocławskiej, 1995, 111 p.

[7]. Das D., Touba N.A. Synthesis of Circuits with Low-Cost Concurrent Error Detection Based on Bose-Lin Codes. Journal of Electronic Testing: Theory and Applications, 1999, Vol. 15, Issue 1-2, pp. 145-155, doi: 10.1023/A:1008344603814.

[8]. Блюдов А.А., Ефанов Д.В., Сапожников В.В., Сапожников Вл.В. О кодах с суммированием единичных разрядов в системах функционального контроля. Автоматика и телемеханика, 2014, №8, стр. 131-145.

[9]. Freiman C.V. Optimal Error Detection Codes for Completely Asymmetric Binary Channels. Information and Control, 1962, Vol. 5, Issue 1, pp. 64-71, doi: 10.1016/S0019-9958(62)90223-1.

[10]. Burkatovskaya Yu.B., Butorina N.B., Matrosova A.Yu. Self-Testing Checker Design for Arbitrary Number of Code Words of (m,n) code. Proceedings of the 10th International Baltic Electronic Conference (BEC`2006), Tallinn, Estonia, October 2-4, 2006, pp. 183-186, doi: 10.1109/BEC.2006.311093.

[11]. Слабаков Е.В., Согомонян Е.С. Самопроверяемые вычислительные устройства и системы (обзор). Автоматика и телемеханика, 1981, №11, стр. 147-167.

[12]. Nicolaidis M., Zorian Y. On-Line Testing for VLSI – A Compendium of Approaches. Journal of Electronic Testing: Theory and Applications, 1998, №12, pp. 7-20. DOI: 10.1023/A:1008244815697.

[13]. Matrosova A.Yu., Levin I., Ostanin S.A. Self-Checking Synchronous FSM Network Design with Low Overhead. VLSI Design, 2000, Vol. 11, Issue 1, pp. 47-58. DOI: 10.1155/2000/46578.

[14]. Tshagharyan G., Harutyunyan G., Shoukourian S., Zorian Y. Experimental Study on Hamming and Hsiao Codes in the Context of Embedded Applications. Proceedings of 15th IEEE East-West Design & Test Symposium (EWDTS`2017), Novi Sad, Serbia, September 29 – October 2, 2017, pp. 1-4.

[15]. Аксёнова Г.П. Необходимые и достаточные условия построения полностью проверяемых схем свертки по модулю 2. Автоматика и телемеханика, 1979, №9, стр. 126-135.

[16]. Гессель М., Согомонян Е.С. Построение самотестируемых и самопроверяемых комбинационных устройств со слабонезависимыми выходами. Автоматика и телемеханика, 1992, № 8 стр. 150-160.

[17]. Busaba F.Y., Lala P.K. Self-Checking Combinational Circuit Design for Single and Unidirectional Multibit Errors // Journal of Electronic Testing: Theory and Applications, 1994, Issue 1, pp. 19-28. DOI: 10.1007/BF00971960.

[18]. Morosow A, Saposhnikov V.V., Saposhnikov Vl.V., Goessel M. Self-Checking Combinational Circuits with Unidirectionally Independent Outputs. VLSI Design, 1998, Vol. 5, Issue 4m pp. 333-345. DOI: 10.1155/1998/20389.

[19]. Göessel M., Ocheretny V., Sogomonyan E., Marienfeld D. New Methods of Concurrent Checking: Edition 1. – Dordrecht: Springer Science+Business Media B.V., 2008, 184 p.

[20]. Ефанов Д.В., Сапожников В.В., Сапожников Вл.В. Условия обнаружения неисправности логического элемента в комбинационном устройстве при функциональном контроле на основе кода Бергера. Автоматика и телемеханика, 2017, №5, стр. 152-165.

[21]. Сапожников В.В., Сапожников Вл.В., Ефанов Д.В., Дмитриев В.В. Новые структуры систем функционального контроля логических схем. Автоматика и телемеханика, 2017, №2, стр. 127-143.

[22]. Мехов В.Б., Сапожников В.В., Сапожников Вл.В. Контроль комбинационных схем на основе модифицированных кодов с суммированием. Автоматика и телемеханика, 2008, №8, стр. 153-165.

[23]. Ефанов Д.В., Сапожников В.В., Сапожников Вл.В. О свойствах кода с суммированием в схемах функционального контроля. Автоматика и телемеханика, 2010. №6, стр. 155-162.

[24]. Блюдов А.А., Ефанов Д.В., Сапожников В.В., Сапожников Вл.В. Построение модифицированного кода Бергера с минимальным числом необнаруживаемых ошибок информационных разрядов. Электронное моделирование, 2012, Том 34, №6, стр. 17-29.

[25]. Сапожников В.В., Сапожников Вл.В., Ефанов Д.В. Модульно-взвешенные коды с суммированием с наименьшим общим числом необнаруживаемых ошибок в информационных векторах. Электронное моделирование, 2017, Том 39, №4

[26]. Сапожников В.В., Сапожников Вл.В., Ефанов Д.В., Котенко А.Г. Модульные коды с суммированием взвешенных переходов с последовательностью весовых коэффициентов, образующей натуральный ряд чисел. Труды СПИИРАН, 2017, №1, стр. 137-164, doi: 10.15622/SP.50.6.

[27]. Lala P.K. Self-Checking and Fault-Tolerant Digital Design. – San Francisco: Morgan Kaufmann Publishers, 2001, 216 p.

[28]. Hamming R.W. Error Detecting and Correcting Codes. Bell System Technical Journal, 1950, 29 (2), pp. 147-160

[29]. Sapozhnikov V., Sapozhnikov Vl., Efanov D., Dmitriev V. Weighted Sum Code Without Carries – is an Optimum Code with Detection of Any Double Errors in Data Vectors // Proceedings of 14th IEEE East-West Design & Test Symposium (EWDTS`2016), Yerevan, Armenia, October 14-17, 2016, pp. 134-141, doi: 10.1109/EWDTS.2016.7807686.

[30]. Sapozhnikov V., Sapozhnikov Vl., Efanov D., Dmitriev V. New Sum Code for Effective Detection of Double Errors in Data Vectors // Proceedings of 13th IEEE East-West Design & Test Symposium (EWDTS`2015), Batumi, Georgia, September 26-29, 2015, pp. 154-159, doi: 10.1109/EWDTS.2015.7493123.

[31]. Сапожников В.В., Сапожников Вл.В., Ефанов Д.В., Дмитриев В.В. Код с суммированием взвешенных информационных разрядов без переносов в системах функционального контроля. Автоматика на транспорте, 2017, Том 3, №3, стр. 414-433.

[32]. Ghosh S., Basu S., Touba N.A. Synthesis of Low Power CED Circuits Based on Parity Codes. Proceedings of 23rd IEEE VLSI Test Symposium (VTS'05), 2005, pp. 315-320.

[33]. Аксёнова Г.П. О функциональном диагностировании дискретных устройств в условиях работы с неточными данными. Проблемы управления, 2008, №5, стр. 62-66.

Modified codes with weighted-transitions summation in concurrent error detection systems of combinational circuits

[1] *D.V. Efanov <TrES-4b@yandex.ru>*
V.V. Sapozhnikov <port.at.pgups@gmail.com>
Vl.V. Sapozhnikov <at.pgups@gmail.com>
[1] *Emperor Alexandex I St. Petersburg state transport university,*
190031, Russia, St. Petersburg, Moscowsky ave., 9

Abstract. A method for constructing modified codes with summation of weighted transitions between bits in data vectors occupying neighboring positions is proposed. New codes with summation have the same number of check bits as the classic Berger codes, but they detect more errors in data vectors. Modified codes with summation of weighted transitions in comparison with Berger codes also have improved error detection characteristics in the area of small multiplicity. In addition, for some values of the lengths of information vectors codes can be constructed with the detection of any twofold and any triple errors. The authors developed a method for synthesizing concurrent error detection systems of combinational circuits, based on the analysis of the topology of the object of diagnosis with the selection of groups of checkable outputs, taking into account the properties of error detection by modified codes with summation of weighted transitions. An algorithm for the synthesis of a concurrent error detection systems has been developed.

Keywords: concurrent error-detection system; combinational circuit; Berger code; weight-transition sum code; double errors detection; triple errors detection.

DOI: 10.15514/ISPRAS-2017-29(5)-3

For citation: Sapozhnikov V.V., Sapozhnikov Vl.V., Efanov D.V. Modified codes with weighted-transitions summation in concurrent error detection systems of combinational circuits. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 5, 2017. pp. 39-60 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-3

References

[1]. McCluskey E.J. Logic Design Principles: With Emphasis on Testable Semicustom Circuits. N.J., Prentice Hall PTR, 1986, 549 p.

[2]. Sogomonjan E.S., Slabakov E.V. [Self-checking devices and fault-tolerant systems]. Moscow: Radio i svjaz', 1989, 207 p. (in Russian).

[3]. Goessel M., Graf S. Error Detection Circuits. London: McGraw-Hill, 1994, 261 p.

[4]. Mitra S., McCluskey E.J. Which Concurrent Error Detection Scheme to Choose? Proceedings of International Test Conference, 2000, USA, Atlantic City, NJ, 03-05 October 2000, pp. 985-994, doi: 10.1109/TEST.2000.894311.

[5]. Berger J.M. A Note on Error Detection Codes for Asymmetric Channels. Information and Control, 1961, Vol. 4, Issue 1, pp. 68-73, doi: 10.1016/S0019-9958(61)80037-5.

[6]. Piestrak S.J. Design of Self-Testing Checkers for Unidirectional Error Detecting Codes. Wrocław: Oficyna Wydawnicza Politechniki Wrocławskiej, 1995, 111 p.

[7]. Das D., Touba N.A. Synthesis of Circuits with Low-Cost Concurrent Error Detection Based on Bose-Lin Codes. Journal of Electronic Testing: Theory and Applications, 1999, Vol. 15, Issue 1-2, pp. 145-155, doi: 10.1023/A:1008344603814.

[8]. Bljudov A.A., Efanov D.V., Sapozhnikov V.V., Sapozhnikov Vl.V. On Codes with Summation of Unit Bits In Concurrent Error Detection Systems. Automation and remote control, 2014, Vol. 75, Issue 8, pp. 1460-1470, doi: 10.1134/S0005117914080098.

[9]. Freiman C.V. Optimal Error Detection Codes for Completely Asymmetric Binary Channels. Information and Control. 1962, Vol. 5, Iissue 1, pp. 64-71, doi: 10.1016/S0019-9958(62)90223-1.

[10]. Burkatovskaya Yu.B., Butorina N.B., Matrosova A.Yu. Self-Testing Checker Design for Arbitrary Number of Code Words of (m,n) code. Proceedings of the 10th International Baltic Electronic Conference (BEC`2006), Tallinn, Estonia, October 2-4, 2006, pp. 183-186, doi: 10.1109/BEC.2006.311093.

[11]. Slabakov E.V., Sogomonjan E.S. Self-check computing devices and systems (survey). Automation and remote control, 1981, Vol. 42, Issue 11, pp. 1551-1566.

[12]. Nicolaidis M., Zorian Y. On-Line Testing for VLSI – A Compendium of Approaches. Journal of Electronic Testing: Theory and Applications, 1998, №12, pp. 7-20, doi: 10.1023/A:1008244815697.

[13]. Matrosova A.Yu., Levin I., Ostanin S.A. Self-Checking Synchronous FSM Network Design with Low Overhead. VLSI Design, 2000, Vol. 11, Issue 1, pp. 47-58, doi: 10.1155/2000/46578.

[14]. Tshagharyan G., Harutyunyan G., Shoukourian S., Zorian Y. Experimental Study on Hamming and Hsiao Codes in the Context of Embedded Applications. Proceedings of 15th IEEE East-West Design & Test Symposium (EWDTS`2017), Novi Sad, Serbia, September 29 – October 2, 2017, pp. 1-4.

[15]. Aksjonova G.P. Necessary and sufficient conditions for design of completely checkable modulo 2 convolution circuits. Automation and remote control, 1979, Vol. 40, Issue 9, pp. 1362-1369.

[16]. Gessel' M., Sogomonjan E.S. Design of self-testing and self-checking combinational circuits with weakly independent outputs. Automation and remote control, 1992, Vol. 53, Issue 8, pp. 1264-1272.

[17]. Busaba F.Y., Lala P.K. Self-Checking Combinational Circuit Design for Single and Unidirectional Multibit Errors. Journal of Electronic Testing: Theory and Applications, 1994, Issue 1, pp. 19-28, doi: 10.1007/BF00971960.

[18]. Morosow A, Saposhnikov V.V., Saposhnikov Vl.V., Goessel M. Self-Checking Combinational Circuits with Unidirectionally Independent Outputs. VLSI Design, 1998, Vol. 5, Issue 4, pp. 333-345, doi: 10.1155/1998/20389.

[19]. Göessel M., Ocheretny V., Sogomonyan E., Marienfeld D. New Methods of Concurrent Checking: Edition 1. Dordrecht: Springer Science+Business Media B.V., 2008, 184 p.

[20]. Efanov D.V., Sapozhnikov V.V., Sapozhnikov Vl.V. Conditions for Detecting a Logical Element Fault in a Combination Device under Concurrent Checking Based on Berger`s Code. Automation and remote control, 2017, Vol. 78, Issue 5, pp. 892-902, doi: 10.1134/S0005117917040113.

[21]. Sapozhnikov V.V., Sapozhnikov Vl.V., Efanov D.V., Dmitriev V.V. New Structures of the Concurrent Error Detection Systems for Logic Circuits, Automation and remote control, 2017, Vol. 78, Issue 2, pp. 300-313, doi: 10.1134/S0005117917020096.

[22]. Mehov V.B., Sapozhnikov V.V., Sapozhnikov Vl.V. Checking of combinational circuits basing on modification sum codes. Automation and remote control, 2008, Vol. 69, Issue 8, pp. 1411-1422.

[23]. Efanov D.V., Sapozhnikov V.V., Sapozhnikov Vl.V. On Summation Code Properties In Functional Control Circuits. Automation and remote control, 2010, Vol. 71, Issue 6, pp. 1117-1123, doi: 10.1134/S0005117910060123.

[24]. Bljudov A.A., Efanov D.V., Sapozhnikov V.V., Sapozhnikov Vl.V. Formation of the Berger Modified Code with Minimum Number of Undetectable Errors of Informational Bits. Jelektronnoe modelirovanie [Electronic Modeling], 2012, Vol. 34, Issue 6, pp. 17-29. (in Russian).

[25]. Sapozhnikov V.V., Sapozhnikov Vl.V., Efanov D.V. Modulo weighted codes with summation with minimum number of undetectable errors in data vectors. Jelektronnoe modelirovanie [Electronic Modeling], 2017, Vol. 39, Issue 4 (in Russian).

[26]. Sapozhnikov V.V., Sapozhnikov Vl.V., Efanov D.V., Kotenko A.G. Modulo Codes with Summation of Weighted Transitions With Natural Number Sequence of Weights. SPIIRAS Proceedings, 2017, Issue 1, pp. 137-164, doi: 10.15622/SP.50.6.

[27]. Lala P.K. Self-Checking and Fault-Tolerant Digital Design. San Francisco: Morgan Kaufmann Publishers, 2001, 216 p.

[28]. Hamming R.W. Error Detecting and Correcting Codes. Bell System Technical Journal, 1950, 29 (2), pp. 147-160, MR0035935.

[29]. Sapozhnikov V., Sapozhnikov Vl., Efanov D., Dmitriev V. Weighted Sum Code Without Carries – is an Optimum Code with Detection of Any Double Errors in Data Vectors. Proceedings of 14th IEEE East-West Design & Test Symposium

(EWDTS`2016), Yerevan, Armenia, October 14-17, 2016, pp. 134-141, doi: 10.1109/EWDTS.2016.7807686.

[30]. Sapozhnikov V., Sapozhnikov Vl., Efanov D., Dmitriev V. New Sum Code for Effective Detection of Double Errors in Data Vectors. Proceedings of 13th IEEE East-West Design & Test Symposium (EWDTS`2015), Batumi, Georgia, September 26-29, 2015, pp. 154-159, doi: 10.1109/EWDTS.2015.7493123.

[31]. Sapozhnikov V.V., Sapozhnikov Vl.V., Efanov D.V., Dmitriev V.V. Code with summation of weighted data bits without transitions within concurrent error detection systems. Avtomatika na transporte [Automation on transport], 2017, Vol. 3, Issue 3, pp. 414-433 (in Russian).

[32]. Ghosh S., Basu S., Touba N.A. Synthesis of Low Power CED Circuits Based on Parity Codes, Proceedings of 23rd IEEE VLSI Test Symposium (VTS'05), 2005, pp. 315-320.

[33]. Aksjonova G.P. On functional diagnosis of discrete devices under imperfect data processing conditions. Problemy upravlenija [Control Sciences], 2008, Issue 5, pp. 62-66 (in Russian).

Синтез частично программируемых схем, ориентированный на маскирование вредоносных подсхем (Trojan Circuits)

А.Ю. Матросова <mau11@yandex.ru>
С.А. Останин <sergeiostanin@yandex.ru>
Е.А. Николаева <nikolaeve-ea@yandex.ru>
Национальный исследовательский Томский государственный университет, 634050, Россия, Томск, пр. Ленина, д. 36

Аннотация. При синтезе современных интегральных схем разработчики все чаще прибегают к услугам сторонних фирм для реализации тех или иных компонент системы (Intellectual Property cores, перепрограммируемых компонент на базе FPGA и т.д.) с целью снижения ее стоимости. В компонентах, изготовленных сторонними фирмами, могут быть спрятаны вредоносные подсхемы (Trojan circuits) с целью разрушения системы или извлечения из нее конфиденциальной информации. Trojan Circuits (TCs) обычно действуют в ситуациях, которые возникают в работающей системе чрезвычайно редко, поэтому они не обнаружимы ни в процессе верификации системы, ни в процессе ее тестирования. В работе предлагается подход к проектированию частично программируемых схем из вентилей, программируемых блоков памяти (LUTs) и программируемых мультиплексоров (MUXs), ориентированный на маскирование TCs. Такой подход к синтезу позволяет либо замаскировать действие TC в случае ее обнаружения, либо получить схему, в которой эффективное введение TCs становится невозможным. Предложен способ перепрограммирования блоков памяти LUTs для маскирования TC. Сформулировано требование к замещающей функции, поступающей на свободный вход программируемого блока, основанное на анализе частичных функций внутренних полюсов комбинационной схемы. Построение частичных функций выполняется с использованием операций над Reduced Ordered Binary Decision Diagrams (ROBDD-графами), строящимися для фрагментов схемы. Операции характеризуются полиномиальной сложностью.

Ключевые слова: частично программируемые схемы; вредоносные схемы (Trojan Circuits); частичные булевы функции; Reduced Ordered Binary Decision Diagrams (ROBDD-графы).

DOI: 10.15514/ISPRAS-2017-29(5)-4

Для цитирования: Матросова А.Ю., Останин С.А., Николаева Е.А. Синтез частично программируемых схем, ориентированный на маскирование вредоносных подсхем (Trojan Circuits). Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 61-74. DOI10.15514/ISPRAS-2017-29(5)-4

1. Введение

Вредоносные подсхемы (Trojan Circuits) могут вводиться в компоненты интегральной схемы с целью разрушения схемы или извлечения из нее конфиденциальной информации. Trojan Circuits (TCs) обычно действуют в ситуациях, которые возникают в условиях функционирования схемы чрезвычайно редко, поэтому они не обнаружимы ни в процессе верификации, ни в процессе тестирования схемы [1, 2]. Вредоносная подсхема (ТС) состоит из двух частей. Триггерная подсхема (Trojan trigger) включается при поступлении на ее входы определенной комбинации значений сигналов. Вторая часть подсхемы (Trojan payload) является исполнительным устройством, включаемым триггерной подсхемой, которое может либо разрушить работу схемы, либо извлечь из нее секретную информацию. Такие вредоносные подсхемы необходимо обнаруживать, и, по возможности, нейтрализовать их действие.

Рассматривается проблема маскирования вредоносных подсхем (Trojan Circuits) в логических схемах, состоящих из вентилей, путем введения в нее программируемых блоков памяти LUTs (Look up Tables) и программируемых мультиплексоров (MUXs). В LUT могут быть свободные входы (в данной работе допускается один свободный вход), которые можно использовать для коррекции схемы с целью маскирования в ней воздействия вредоносной подсхемы. Исследуются возможности использования свободного входа в условиях перепрограммирования LUT и MUXs либо для маскирования ТС в случае ее обнаружения (в этой ситуации перепрограммируется только один соответствующий LUT), либо для перепрограммирования нескольких LUTs (не обязательно всех) таким образом, что включение в схему вредоносных подсхем оказывается неперспективным. Здесь речь идет о превращении исходной логической схемы из вентилей в схему, защищенную от вредоносных подсхем. Это значит, что при сохранении спецификации схемы введение в нее TCs оказывается неэффективным – они с большой вероятностью обнаруживаются в процессе верификации схемы или в результате ее тестирования. Типы вредоносных подсхем и способы их включения в схему могут быть различными. В работе для удобства рассматриваются TCs, вводимые в линию связи между элементами схемы, так что значение на выходе этой линии в условиях активизации входа ТС заменяется противоположным. Однако предлагаемый подход годится для произвольного включения ТС, важно лишь, что выход вредоносной подсхемы подключен к некоторой линии. Активизация ТС обеспечивается достижением соответствующего входного состояния комбинационной схемы (полного

состояния, если речь идет о комбинационном эквиваленте последовательностной схемы). Перепрограммируемые мультиплексоры применяются с целью маскирования нескольких TCs одним и тем же перепрограммируемым LUT.

Предполагается, что исходная схема состоит только из вентилей и реализует заданную спецификацию – систему полностью определенных булевых функций. Требуется покрыть некоторые ее подсхемы программируемыми блоками памяти (LUTs) таким образом, чтобы обеспечить либо маскирование TC в случае ее обнаружения, либо сделать схему более защищенной от вредоносных подсхем. Задача сводится к маскированию линий в схеме, константные неисправности которой трудно обнаружимы, то есть множества тестовых наборов для таких неисправностей малы и не превышают некоторого заданного порога. Предполагается, что именно такие линии удобно использовать для подключения к ним TCs.

В работах [3, 4] схема сначала строится из вентилей. Затем некоторые ее подсхемы покрываются программируемыми блоками (LUTs), так что один вход LUTs, не обязательно всех, остается неиспользованным. Выделяется множество линий, константные неисправности которых необходимо замаскировать одним из двух способов. Возможность маскирования конкретным LUT конкретной линии сводится к определению выполнимости соответствующей Quantified Boolean Formula (QBF). Речь идет о квантифицированной конъюнктивной нормальной форме. Используются различные решатели (QBF-solvers), автоматизирующие процесс. Проблема выполнимости такой формулы относится к PSPACE-полным. Зарубежные исследователи отмечают, что решатели для QBF формулы работают гораздо медленнее, чем решатели, анализирующие конъюнктивную нормальную форму (КНФ) на выполнимость. В случае возможности маскирования определяется способ перепрограммирования маскирующего LUT. В проектируемой схеме проводится дополнительная линия, связывающая свободный вход LUT с выходом элемента схемы (вентиля или другого LUT), который совместно с перепрограммируемым LUT может маскировать неисправность соответствующей линии.

Недостатком такого подхода является, на наш взгляд, отсутствие стратегии выбора подсхем для покрытия их программируемыми блоками (LUTs) и отсутствие формальных критериев возможностей маскирования. Данная работа ориентирована на преодоление этих недостатков. С этой целью используются и анализируются частичные функции, сопоставляемые внутренним полюсам и линиям схемы. Частичные функции вычисляются путем выполнения операций над ROBDD-графами, построенными для фрагментов заданной комбинационной схемы. Такие операции, как известно, характеризуются полиномиальной сложностью. Использование QBF solvers не требуется.

Мы предлагаем сначала найти линии в схеме, неисправности которых трудно обнаружимы. В эти линии могут быть включены вредоносные подсхемы. Поведение активированной ТС аналогично проявлению константной неисправности линии на подмножестве входных наборов схемы (возможно, только на одном), которые не обязательно являются тестовыми наборами этой неисправности в условиях отсутствия в схеме вредоносных подсхем. Выделенные линии маскируются LUTs с фиксированным числом входов, один из которых является свободным. Если маскируется одна из линий множества при подключении к ней ТС, то действие подключенной к ней вредоносной подсхемы нейтрализуется. В этой ситуации соответствующий программируемый блок (LUT) перепрограммируется. Если маскируются все или почти все линий выделенного подмножества, то в результате строится схема, устойчивая относительно введения в нее вредоносных подсхем.

Предлагаемый подход основан на вычислении частичных булевых функций внутренних полюсов и линий комбинационной схемы с помощью операций над ROBDD-графами, построенными для ее подсхем.

2. Постановка задачи

Рассматривается комбинационная схема C (комбинационная составляющая последовательностной схемы), состоящая из вентилей. Используются способы (рис. 1, рис. 3) маскирования одиночных константных неисправностей на линиях связей между логическими элементами с помощью LUTs и MUXs, предложенные в работах [3, 4]. Число входов программируемого блока (LUT) фиксировано. Один из входов может быть свободным, не используемым при покрытии вентилей схемы C. Для заданного множества линий требуется выполнить покрытие фрагментов схемы C из вентилей программируемыми блоками, так чтобы при последующем их перепрограммировании замаскировать как можно большее количество линий, к которым возможно подключение вредоносных подсхем (TCs). Договоримся в дальнейшем схему из вентилей и схему, в которой некоторые подсхемы покрыты программируемыми блоками, обозначать одним и тем же символом C.

Будем иметь в виду, что каждому внутреннему полюсу v комбинационной схемы (схема реализует функцию f, являющуюся спецификацией) в общем случае сопоставляется частичная функция f_v от входных переменных схемы C, представляемая множествами $M_1(f_v)$ ($M_0(f_v)$) нулевых и единичных наборов значений входных переменных. Полностью определенную функцию внутреннего полюса v будем обозначать символом $f(v)$. Полюс v является выходом подсхемы C_v, входы этой подсхемы совпадают с входами схемы C. Частичность функции подсхемы C_v возникает за счет окружения ее элементами схемы C.

Пусть C является одно выходной комбинационной схемой. Рассмотрим способ маскирования, представленный на рис. 1.

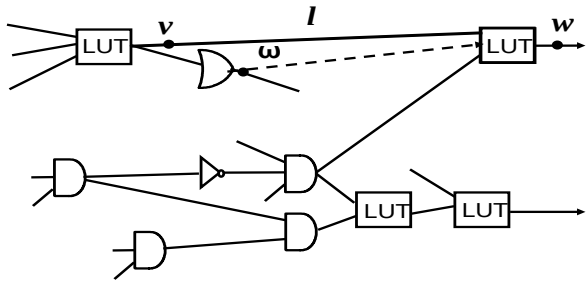


Рис. 1. Маскирование линии l, подключенной к входу LUT
Fig. 1. Masking line l that is LUT input

Полюс v соединен линией l с входом u_i программируемого блока LUT, покрывающего подсхему C_{LUT} из вентилей. На рис. 1 линия l выделена жирным. Входы подсхемы C_{LUT} являются либо входными, либо внутренними переменными схемы C. Число входных переменных LUT фиксировано, одна из переменных свободна, а функция от оставшихся переменных реализует подсхему из вентилей C_{LUT}, покрывающую соответствующий фрагмент из вентилей схемы C. Пусть в линию l включена вредоносная подсхема. Будем маскировать ее, используя свободный вход LUT. В дальнейшем этот LUT будем называть корректирующим. Обозначим его выход символом w. Отметим, что позиция пунктирной линии на рис. 1 заранее не определена, она может соединять свободный вход LUT с выходом другого элемента схемы, при выполнении определенных условий.

Обозначим символом m число входов корректирующего программируемого блока (LUT), тогда $(m - 1)$ – число используемых входов в предположении, что в схеме отсутствуют вредоносные подсхемы. Среди используемых входов находится вход u_i, соединенный линией l с полюсом v. Для определенности будем считать, что свободный вход имеет номер m. Пусть в линию l включена вредоносная подсхема. Она изменяет подсхему C_w и функцию, вычисленную по структуре этой подсхемы. Напомним, что входами подсхемы C_w являются входы схемы C. Частичная функция f_w^*, сопоставляемая полюсу w в присутствии ТС, также изменяется, поскольку вредоносная подсхема в условиях активации изменяет реакцию схемы на некоторых ее входных наборах, возможно, только на одном. В этом случае необходимо маскировать линию l за счет использования свободного входа корректирующего LUT. В дальнейшем покажем, что маскирование линии l способом, представленным на рис. 3, аналогично. Сначала введем ряд необходимых понятий.

3. О построении частичной функции и ее реализации

Выделим в схеме C внутренний полюс v, являющийся выходом вентиля схемы или некоторого LUT и сопоставляемый исходящей из него линии l. Построим

по схеме C подсхему C_l, объявив линию l входом этой подсхемы наряду с переменными $x_1,\ldots, x_n$. Схема C_l получается из схемы C обрывом линии l и устранением всех элементов схемы C, связанных с линией l и не связанных с выходом схемы C. Сопоставим подсхеме C_l ROBDD-граф $R(C_l)$ от переменных $x_1, \ldots, x_n, l$. В графе $R(C_l)$ переменная l выбирается первой при разложении Шеннона с целью построения этого графа.

Будем иметь в виду, что полюсу v, из которого исходит линия l, сопоставляется полностью определенная функция $f(v)$, реализуемая подсхемой C_v (она представляется ROBDD-графом $R(C_v)$), выходом которой является полюс v, а входами – переменные $x_1, \ldots, x_n$. Этому полюсу сопоставляется также частичная функция $\mu(x_1, \ldots, x_n)$, реализуемая на полюсе v от тех же входных переменных, в условиях, когда подсхема C_v является частью схемы C. Частичная функция определена на некоторых единичных $M_1(f_v)$ (нулевых $M_0(f_v)$) наборах полностью определенной функции $f(v)$. Только на этих наборах изменение значения функции $f(v)$ влияет на значение функции f, реализуемой схемой C.

Множества единичных и нулевых наборов частичной функции предложено представлять двумя ROBDD-графами $R_1(v)$ и $R_0(v)$. Построение частичной функции сводится к поиску всех тестовых наборов для константных неисправностей полюса v. Множество всех тестовых наборов для неисправности константа 0 есть множество $M_1(f_v)$, а множество всех тестовых наборов для неисправности константа 1 есть множество $M_0(f_v)$ частичной функции $\mu(x_1, \ldots, x_n)$. Метод построения ROBDD-графов для множеств единичных и нулевых наборов частичной функции внутреннего полюса схемы представлен в работах [5, 6].

Обозначим символами $R_l, R_{\bar{l}}$ ROBDD-графы, полученные из графа $R(C_l)$ следующим образом. Корнем графа R_l, является вершина, в которую заходит дуга, сопоставляемая переменной l графа $R(C_l)$, а корнем графа $R_{\bar{l}}$ является вершина, в которую заходит дуга, сопоставляемая инверсии этой переменной.

Будем иметь в виду, что ROBDD-граф $\overline{R}$ получается из ROBDD-графа R переименованием терминальных вершин: 1 – терминальная вершина становится 0 – терминальной вершиной и наоборот.

Тогда граф $R_1(l)$, представляющий множество тестовых наборов для неисправности константа 1 на линии l, вычисляется по формуле:

$$R_1(l) = (R_l \wedge \overline{R_{\bar{l}}} \vee \overline{R_l} \wedge R_{\bar{l}}) \wedge \overline{R(C_v)},$$

а граф $R_0(l)$, представляющий множество тестовых наборов для неисправности константа 0 на линии l, вычисляется по формуле:

$$R_1(l) = (R_l \wedge \overline{R_{\bar{l}}} \vee \overline{R_l} \wedge R_{\bar{l}}) \wedge R(C_v).$$

Оба графа задают частичную функцию линии l. Если полюс v не является точкой ветвления, то частичные функции линии l и полюса v совпадают.

Рассмотрим частичную функцию f_1 и полностью определенную функцию f_2, представленные парами множеств единичных и нулевых наборов значений своих переменных: $M_1(f_1)$, $M_0(f_1)$; $M_1(f_2)$, $M_0(f_2)$. Будем говорить, что полностью определенная функция f_2 реализует частичную функцию f_1, если выполняются условие: пересечения множеств $M_1(f_1)$, $M_0(f_2)$ и множеств $M_0(f_1)$, $M_1(f_2)$ пусты. Это значит, что $M_1(f_2)$ содержит $M_1(f_1)$ и $M_0(f_2)$ содержит $M_0(f_1)$.

Из определения следует, что полностью определенная функция $f(v)$ является реализацией функции $\mu(x_1, \ldots, x_n)$.

Утверждение 1. Подстановка вместо переменной v любой реализации функции $\mu(x_1, \ldots, x_n)$ сохраняет функцию f схемы C.

Доказательство следует из определения реализации частичной функции.

Частичная функция f_2 поглощает частичную функцию f_1, если выполняются следующие условия:

1) $M_1(f_2)$ содержит $M_1(f_1)$ и $M_0(f_2)$ содержит $M_0(f_1)$;

2) пересечение множеств $M_1(f_1)$, $M_0(f_2)$ и множеств $M_0(f_1)$, $M_1(f_2)$ пусто.

Отношение поглощения частичных функций иллюстрируется на рис. 2.

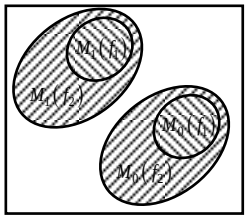


Рис. 2. Функция f_2 поглощает f_1
Fig. 2. Function f_2 covers f_1

Может оказаться, что при покрытии схемы из вентилей программируемыми блоками нет возможности подключить линию l к входу LUT. Тогда воспользуемся способом маскирования (рис. 3), предложенным в работах [3, 4]. При таком способе маскирования линия l лежит на пути, проходящем через элементы схемы, связывающем эту линию с входом u_i корректирующего LUT. На рис. 3 неисправная линия выделена жирным шрифтом.

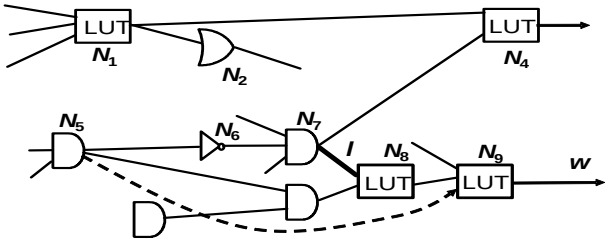


Рис. 3. Маскирование линии l, соединенной через элементы пути с входом корректирующего LUT
Fig. 3. Masking line l that is connected by path with input of correcting LUT

Для обоих способов маскирования линии l (рис. 1, рис. 3) справедливо следующее утверждение.

Утверждение 2. Спецификация схемы C (функция f) сохраняется если и только если частичная функция f_w^* поглощает частичную функцию f_w.

Доказательство очевидно, и следует из определения частичной функции.

Коррекцию будем выполнять, следуя утверждениям 1 и 2, причем, коррекция не должна менять функций элементов схемы C, входящих в окружение подсхемы C_w. Будем иметь в виду, что способы маскирования, представленные на рис. 1, 3, удовлетворяют этому условию.

4. Маскирование неисправности

Для способа, предложенного на рис. 1, и способа, предложенного на рис. 3, будем записывать в корректирующий программируемый блок единичные наборы полностью определенной функции, реализуемой подсхемой из вентилей, покрытой этим блоком и зависящей от его входных переменных, а именно от его (m – 1) переменной. Каждому единичному набору сопоставляется два набора в пространстве m переменных: один с единичным значением переменной u_m, другой – с нулевым значением этой переменной.

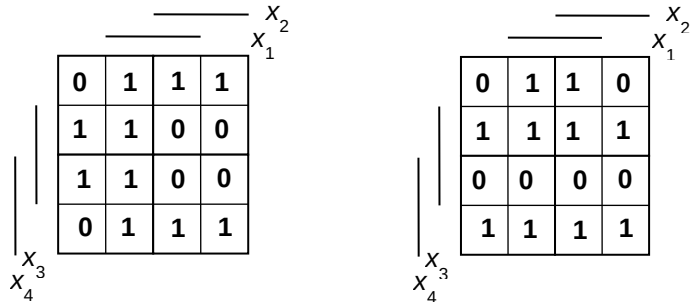
При перепрограммировании LUT формируем функцию

$$f_{LUT}(u_i = 1) \wedge u_m \vee f_{LUT}(u_i = 0) \wedge \overline{u_m}$$

из единичных наборов функции корректирующего LUT. Эти наборы получены непосредственно по структуре подсхемы, покрытой корректирующим LUT, и теперь представляют функцию этого LUT в пространстве $u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_m$ его входных переменных. Она отличается от исходной функции LUT тем, что в ней переменная u_i заменена переменной u_m, и переменная u_m, (как прежде переменная u_i) теперь является существенной. С целью маскирования ТС делаем переменную u_i несущественной. Это значит, что каждому единичному набору функции $f_{LUT}(u_i = 1) \wedge u_m \vee f_{LUT}(u_i = 0) \wedge \overline{u_m}$ необходимо сопоставить два набора в пространстве m переменных: один с единичным значением переменной u_i, а другой – с нулевым значением этой переменной. Тогда ТС, подключенная к входу u_i, не может изменить корректного поведения схемы C.

На рис. 4 приведен пример исходной функции корректирующего LUT (рис. 4а) и функции, представляющей результат коррекции (рис. 4б). Здесь вход u_i сопоставляется переменной x_2, а переменная x_4 соответствует свободному входу корректирующего LUT.

В результате перепрограммирования корректирующий LUT реализует функцию от $(m-1)$ переменной, отличающуюся от его прежней функции тем, что выполнена замена переменных (u_i заменена на u_m). Далее вместо u_m (рис. 1) подставляется полностью определенная функция, реализующая частичную функцию $\mu(x_1, \ldots, x_n)$ полюса v. Полностью определенная функция реализуется на полюсе ω.



а) *б)*

Рис. 4. Перепрограммирование LUT
Fig. 4. Reprogramming LUT

Поскольку функция корректирующего LUT получена из исходной заменой переменной u_i на u_m, а полностью определенная функция, сопоставляемая u_m, соединенным с полюсом ω, реализует частичную функцию, сопоставляемую переменной u_i, то на основании выше приведенных утверждений заключаем, что схема C реализует в результате коррекции ту же функцию f, то есть спецификация схемы C сохраняется. Это значит, что подключение ТС к линии l не изменяет функцию f.

Заметим, что входу u_i в конструкции, представленной на рис. 3, не обязательно сопоставляется слабо определенная частичная функция, как в случае, когда вход u_i инцидентен линии, неисправность которой трудно обнаружима (рис. 1).

5. Синтез схемы

Покрытие исходной вентильной схемы программируемыми блоками памяти (LUTs) с целью маскирования вредоносных подсхем (TCs) выполняем следующим образом. Определяем множество L линий, неисправности которых трудно обнаружимы. Двигаемся в порядке, например, не возрастания мощности тестовых наборов частичных функций. Мощности единичных и нулевых наборов частичной функции линии не должны превышать заданного

порога. Находим линию *l*, сопоставляемую трудно обнаружимым константным неисправностям 0, 1, пытаемся покрыть ее LUT, так что покрываемая линия оказывается внутри покрытой им подсхемы и не является входом LUT. Этот LUT не нуждается в коррекции, а покрытая им линия вычеркивается из списка.

Исчерпав такие возможности, пытаемся покрыть подходящие подсхемы программируемыми блоками (LUTs), так что очередная линия из списка является входом u_i некоторого LUT (рис. 1). Этот LUT корректируется описанным выше способом, если удается найти полюс ω в схеме *C*, такой что его полностью определенная функция реализует частичную функцию полюса *v* (входа u_i). Формируется дополнительная (обозначенная пунктиром) линия, связывающая полюс ω с входом u_m.

В случае, если возможно маскирование нескольких линий одним и тем же корректирующим LUT, воспользуемся мультиплексором, как это предложено в работах [3, 4].

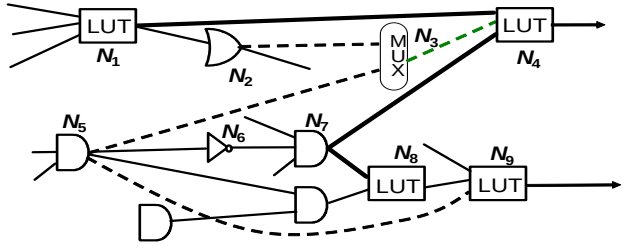


Рис. 5. Маскирование двух линий с помощью мультиплексора
Fig. 5. Two line masking with using MUX

Продолжаем процедуру, до тех пор, пока это возможно. Далее пытаемся маскировать оставшиеся линии, следуя конструкции, представленной на рис. 3, формируя дополнительные линии. В результате получим схему из вентилей, LUTs и MUXs, в которой действия вредоносных подсхем могут быть замаскированы. Имеется возможность либо замаскировать действие ТС в случае ее обнаружения, перепрограммируя соответствующий LUT, либо получить схему, в которой эффективное введение TCs становится невозможным за счет перепрограммирования соответствующих LUTs.

Для многовыходной схемы *C* предлагается вычислять частичные функции для линии каждой из подсхем, которым она принадлежит. Для каждой из линий получаем систему частичных функций. В результате находим множество линий, неисправности которых трудно обнаружимы. Во множество включаем линию, для которой неисправность трудно обнаружима на каждом из выходов схемы, связанном с этой линией. Аналогичным образом вычисляем частичные функции для внутренних полюсов схем, если они являются точками

ветвления. Для остальных полюсов их частичные функции совпадают с частичными функциями исходящих из них линий.

В качестве полюса ω для очередной линии l выбираем полюс, такой что его полностью определенная функция реализует каждую из частичных функций системы, сопоставляемых полюсу v.

Предварительные эксперименты на контрольных примерах для многовыходных схем показали, что существуют пары полюсов, такие, что частичные функции одного полюса реализуется полностью определенной функцией другого полюса.

6. Заключение

Предложен метод синтеза частично программируемых комбинационных схем (комбинационных составляющих последовательностных схем) из вентилей, программируемых блоков памяти (LUTs) и программируемых мультиплексоров (MUXs), ориентированный на маскирование вредоносных подсхем (TCs) в условиях наличия свободного входа у программируемых блоков памяти. Наряду с маскированием TC за счет перепрограммирования соответствующего LUT в случае обнаружения вредоносной подсхемы предлагается перепрограммирование нескольких LUT таким образом, что включение вредоносных подсхем оказывается нецелесообразным: включение их в линии схемы может с большой вероятностью обнаружено либо в процессе верификации, либо в процессе тестирования. Сформулировано требование к замещающей функции, поступающей на свободный вход программируемого блока, основанное на анализе частичных функций внутренних полюсов комбинационной схемы. Построение частичных функций выполняется с использованием операций над ROBDD-графами, строящимися для фрагментов комбинационной схемы.

Список литературы

[1]. Karri R., Rajendran J., Rosenfeld K., Tehranipoor M. Trustworthy Hardware: Identifying and Classifying Hardware Trojans. Computer, vol. 43, no. 10, 2010, pp. 39-46. DOI: 10.1109/MC.2010.299.

[2]. Yoshimura M., Bouyashiki T., Hosokawa T. A sequence Generation Method to detect Hardware Trojan Circuits. The 16-th IEEE Workshop on RTL and High Level Testing, Proceeding, 2015, pp. 84-89.

[3]. Yamashita S., Yoshida H., Fujita M. Increasing yield using partially-programmable circuits. Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), Proceeding, 2010, pp. 237-242.

[4]. Jo S., Matsumoto T., Fujita M. SAT-based automatic rectification and debugging of combinational circuits with LUT insertions. IEEE Asian Test Symposium, Proceeding, 2012, pp. 19-24. DOI: 10.1109/ATS.2012.55.

[5]. Матросова А.Ю., Останин С.А., Бухаров А.В., Кириенко И.Е. Поиск всех тестовых наборов для неисправности логической схемы и представление их ROBDD-графом. Вестн. Том. гос. ун-та. УВТиИ. № 2(27), 2014, стр. 82-89.

[6]. Matrosova A.Yu., Ostanin S.A., Kirienko I.E. Generating all test patterns for stuck-at faults at a gate pole and their connection with the incompletely specified Boolean function of the corresponding subcircuit. The 14th Biennial Baltic Electronics Conference, Proceeding, 2014, pp. 85-88. DOI: 10.1109/BEC.2014.7320562.

Partially Programmable Circuit Design Oriented to masking Trojan Circuits

A.Yu. Matrosova <mau11@yandex.ru>
S.A. Ostanin <sergeiostanin@yandex.ru>
E.A. Nikolaeva <nikolaeve-ea@yandex.ru>
National Research Tomsk State University
36, Lenin Avenue, Tomsk, 634050, Russia

Abstract. The enhanced utilization of outsourcing services for a part of VLSIs (Intellectual Property cores, reprogramming components based on FPGA and so on) to cut VLSI cost increases risk of inserting Trojan Circuits (TCs) that may destroy VLSI or provide leakage of confidential information. TCs as a rule act in rare operation situations, therefore they are not detectable neither during VLSI verification nor VLSI testing. The approach to partially programmable circuit design from gates, programmable LUTs and MUXs oriented to masking TCs is suggested. The approach allows getting a circuit that masks TC when it has been found or deriving a circuit that is tolerant to TCs actions. The method of reprogramming LUTs for masking TCs is developed. The condition of replacing a function corresponding to free LUT input is formulated. It is based on using incompletely specified Boolean functions of internal nodes of the circuit. The functions are obtained with using operations on ROBDDs corresponding to the circuit fragments. The operations have a polynomial complexity.

Key words: Partially Programmable Circuits, Trojan Circuits (TCs), Incompletely Specified Boolean Functions, ROBDDs.

DOI: 10.15514/ISPRAS-2017-29(5)-4

For citation: Matrosova A.Yu., Ostanin S.A., Nikolaeva E.A. Partially Programmable Circuit Design Oriented to masking Trojan Circuits. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017, pp. 61-74 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-4

References

[1]. Karri R., Rajendran J., Rosenfeld K., Tehranipoor M. Trustworthy Hardware: Identifying and Classifying Hardware Trojans. Computer, vol. 43, no. 10, 2010, pp. 39-46. DOI: 10.1109/MC.2010.299.

[2]. Yoshimura M., Bouyashiki T., Hosokawa T. A sequence Generation Method to detect Hardware Trojan Circuits. The 16-th IEEE Workshop on RTL and High Level Testing, Proceeding, 2015, pp. 84-89.

[3]. Yamashita S., Yoshida H., Fujita M. Increasing yield using partially-programmable circuits. Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), Proceeding, 2010, pp. 237-242.

[4]. Jo S., Matsumoto T., Fujita M. SAT-based automatic rectification and debugging of combinational circuits with LUT insertions. IEEE Asian Test Symposium, Proceeding, 2012, pp. 19-24. DOI: 10.1109/ATS.2012.55.

[5]. Matrosova A.Yu., Ostanin S.A., Buharov A.V., Kirienko I.E. Generating all test patterns for a given stuck-at fault of a logical circuit and its ROBDD implementation. Tomsk State University Journal of Control and Computer Science [Vestn. Tom. gos. un-ta. UVTiI], № 2(27), 2014, pp. 82-89 (in Russian).

[6]. Matrosova A.Yu., Ostanin S.A., Kirienko I.E. Generating all test patterns for stuck-at faults at a gate pole and their connection with the incompletely specified Boolean function of the corresponding subcircuit. The 14th Biennial Baltic Electronics Conference, Proceeding, 2014, pp. 85-88. DOI: 10.1109/BEC.2014.7320562.

A Flat Chart Technique for Embedded OS Testing[1]

V.V. Nikiforov<nik@ iias.spb.su>
S.N. Baranov<snbaranov@ iias.spb.su>
St. Petersburg Institute for Informatics and Automation
of the Russian Academy of Sciences,
39, 14 liniya, St. Petersburg, 199178, Russia

Abstract. Modern automatic devices are more and more equipped with microcontroller units. The logic of work of the automatic equipment is supported by a number of various embedded software applications, which run under an embedded real-time operating system (OS). The OS reliability is extremely important for correct functionality of the whole automatic system. Therefore, the embedded OS should be tested thoroughly with an appropriate automated test suite. Such test suite for testing of an embedded OS is usually organized as a set of multi-task test applications to be executed in a data-driven manner. The paper features a special language to define the respective testing task logic and the concept of flat charts to efficiently perform an embedded OS execution-based testing. To avoid heavy interpreting of text strings during the test run, the respective test presentation is pre-processed in order to convert the initial string form into a regular array form and thus to increase its efficiency.

Keywords: Embedded Applications; Operating Systems; Software Testing; Real-Time Systems.

DOI:10.15514/ISPRAS-2017-29(5)-5

For citation: Nikiforov V.V., Baranov S.N. A Flat Chart Technique for Embedded OS Testing. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017, pp. 75-92. DOI: 10.15514/ISPRAS-2017-29(5)-5

1. Introduction

Software applications, which control various automatic devices, are usually built as a set of two kinds of sequential executing threads: *tasks* and *interrupt service routines* (*ISRs*). Coordination of execution of these threads is realized by the kernel of the embedded real-time operating system (OS). OS reliability is extremely important for correct functioning of the automatic technical device under software

[1] This paper is an extended version of a presentation at the Industrial track poster session of the 29th IFIP International Conference on Testing Software and Systems (ICTSS-2017), St. Petersburg, Russia, October 9-11, 2017.

control.

The variety of requirements for such an OS grows along with the variety of technical devices, for which embedded systems are designed, especially for devices built on the basis of *microcontroller units* (MCUs). Each OS for an MCU should be tested thoroughly to avoid a crash of an embedded application. Verification of embedded real-time software is a well-known problem [1], [2]. Thorough execution-based testing [3] of an embedded OS requires significant effort along two axes: full-bodied *test suite design* and *test suite execution*.

Effort reduction for test execution may be achieved by designing a highly automated test suite. Effort reduction for design of such a test suite may be achieved through efficient testing techniques, languages, and tools.

The paper describes a special language to define the testing task logic based on the concept of flat charts to efficiently run embedded OS execution-based testing.

2. Approach to Testing an Embedded OS

Usually, an embedded OS provides static and dynamic services for applications to run on top of this OS. *Static services* are used to specify static configuration features of the application: the set of its tasks and ISRs, the subset of the used OS functions, basic task properties (e.g., task priorities), static resource distribution among the application tasks (allocation of memory, stacks and other special structures). *Dynamic services* may be further split into basic and additional ones.

Basic dynamic services ensure:

- run-time distribution of resources among the threads (memory, special structures, processor time);
- exchange of data and signals among tasks;
- passing data and signals from ISRs to tasks;
- error (fault, exception) handling which provides data on an abnormal situation in the application.

Additional dynamic services support specific functions:

- run-time generation of threads, tasks, and ISRs;
- run-time updating of the basic task properties (e.g., the task priority);
- run-time stack reallocation;
- mathematical calculations, string processing, etc.

The problem of basic dynamic services testing will be considered in this paper from two points of view:

- functional testing – checking the correctness of the basic OS directives execution logic; and
- timing testing – measurement of time intervals required for execution of basic OS directives.

Functional testing is aimed at checking the correctness of the OS behavior through

finding defects in:

- execution of basic OS directives invoked from application tasks and ISRs;
- processor switching among threads;
- data and signal transactions;
- error handling routines.

Timing testing is aimed at obtaining the following timing data on OS execution:

- execution time of a particular OS directive (local time measurement);
- total execution time of the whole application (global time measurement);
- time interval between the moments when the interrupt occurred and when a respective ISR started this interrupt processing (latency measurement).

The described flat chart technique is aimed at both kinds of testing of embedded OS basic dynamic services, functional and timing, through a unified approach.

2.1 Testing Rules

The following generally established testing rules [4], [5] are usually observed for embedded OS testing:

- *focus* – each test should check only one OS feature under particular conditions with only two possible outcomes: pass or fail;
- *repeatability* – the test behavior should be the same at each execution;
- *non-interference* – the test should not intrude into OS functioning (no direct access to OS variables, command lines, or structures), the test uses the OS services as a regular application;
- *black-box approach* – each test should be developed with no knowledge or assumptions about the OS inner structures, with information at the user's level only.

The above rules for focus and repeatability impose structural constraints for tests because with these rules each single test should be a multi-task application which starts from a known inital state. The most reliable way to bring the system under test into this state is system restart with re-initialization. Therefore, the size of each test for an embedded OS is that of a multi-task application, and the test execution time includes the time required for system initialization.

The repeatability rule requires special solutions to ensure it. Regular real-time applications running under an embedded OS usually lack repeatability: their tasks and ISRs work *asynchronously* without any pre-defined order. Test applications should be built in such a way as to avoid such indetermination.

2.2 Repeatability of Testing

The test scheme in Fig. 1 shows how variations of the test behavior may occur. The test *DelayCoEnd* below is related to the most basic service of an embedded OS – the *delay service.* The operator *Delay*(*N*) holds up the task execution for *N* ticks where

'tick' is an atomic time interval, usually part of a millisecond. The test *DelayCoEnd* checks the correctness of OS behavior when delay intervals of two tasks come to a completion simultaneously. The scheme uses a C-like notation. A digit in the name of the task starting point corresponds to the task priority. The task that starts at the point *Task_1* has higher priority than the task labeled *Task_2*.

```
// -----  "DelayCoEnd" test application  ------
  Task_1:                          |      Task_2:
/* Step_01 */   Delay(50);         | /* Step_02 */ for(i = 0;  i == WAIT_CONST;  i++ );
/* Step_05 */   GlobFlag = 1;     | /* Step_03 */       Delay(30);
/* Step_06 */   TaskEnd( );       | /* Step_07 */       GlobFlag = 1;
                                   | /* Step_08 */     TaskEnd( );
  Task_3:
/* Step_04 */   for(GlobFlag = 0; GlobFlag == 0 ; );
/* Step_09 */   End_of_Test( );
```

Fig. 1. Variations of the test behavior

Step numbers shown in comments indicate the expected order of execution. At *Step_01* execution of *Task_1* is held up for 50 tics, the processor switches to *Task_2*, and the 'for'-operator of *Task_2* (*Step_02*) starts. The value of *WAIT_CONST* should ensure a simultaneous completion of the two delay intervals. While both intervals have not been completed, the 'for'-operator of *Task_3* (*Step_04*) is executed. The idea of the whole scheme is in selection of a *WAIT_CONST* value which ensures simultaneous completion of both delay intervals, so that if OS correctly handles this, then the actual sequence of steps follows that of the step numbers (additional steps may appear between them).

For automatic registration of the sequence of executed steps, the *Trace*(*i*) operator should be substituted for each comment *Step_i*. The procedure *Trace*(*i*) checks whether its parameter *i* corresponds to the current step in the expected step sequence and signals an error otherwise. This trace operator should be inserted at each point where the execution sequence should be checked.

At looking at these three tasks, one may decide that the only issue for checking the correctness of the OS delay function is an appropriate selection of the *WAIT_CONST* value which ensures simultaneous completion of the two delay intervals and therefore, *DelayCoEnd* repeatability. However, this is not true at a closer consideration.

Non-repeatability of such test execution is caused by the fact, that *Step_01* may start either at the beginning of an atomic tic interval or closer to its end and the required value of *WAIT_CONST* is different for these two situations. To make the test consistently correct, the operator *Step_01* should be shifted to the end of an atomic tic by inserting an additional delay operator before *Step_01*.

The requirement for repeatability is specific for a test application, which differs

from the real one with asynchronous execution of application tasks where the order of operators from parallel tasks may vary from one execution to another.

Test applications require special efforts for strict task synchronization. As a result, the sequence of operations from parallel tasks in test execution becomes strictly determined as if it were from a sequential process.

In the listing in Fig. 2 the *flag synchronization* method with *WaitFlag*( ) and *SetFlag*() procedures ensures test repeatability:

```
void WaitFlag ( )                          {
   int i;
   for(GlobFlag = 0; GlobFlag == 0 ; )
       if (i++ > LONG_WAIT) break;     }

void  SetFlag ( )   {
     GlobFlag = 1;  }
```

Fig. 2. Flag synchronization method ensures test repeatability

Here *GlobFlag* is a global variable and *LONG_WAIT* is a constant, which limits the time of waiting to avoid an infinite execution of the loop. A simple procedure in Fig. 3 prevents any task to gain access to the processor during the time interval specified with the *CycleNum* value.

```
void HoldTime (int CycleNum)     {
   int i;
   for(i = 0;   i++;   i <  CycleNum); }
```

Fig 3. A simple procedure to prevent gaining access to the processor

3. Flat Charts

The straightforward multi-task test description presented above has a weak point. When a test designer follows the test logic step by step, his attention jumps from one task to another across the text description. In spite of a clear execution order, it is too difficult to recognize the test logic even for very short and simple tests. And this is much more difficult for tests of the length of dozens of operators and more. A more suitable form for test description, the *flat chart form*, was developed by the authors and later was improved with creation of a number of real test suites for various embedded OSs.

The simplest flat chart form is based on the following assumptions:

- repeatability of the test execution order is maintained;
- test utilities in the test application are simple and small in number;
- tests contain invocations of only OS services and test utilities;
- all tasks are generated statically;
- task priorities are static;
- no two tasks have the same priority.

A sequence of actions performed by the test application consists of two kinds of

operations: *OS service operations* (for *DelayCoEnd* these are *Delay()* and *TaskEnd()* operators) and *special testing utility operations* (like *HoldTime()*, *SetFlag()*, *WaitFlag()*, and *End_of_Test()* operators). This structure is typical for any embedded OS test suite. Each OS service and testing utility has a limited number of parameters. With such assumptions, information about each test step may be described with the data structure shown in Fig 4.

```
typedef struct Test_Step   {
        int TaskId;
        void (* UtilServ) ( );
        StepArg Arg_1;
        StepArg Arg_2;        } TestStep;
```

```
union StepArg                {
        int IntArg;
        char* StringArg;
        void (* FunArg) ( );
          ...                };
```

Fig. 4. Information about each test step

where *TaskId* identifies the task that performs the operation. The field *UtilServ* stores a pointer to a procedure which either performs some actions with the test application variables (a procedure from the test utility library), or performs an OS service call. The fields *Arg_1* and *Arg_2* are used to represent the procedural parameters of the test utility or the OS service. As the type of arguments may vary from one operator to another, a union type *StepArg* in this C-like notation is defined, where ... denotes other types of parameters used in service or utility calls.

Now the operator sequence of the test *DelayCoEnd* steps from subsection 2.2 may be described as an array of the *TestStep* type (Fig. 5; steps 01, 05, and 07 were added to ensure repeatability of the test as explained above):

```
TestStep DelayCoEnd [ ] =       {
      1,&CallDelay, 5, 0 ,                    // Step_01
            2,&WaitFlag,  0,  0,              // Step_02
      1,&SetFlag,  0,  0,                     // Step_03
      1, &CallDelay, 50,  0,                  // Step_04
            2,&HoldTime,  0,  0,              // Step_05
            2,&CallDelay, 30,  0,             // Step_06
                  3,&WaitFlag,  0,  0,        // Step_07
      1,&SetFlag,  0,  0,                     // Step_08
      1, &CallTaskEnd,  0,  0,                // Step_09
            2,&SetFlag,  0,  0,               // Step_10
            2, &CallTaskEnd,  0,  0,          // Step_11
                  3,&End_of_Test,  0,  0,     // Step_12
 0,&End_of_scheme,  0, 0        };
```

Fig. 5. Operator sequence of the test DelayCoEnd steps

where the item <0,&*End_of_scheme*,0,0 > terminates the test description. There is no task with the number 0; therefore, a zero in the *Task_Id* field means that this item does not describe an application step, but is an auxiliary one.

Representing a test scheme in form of a single entity (flat chart) is convenient for visual analysis of the test logic as well as for realization of the flat chart interpreter because this simplifies control over the correctness of the order, which steps of the application tasks are executed in.

This *DelayCoEnd* array provides a complete specification of the test application logic. Two important features in this form of test presentation are worth mentioning. First, the order of the *TestStep* structure items is that, which they should be executed in. There's no need to specify step numbers as they are determined by ordering of the *DelayCoEnd* array elements.

The second feature relates to the starting position of each line in the *DelayCoEnd* array description. The test description may be regarded as a table of 12 rows and 3 columns. Columns correspond to tasks. If the line describes an operation to be performed by *Task_i*, then its description shall start in the *i*-th column. Thus, the column order reflects the task priorities and the order of rows reflects the execution sequence.

The authors' experience with test suites design proves the efficiency of this table form called a *flat chart* for describing multi-task test applications. It turned out to be an effective tool for test logic design, understanding, and updating. Moreover, it allows for automation of test suite development for testing embedded real-time operating systems.

4. Data-Driven Test Applications

Automatic processing of flat charts is performed through a corresponding interpreter. An instance of the interpreter is initialized for each task and all such instances run concurrently. Each interpreter instance scans the flat chart specification line by line. Suppose, that each instance has its own variable

```
TestStep* CurrentStep;
```

which points to the flat chart element being analyzed or interpreted.

The *i*-th instance of the interpreter executes only those lines of the flat chart which correspond to the *i*-th task. All others lines are skipped as they are executed by other interpreter instances. When the next line for execution is found, its number is checked – it should be the first line number in the whole flat chart not yet executed by any interpreter instance.

The number of interpreter instances equals to the total number of tasks, which use the same interpreter body parameterized with the task number at the respective interpreter instance initialization.

Such data-driven organization of the test suite has an important advantage: the test application calls the OS under test at only one point, where the test interpreter

invokes a service procedure or a utility pointed to by the flat chart line being interpreted. This simplifies realization of local time measurements (subsection 6.1).

A flat chart description for the *DelayCoEnd* test application in section 4 is represented as an array of *TestStep* structures. Such a form may be used directly for developing a test suite in C. OS services like *CallDelay()* may look as Fig 6 shows.

```
void CallDelay ( )                                  {
        /* Global test variables */
   Delay  ( (CurrentStep->Arg_1).IntArg );      }
```

Fig. 6. OS services CallDelay()

To avoid heavy line interpreting during test runs in real-time, a Forth-like method based on threaded code [6] may be used: the test representation shall be pre-processed in order to convert the initial form into a regular array which elements store the task number, a pointer to the procedure to be called, and the procedure parameters.

5. Developing Scenario Tests with Flat Charts

In accordance with the focus rule (subsection 2.1) each functional test such as *DelayCoEnd* checks a particular OS feature under specific conditions. In this respect, functional tests are not like regular applications. A complete test suite should include also a set of *scenario tests*, which are much closer to regular applications. Each scenario test realizes a sequence of actions, which is based on some underlying idea and uses the OS in a way close enough to real functioning. The flat chart technique is suitable to describe them. The following array (Fig. 7) describes a scenario test for message passing between four threads – three tasks and an ISR.

```
// -----   Flat chart for message passing test application   ------
TestStep MsgTravel [  ] =   {
   1,&CallGetMsg, &mes1_ptr,  0,              // No msg, TASK_1 is waiting
      2,&ResumeIsr,  0,  0,                   //  Interrupt is  simulated
   -1,&CallPutMsg, TASK_3, TEST_MSG,          // Send msg to TASK_3
      2,&CallGetMsg,  &mes2_ptr,  0,          //  No msg, TASK_2 is waiting
            3,&CallGetMsg,  &mes3_ptr,  0,    //  TASK_3 received Msg
            3,&CallPutMsg,  TASK_1, &mes3_ptr,     //  Activate TASK_1
   1,&CallPutMsg,  TASK_2, &mes1_ptr,         // TASK_2 becomes ready
   1,&CallTaskEnd,   0,  0 ,                  //  Activate TASK_2
      2,&Check_Equal,  &mes2_ptr, TEST_MSG,      //  Is msg the same
      2,&End_of_Test,   0,   0,                  // as TEST_MSG?
   0,&End_of_scheme,  , 0 }
```

Fig. 7. A scenario test for message passing between four threads

A negative number in the *TaskId* field corresponds to an operator to be executed by an ISR. Its absolute value specifies the nesting level of the interrupt, which this particular line of the flat chart is interpreted at, rather than a particular ISR.

The scheme *MsgTravel* was designed to check the message exchange mechanism, which provides message pointer passing from an ISR to a task or from one task to another. Variables *mes1_ptr*, *mes2_ptr*, and *mes3_ptr* are message pointers. The value of the constant *TEST_MSG* is a pointer to some initialized message instance.

The scenario of message passing between tasks consists of the following events:

- *Task_1* tries to receive a message and becomes suspended because there's no message for it yet (*Step_01*);
- the ISR passes the message *TEST_MSG* to *Task_3* which is not ready yet to receive it (*Step_02*, *Step_03*);
- *Task_2* tries to receive a message which is absent and therefore becomes suspended (*Step_04*);
- *Task_3* receives the message *TEST_MSG* sent previously by the ISR and resends it to suspended *Task_1* waiting for it (*Step_05*, *Step_06*);
- *Task_1* resends the message to *Task_2* and frees the processor through invoking the service procedure *TaskEnd()* (*Step_07*, *Step_08*);
- upon termination of *Task_1* the message received by *Task_2* is compared to *TEST_MSG* – the two message pointers should coincide (*Step_09*).

The utility procedure *ResumeIsr()* initializes ISR invocation. The simplest way to do this is to throw a software interrupt. Each ISR scans the flat chart line after line, similar to a task. Therefore, an instance of the same common flow chart interpreter is generated for this ISR. Its configuring is performed by just a few operators executed by ISR before entering the common interpreter body. Hence, the same unified flat chart interpreter is used by tasks and by ISRs.

5.1. Loops in Flat Charts

Auxiliary items, such as the terminator *End_of_scheme* mentioned in previous sections are used to build flat charts. Two other kinds of auxiliary items are described below: a flat chart *loop delimiter* (this subsection) and an *error checking operator* (subsection 5.2).

The flat chart loop mechanism allows to prevent construction of a long scheme with repeated fragments. The test *MessQueue* checks the message queue mechanism: a queue of 10 messages is formed for *Task_2*, which then consumes these messages from the queue one after another.

```
//  -----   Flat chart with single loop   ------
TestStep MessQueue [ ] = {
// ----  The message queue with 10 messages is formed for Task_2
0, &LoopStart, &cycle_var,  10,
   1,&CallPutMessage, Task_2, &mes1_ptr,  // Repeat msg send
0, &LoopEnd, &cycle_var,  0,
   1, &CallTaskEnd,  0,  0,                          // Task_1 terminates
 // ---- The message queue of 10 messages is consumed by Task_2
0, &LoopStart, &cycle_var,  10,
          2,&CallGetMessage,  &mes2_ptr,   0,  // Repeat msg receive
0, &LoopEnd, &cycle_var,  0,
0,&End_of_scheme,  , 0        }
```

Fig. 8. Flat chart with single loop

The variable *cycle_var* is used as the loop control variable. Nested loops may by designed, each with its own control variable; e.g., *cycle1_var* for an outward loop and *cycle2_var* for an inner loop.

For each loop a boundary condition shall be satisfied: the task state at the *LoopEnd* delimiter shall be the same as its state when the corresponding *LoopStart* was encountered.

5.2. Testing the Error Handling Service

Test applications *DelayCoEnd* and *MessTravel* demonstrate the suitability of the flat chart technique for testing most of the OS basic services. Each one checks the order of processor switching among threads of actions. *MessTravel* checks correctness of data passing between tasks and from an ISR to a task. Allocation of memory and of special data structures may be checked in a similar way.

Flat chart forms may be further extended to cover testing of the error handling service as well. The following flat chart sample illustrates this possibility (Fig. 9).

```
//  -----   Flat chart for error service testing   ------
   TestStep MemReqErr [ ] =  {
// .......Steps from Step_01 to Step_i exhaust all memory resource
       1,&GetMemory,  30, &mem_ptr,            // Step_i+1
       0, &CheckErrData, NO_MEMORY,  0,
 // ..................... Remaining elements of the MemReqErr array
       0,&End_of_scheme,  , 0 }
```

Fig. 9. Flat chart for error service testing

Task_1 requires 30 memory blocks, which causes an error because the memory resource becomes exhausted. The proposed technique of testing the error handling

service is based on the same interrupt simulation technique as in the *ResetIsr()* utility. An error invokes a special thread of actions, which the flat scheme interpreter body enters. The interpreter finds the respective auxiliary line in the flat chart and performs the *CheckErrData()* utility assuming that the OS reports the *NO_MEMORY* error code into the error handling block.

Thus, auxiliary items extend the flat chart technique and allow to build tests for checking the OS error handling service.

6. Automated Test-Run Sessions

A test suite for an embedded OS shall include automated means for building a test application, for loading it into the target device, for test run, and for producing test-run reports with analysis of the test-run session. Automation tools are intended to organize a specified test session. The test session specification describes an action list for building, loading, running test applications, and analyzing the results.

Scalability is one of the most important requirements for an embedded OS testing application. The user may configure its options to achieve the needed level of efficiency in terms of speed, memory usage, and the needed inventory of services to be used. The number of such OS clones grows exponentially with the number of options. A dozen of binary options correspond to a thousand and more of different OS clones to be tested. A wide set of tests should be built, loaded, run, and analyzed for each such clone, their total number may be a million and more. This results in the need for automation of test sessions with tools to specify them.

Beyond OS scalability there are at least two more reasons for test session automation: OS *projected enhancements* and OS *porting* to other MCUs.

When an OS is ported to a different MCU, the test suite should be ported as well and such porting should take much less effort than initial development. The flat chart technique and automated test sessions allow to save porting efforts .

6.1. Local Time Measurement

There are three basic points in the flat chart interpreter body executed by every thread in the test application. They are:

- the main interpreter loop start point;
- the main interpreter loop end point;
- points of invocation of an OS service or utility.

These points split the body of the interpreter *FlatChartInterpreter()* into the following three sections:

- Section 1 – thread configuring to prepare the thread to enter the main interpreter loop: initialize local variables, determine the *TaskId* or the ISR nested level, set *CurrentStep* to point at the top of the *TestStep* array, and specify the start point of the main interpreter loop;

- Section 2 – organize interpretation of the flat chart through a search of the appropriate item in the *TestStep* array: set *CurrentStep* at the appropriate value, check correctness of the operation sequence, and perform actions prior to invocation of a respective service or utility;
- Section 3 – perform a call of the *UtilServ* utility: (*A*) for local time measurement read the timer register, (*B*) call the *UtilServ* utility, (*C*) for local time measurement read the timer register again, (*D*) perform log operations.

The flat chart technique simplifies realization of local time measurements. Calling an OS service or a utility in the point *B* is performed through indirect addressing of the called procedure. All time measurement actions are around the point *B* (in points *A* and *C*). Storing the result of time measurement is performed in point *D*.

In case of a context switch resulting from performing operator *D*, operator *A* may be performed in a thread other than that, which operator *C* was performed in.

6.2. Global Time Measurement

The OS time service directives are not appropriate for local time measurements. Their precision is not adequate and a direct access to the hardware time register is needed. In contrast, global time measurements are less precise; therefore, the OS time service may be used for them. The structure of a flat chart for global measurements may look as Fig. 10 shows.

```
TestStep HighPriorTaskSwitching [ ] =    {
  1,&CallSysTime, &start_time,  0,          // Store the start time
  0, &LoopStart, &cycle_var,  1000,         // Initialize the loop
//  ------ The set of operations for measurements  ------
  1,&CallGetMessage, &mes1_ptr,  0,         //  Suspend Task_1
     2,&CallPutMessage, &mes1_ptr,  0,      // Send msg to Task_1
  0, &LoopEnd, &cycle_var,  0,              // Terminate loop operations
  1,&CallSysTime, &finish_time,  0,         // Store the end time
  0,&LogGlobalTime,  0,  0,                 // Store the result
  0,&End_of_scheme,  , 0                }   // End of scheme
```

Fig. 10. Structure of a flat chart for global measurements

The number N of cycles required for measurement depends on the relation between the precision ΔT_m of the *SysTime()* mechanism and the duration T_c of one application cycle. Another factor is the duration T_p of the *LoopStart()* and *LoopEnd()* operations (assuming they are equal). The larger the value of $N \times (T_c/(\Delta T_m + T_p))$, the more precise measurement results will be obtained.

Global measurements provide an answer the question: "Does the time of context switching depend on the task priority?" To answer this question, compare the result

of the *HighPriorTaskSwitching* test with the result of the test shown in Fig. 11.

```
TestStep LowPriorTaskSwitching [ ] = {
//  ..... Suspend the set of 100 tasks with high priorities
      101,&CallSysTime, &start_time,  0,        // Store the start time
      0, &LoopStart, &cycle_var,  1000,         // Initialize the loop
// ------ The set of operations for measurements ------
      101,&CallGetMessage, &mes1_ptr,  0,       // Suspend Task_101
        102,&CallPutMessage, &mes1_ptr,  0,     // Msg to Task_101
      0, &LoopEnd, &cycle_var,  0,              // Terminate the loop
      101,&CallSysTime, &finish_time,  0,       // Store the end time
      0,&LogGlobalTime,  0  0,                  // Store the result
      0,&End_of_scheme,  , 0          }         // End of scheme
```

Fig. 11. The test, in which a set of high priority tasks is suspended prior to entering the flat chart loop

The only difference between flat chart loops in the tests *HighPriorTaskSwitching* and *LowPriorTaskSwitching* is in the task priority. In the second test, a set of high priority tasks was suspended prior to entering the flat chart loop. If the OS context switching is performed at the same time for tasks with different priorities, then the measurement results will be the same for both tests.

The considered two tests are a particular case of a round-robin processor switching among tasks. Such a scheme may include an arbitrary number of tasks with different priorities. Changing the number of operating tasks allows to establish the dependency between OS performance and its load while changing the task priorities may impact the speed of task scheduling.

6.3. Latency Testing

For a multi-threaded application executed on a single processor, the tasks and ISRs operations are executed in a quasi-parallel mode. Flat charts are convenient for specifying such quasi-asynchronous processes. From the OS point of view, all threads are asynchronous, but the test logical structure guarantees strong synchronization of all operations in different threads.

However, a true asynchronous mode of operation is needed for measuring the application latency w.r.t. external interrupts. The simplest statistical way of latency measurement assumes simultaneous execution of two logically isolated components:

- a benchmark application with a set of interacting tasks;
- a special measurement ISR to calculate time difference between the moment of the measurement interrupt and the moment when its processing started.

The benchmark application determines conditions for measuring the latency value.

It is built in form of a flat chart within a loop with a large number of iterations, which ensures the repeatability of the conditions of latency measurement.

With this approach, a single result L_m of measuring the latency value will be less than or equal to the maximal possible latency value L_r: $L_r \leq L_m$. The difference $d=L_r-L_m$ represents the inaccuracy a single latency measurement. Let the acceptable inaccuracy Δt of the final result of measurement and the time interval T of time measurement interrupts be greater than the duration of one iteration of the benchmark application. Then the probability P that the required accuracy of measurement is achieved ($d<\Delta t$) is greater than or equal to $\Delta t/T$: $P \geq \Delta t/T$. To achieve higher accuracy of the latency measurements, single measurements are performed n times and the maximum of the values L_m is considered as the final result. The required accuracy of the final result is achieved with the probability P not less than $1-(1-\Delta t/T)^n$: $P \geq 1-(1-\Delta t/T)^n$.

6.4. Measuring Code Coverage

A straightforward technique to measure code coverage of the OS under test by a given test suite is based on direct tracing of the OS code control flow supported with designated software-hardware means. It's hardware component should have a mechanism of trace interrupts with a designated vector (TRAP-interrupts). This software component is composed by a handler of step-wise interrupts which performs the role of the tracing program. Execution of each OS instruction is preceded by an interrupt on the TRAP-vector, which results in the next activation of the tracing program.

This technique of direct tracing matches the rule for non-interference (subsection 2.1). However, it may be inapplicable for embedded systems because an embedded application under test may work much slower when running in parallel with the tracing program. Some operators covered in a real run may be unreachable in the mode of coverage measuring.

A more appropriate technique of measuring code coverage is based on using codes of prohibited TRAP instructions. This mechanism is realized with another designated vector of TRAP-interrupts. In this case, the respective interrupt handler plays the role of the tracing program and the coverage measurement process consists of the following steps:

- the contents of the memory area with the OS body (its code) is saved in a special array and then is filled with the codes of TRAP instructions;
- execution of the test application is started and a software TRAP-interrupt occurs when any OS service is invoked;
- the tracing program is invoked as the interrupt handler, it restores the original OS instruction from the special array and passes control to it;
- the restored original OS instruction is executed;

- if the next instruction to be executed is from the OS body, then it may be either restored through previous executions or still replaced with a TRAP instruction and then another TRAP-interrupt occurs which restores the original OS instruction so that more and more OS instructions are restored.

Upon termination of the test application all OS instructions needed for this application will be restored and their number equals to the number of invocations of the tracing program.

This technique of code coverage measurement with TRAP instructions decreases the time of the test application execution if compared to technique with direct tracing. Each OS instruction corresponds to at most one invocation of the tracing program and therefore the overall execution pace becomes close to that of a regular execution without tracing. A complete match of these two paces is achieved when only one OS instruction, which we'd like to find whether it's covered or not is replaced:

- this one OS instruction is saved and replaced with a TRAP instruction;
- the test application runs to termination and if the instruction is not restored then it was not covered.

This technique with single instruction replacing requires much more processor time because complete measurement of code coverage assumes iterative runs of the test application as many times as there are instructions in the OS body.

6.5. Enhancements of the Flat Chart Technique

As noted in subsection 6.3, the flat chart technique allows to describe a quasi-asynchronous order of test application runs only. To represent true asynchronous threads of actions (as required for latency measurements), methods beyond the flat chart scheme should be used.

The quasi-asynchronous order fits well for testing OS kernel services. However, for testing services related to peripheral devices an extension of the flat chart technique is needed which allows to specify real asynchronous action flows. This may be done through introducing new forms, which specify alternatives in the action flow similar to loop forms in subsection 5.1.

The flat chart technique may be further extended to distributed OS testing. In this case, a test application is a program with true parallelism and if quasi-asynchronous execution turns out to be suitable for particular testing, then the only extension needed is refinement of action flows naming. Otherwise, a separate flat chart should be developed for each physical processor with additional means for cross-referencing among elements of these flow charts.

Flat charts form representations considered above are suitable for usage in C-programs. Similar syntax forms, which require no any special pre-processing, may be developed for other programming languages. However, when moving from one language to another flat charts should be completely reworked which is effort

consuming as the total size of flat charts in a test suite may reach hundreds of thousands lines. Thus, it is reasonable to develop a language independent unified syntax for flat chart forms. Then porting a test suite to another platform requires only to develop a pre-processor of several hundred lines of code. Development of a universal syntax forms for test representation opens the opportunity to build standardized test suites for embedded OS testing. A universal language for OS test applications could be a step forward in development of an automatic test generator [7], [8].

7. Results of Experiments

Experimental data provided below come from authors' experience in developing and testing a particular software product – a compact embedded OS for real-time applications with specific features requested by the customer. The overall approach to developing this OS follows the classical one [9] initially designed for 16-bit single board controllers manufactured by DEC since early 1980-ies. To emphasize the compactness and specifics of such OSs they are usually named "*kernals*" or "*executives*". The usual size of such an OS developed within this approach is about several thousand lines of code in C plus several hundred lines in assembler.

The MCUexec (MicroController Unit EXECutive) product, which development the authors participated in, supported execution of software applications on microcontrollers HC-11 and HC-12 originally manufactured by Motorola, Inc. and since 2015 by NXP Semiconductors. To test the MCUexec functional features, 9 groups of flat charts were developed with the described technique.

For integration testing of MCUexec additional 234 flat charts split in 17 groups were developed, the total number of the developed flat charts being 378. Running all these test suites resulted in 8 detected defects in different versions of MCUexec, each of about 5 KLOCs in assembler. The overall effort for developing these flat charts, running the test suites, and analyzing test run results was 6 staff-months.

Table 1. Nine groups of flat charts for testing the MCUexec functional features

Test group identifier	Brief description	Number of flat charts
Basic	Task delay, system configuration and reconfiguration	10
TaskId	Getting the task Id	3
Task	Task suspending/resuming	12
EventU	Updating and checking of events	21
EventW	Waiting for an event to be set or cleared	30
Slice	Time-slicing features	6
Buf	Buffer manipulating	23
MesS	Message sending and receiving	20
MesR	Reply features	19
	TOTAL:	**144**

8. Conclusion

The flat chart technique gives an efficient way to develop test suites for embedded OS execution-based testing. Flat chart forms allow to build well-structured and understandable descriptions of test applications with specifications of tasks and ISRs for parallel execution. The flat chat technique is suitable for checking the correctness of implementation of basic OS mechanisms – data and signal exchange among action threads, run-time allocations of memory, special structures, and processor's time. Flat charts are efficient not only for developing functional tests but for local and global time measurements, for measuring the OS latency and code coverage. Standardized test suites for embedded OS testing may be built with the described flat chart technique.

References

[1]. Li Q., Yao C. Real-time concepts for embedded systems. CRC Press (2003).
[2]. Thane H., Hansson H. Testing distributed real-time systems. Microprocessors and Microsystems 24(9), 463–478 (2001).
[3]. Desikan S. Software testing: principles and practice. Pearson Education India (2006).
[4]. Myers G.J., Sandler C., Badgett T. The art of software testing. 3rd Edition. John Wiley & Sons, New York (2011).
[5]. Hailpern B., Santhanam P. Software debugging, testing, and verification. IBM Systems Journal 41(1), 4–12 (2002).
[6]. Brodie L. Thinking Forth. Punchy Pub (2004).
[7]. Biswal B. N. Pragyan N., Durga P. M. A novel approach for scenario-based test case generation. In: International Conference on Information Technology 2008 (ICIT'08). IEEE, (2008).
[8]. Lefticaru R., Florentin I. Automatic state-based test generation using genetic algorithms. In: International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)
[9]. Comer D. Operating System Design: The Xinu Approach, 2nd Edition. – Boca Raton: CRC Press, Taylor & Francis Group, 668 p. (2015).

Техника плоских схем для тестирования встроенных операционных систем

В.В. Никифоров <nik@ iias.spb.su>
С.Н. Баранов <snbaranov@ iias.spb.su>
Санкт-Петербургский институт информатики и автоматизации
Российской академии наук,
199178, Россия, Санкт-Петербург, 14 линия, 39

Аннотация. Современные автоматические устройства все чаще оснащаются микроконтроллерами. Логика работы автоматического оборудования поддерживается рядом различных встроенных программных приложений, которые выполняются под управлением встроенной операционной системы реального времени (ОС). Надежность

ОС чрезвычайно важна для правильной работы всей автоматической системы. Поэтому встроенную ОС следует тщательно тестировать с помощью соответствующего набора автоматических тестов. Такой набор тестов для тестирования встроенной ОС обычно организуется как набор многозадачных тестовых приложений, которые должны выполняться под управлением данных. В статье представлены специальный язык для определения соответствующей логики задачи тестирования и концепция плоских съем для эффективного выполнения тестирования встроенной ОС. Чтобы избежать интенсивной интерпретации текстовых строк во время тестового прогона, предварительно образуется специальное представление теста, в котором исходная строковая форма преобразуется в форму регулярного массива и, таким образом, повышается эффективность тестирования.

Ключевые слова: встроенные приложения; операционные системы; тестирование программного обеспечения; системы реального времени

DOI: 10.15514/ISPRAS-2017-29(5)-5

Для цитирования: Никифоров В.В., Баранов С.Н. Метод плоских схем. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 75-92 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(5)-5

Список литературы

[1]. Li Q., Yao C. Real-time concepts for embedded systems. CRC Press (2003).
[2]. Thane H., Hansson H. Testing distributed real-time systems. Microprocessors and Microsystems 24(9), 463–478 (2001).
[3]. Desikan S. Software testing: principles and practice. Pearson Education India (2006).
[4]. Myers G.J., Sandler C., Badgett T. The art of software testing. 3rd Edition. John Wiley & Sons, New York (2011).
[5]. Hailpern B., Santhanam P. Software debugging, testing, and verification. IBM Systems Journal 41(1), 4–12 (2002).
[6]. Brodie L. Thinking Forth. Punchy Pub (2004).
[7]. Biswal B. N. Pragyan N., Durga P. M. A novel approach for scenario-based test case generation. In: International Conference on Information Technology 2008 (ICIT'08). IEEE, (2008).
[8]. Lefticaru R., Florentin I. Automatic state-based test generation using genetic algorithms. In: International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)
[9]. Comer D. Operating System Design: The Xinu Approach, 2nd Edition. – Boca Raton: CRC Press, Taylor & Francis Group, 668 p. (2015).

Designing variability models for software, operating systems and their families[1]

[1,2]*E.M. Lavrischeva <lavr@ispras.ru>*
[1]*V.S. Mutilin <mutilin@ispras.ru>*
[1]*A.G. Ryzhov <ryzhov@ispras.ru>*
[1]*Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, A. Solzhenitsyn st., Moscow, 109004, Russia*
[2]*Moscow Institute of Physics and Technology, 9, Institutskiy per., Dolgoprudny, Moscow region, 141701, Russia*

Abstract. The complexity of existing Legacy systems and the difficulty of amending it led to the development of the new concept of variability of systems specified by a model of the characteristics of FM (Feature Model). In the paper, we discuss the approaches to formal definition of FM and creating on its basis variants of program systems (PS), operating systems (OS) and families of program systems (FPS) for PS and OS. We give methods of manufacturing of PS in the Product Family/Product Lines, the conveyor of K.Czarnecki for assembling of artifacts in the space of problems and solutions, logical-mathematical modeling of PS from the functional and interface objects by Object-Components Method (OCM), extraction of the functional elements from OS kernel to FM for the generation of new variants of the OS. We discuss approaches for formalization of variability of legacy and new PS and their FPS. The new concept of management of variability systems with help OCM is defined. The approach to verify models of the FM, PS, FPS and OS and to configuration of functional and interface objects for obtaining the variants of the resulting product are proposed. We elaborate the characteristics for the testing process of variants of the PS, OS and FPS.

Keywords: variability model; software systems; family of systems; configuration; variant; functional, interface element; requirement; management

DOI:10.15514/ISPRAS-2017-29(5)-6

For citation: Lavrischeva E.M., Mutilin V.S., Ryzhov A.G. Designing variability models for software, operating systems and their families. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017, pp. 93-110. DOI: 10.15514/ISPRAS-2017-29(5)-6

[1]The work is supported by RFFI grant N16-01-00352

1. Introduction

In the recent years, new modeling methods of the program systems (PS) and families (FPS) appeared in software engineering. The methods are aiming to ensure the variability of software systems, both legacy and newly produced ones. One of the first Feature Models (FM) called Product Line/Product Family was developed at the Software Engineering Institute (sei.cmu.edu) for manufacturing software products and their families basing on the assets by customers requests. Product line is group of products or services sharing a common managed set of features that satisfy specific needs of a selected market or mission. K.Czarnecki proposed a concept of generation of PS and FPS based on FM from reuses and artifacts. Object-Component Method (OCM) enables modeling of functional elements with support for variability [1-15].

In the paper, we introduce new models with functional and interface elements and FM from these elements for generation of variants of PS and their families.

2. The Basic Foundation of the Variability of Systems and Families

The FM for software products was first proposed by K.Pohl [1, 2] as a basis for creating variants of software and OS [3-9]:

1) requirements for software are specified by means of the languages – FODA, RSEB, Forfamel, RequiLine, CBFM , Use Case precedents UML etc.;
2) tools – ConIPF, CONSUL/Pure::Variants, GEARS are used for integrating the variability of artifacts with special languages, like Koala, xADL, OVM, VSL etc.;
3) OS mechanisms and functions (e.g. Unix, Linux, etc.), which can be generated in LEADS, OCM[16-27] with the languages (VSL, ConIPF, CBFM, Koalish, Pure::Variant, COVAMOF and others) establishing relationships between characteristics of FM and the variants of PS.

This paper sets out the basic principles of simulation of variability of PS and FPS in existing approaches of K.Pohl, K.Czarnecki, etc. and proposes the Object Component Method of presenting FM on four-level design using a functional and interface objects. We also define configuration management process in accordance with the Deming cycle [22] to obtain variants of the PS and FPS.

2.1. Variability of Products and Systems

K.Pohl introduced the concept of variability in FM out of existing artifacts, reuses, etc. Variability is a property of a product (system) for expanding, changing, adaptation, or configuration for use in a particular context and to ensure its subsequent evolution [1, 2]. The FM model includes common functional and non-functional characteristics of items that can be used by members of the family of FPS for creating different variants of the PS configured at variation points [1-29].The

variation point is a place in the Legacy-system, which are used for production variant of the big systems.

FM defines the process of creating a product from existing software elements, which are called Ready to Use Component (RUC) [11], which includes – reuses, assets, applications, etc. The FM in Software Product Line Engineering (SPLE) is based on two processes: engineering of domain and application engineering.

The main aspects of variability of products and systems are:

- model characteristics of the FM with variation points for functional elements;
- variability of the system architecture with variation points;
- managing variability of RUC.

2.2. Variability in the Space of Problems and Solutions

K.Czarnecki [3, 6] provides a modeling of the architecture of the PS and FPS in the problem space and problem solution similar to SPLE approach. The basis of the approach is the characteristics of RUC that appear in FM implementing requirements to PS or FPS. Between characteristics (n) and requirements (m) there may be $n \times m$ relations. Each PS is defined by selecting a group of characteristics.

FM consists of the functions that are available to the user of the system and can be in the spaces of problems and solutions, and describes the domain model by means of DSL (Domain Specific Language) with a means for increasing the level of abstraction of FPS.

2.3. Variability of the Functional and Interface Elements in OCM

The Object-component method OCM proposes a four-level design of object model (OM) of PS and FPS [21]. After design of the OM, we obtain the graph G, which has the form:

$G = (G_{t1}, G_{t2}, G_{t3}, G_{t4})$, where

G_{t1} – objects at the synthesis level (t = 1);

G_{t2} – FM at the characteristics level (t = 2);

G_{t3} – functions at the structural level (t = 3);

G_{t4} – interfaces relationships at the behavioral level (t = 4).

Fig. 1 shows the elements of processing on the levels of design and structure of OCM objects, their structure, characteristic functions $F_o = (f_{o1}, \ldots, f_{on})$ and interface elements of them $I_o = (i_{o1}, \ldots, i_{om})$ [4, 13, 21].

These features of functional and interface object of OM are located in the PS and FPS.

A functional object f_o specifies a formal description of application functions PS, which ensure the solution of problems of a particular domain. This object is given by a triple: the name, data types and their values.

Interface object i_o specifies a formal description of the operations and data of functional objects. The i_o object is a mediator of interacting functional objects and $I_o(f_o)$ equals to $In(f_o)$ *or* $Out(f_o)$ *or* $Inout(f_o)$, where

$In(f_o)$ is a set of input interfaces for transferring data from the f_o to the other objects;

$Out(f_o)$ is a set of output interfaces for transferring output data back to the object f_o;

$Inout(f_o)$ is an intermediate interface that converts data from/to f_o.

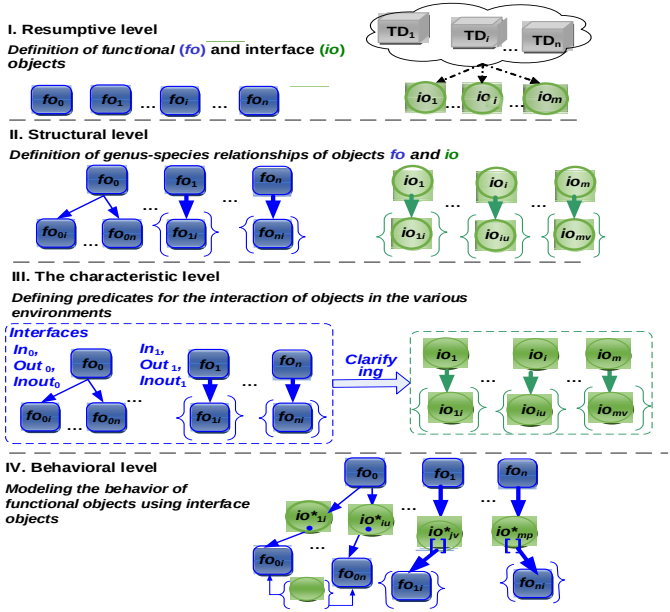


Fig. 1. OM of graph G with functional and interface objects

Axiom. For each functional object, the FPS has at least one characteristic (internal or external) that defines semantics and a unique identification it in the set of F_o and interfaces I_o.

Features allow to establish the truth of the matching types with $Con_i = (P_{i1}, \ldots, P_{ik})$ where P_{ik} is a condition predicate on F_{oi}.

Four-level mathematical design FPS of functional and interface objects is defined by the graph: $G = (F_o, I_o, R)$, where F_o– the set of functional objects, I_o – the set of interface objects, R – the set of relations between these objects [13].

Graph G includes a front-end objects I_o (Fig. 2), which call the other object and pass appropriate data with the required type and size.

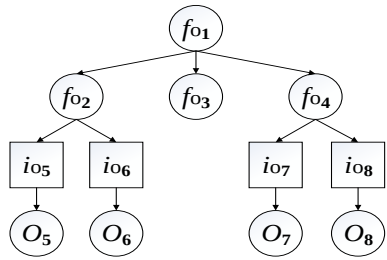


Fig. 2. The graph G on the set of functional and interface objects

Nodes of G represent functional elements – f_{o1}, f_{o2}, f_{o3}, f_{o4}, f_{o5}, f_{o6}, f_{o7}, f_{o8} and interface elements – i_{o5}, i_{o6}, i_{o7}, i_{o8}, and edges correspond to relations R between all types of objects.

Elements $f_{o1} – f_{o8}$ are described in any programming language, and front-end objects $i_{o5} – i_{o8}$ are described in IDL. The parameters of the external characteristics of the interface objects are passed between objects through interfaces, and are marked as *In* (input), *Out* (output) and *Inout* (input and output).

The relationship between the functional objects f_{ok}, f_{ol} is provided by interface objects, i.e. f_{ok} is $In(f_{ok})$ or $Out(f_{ok})$; f_{ol} is $In(f_{ol})$ or $Out(f_{ol})$:

$$f_{ok} \cdot f_{ol} = (Out(f_{ol}), In(f_{ok}));$$

Theorem. The functional interaction between two functional objects is correct, if the first object fully matches functions and data that are required by another object: $In(f_{ok}) \subseteq Out(f_{ol})$.

With graph G it is possible to construct individual programs P_0 - P_3 using mathematical operation $\cup$ and corresponding *link* operation:

1) $P_0 = (P_1 \cup P_2 \cup P_3)$;
2) $P_1 = f_{O2} \cup f_{O5}$, *link* $P_1 = In\ i_{O5}(f_{O2} \cup f_{O5})$;
3) $P_2 = f_{O2} \cup f_{O6}$, *link* $P_2 = In\ i_{O6}(f_{O2} \cup f_{O8})$;
4) $P_3 = f_{O4} \cup f_{O7}$, *link* $P_3 = In\ i_{O7}(f_{O4} \cup f_{O7})$;

Below we consider the design of models of systems and their variability models.

2.4. The definition of models of the PS, OS and their variability

This section discusses models of PS, FPS, OS and their variability.

2.4.1 The models of PS, FPS and their variability

Model of PS – $M_{ps} = (C_L, M_f, M_s, M_i, M_d)$, where

C_L – are languages $L = L_1, L_2, ..., L_N$,

$M_f = (f_{o1}, f_{o2}, ..., f_{or})$ – functional objects;

$M_s = (Ms_{in}, Ms_{out}, Ms_{inout})$ is the set of services – input Ms_{in}, output Ms_{out} and server Ms_{inout};

M_i – is a set of interfaces in IDL;

M_d – data of the *PS* [9, 13-22].

Model of variability PS – $M_{var} = (SV, AV)$ [14, 15], where

SV – submodel of variable architecture *PS*;

AV – submodel of variability of artifacts FPS or RUC.

M_{var} enables variability of products and reduces development costs with the help of RUC.

The submodel *AV* determines the structure of the *PS* from RUC, which are stored in the repository. This submodel displays the characteristics of FPS, as well as aspects of the relationship (through the interfaces) between different levels of the OM. Variation points are handled by the configurator and replaced by some other RUC (correct ones or new).

The submodel $SV = ((G_t, tr_t), Con, Dep)$, where

$G_t = (F_t, LF_t)$ – is a graph of artifacts on level *t*;

tr_t – connection between artifacts on level *t*;

Con, Dep – the predicates of the sets of artifacts that define the constraints and dependencies among the functional elements and their indicators of quality.

The concept of a family of programs introduced Dijkstra (1970), which is based on "family" which can be derived from different versions of programs, and can be adjusted and replaced according to requirements [11-24]. Family of program systems – FPS is a set of systems with a common set of concepts, specific data, and functional and interface characteristics that are inherent to every member of the family.

Model of FPS family has the form:

$M_{FPS} = \{M_{OM}, M_{FM}, M_{var}, MC\}$, where

M_{OM} – OM;

M_{FM} – FM;

M_{var} – variability model of FPS;

MC – model of configuration assemblies;

Model of variability of the FPS has the form [18]:

$SV_{FPS} = ((CF; (DR, TC); (CM, FR, TS, TA); (ER, TF))$; *Con, Dep*), where

CF – characteristics of the system,

DR – detailed characteristics related to requirements of PS;

TC – relations between the requirements of PS and consumer properties;

CM – the set of formally described software elements of the set of functions *FR*;

TS – the set of formally described tests;

TA – interfaces between elements of FPS;

ER and *TF* – database for processing elements of *CM*;

Con – are domain constraints;

Dep – are dependencies between artifacts of FPS.

To assess the variability of the FPS an orthogonal variability model (OVM) is created [15, 16]. It coordinates the composition and interrelation of the family elements and artifacts of the assembly processes of the PS and FPS. The evaluation model is included in the integrated model of variability of the OVM. It is used to assess the level of variability, taking into account the requirements for artifacts.

The model OVM has the form:

OVM = (EVM, VP), where

EVM = (VL, VR),

VL – model for estimating the level of variability, taking into account the requirements to the components of the architecture, artifacts and data;

VR – model for estimating the level of variability in the FPS, taking into account the requirements, the architecture, artifacts, and data.

VP – the sets of variation points in the FPS structure that specify individual characteristics of the PS, including the constraints *Con* and *Dep* dependencies;

The OVM model defines two types of assessments of the FPS variability – level and relevance to the consumer needs. The assessment of the level of variability and the degree to which it meets the needs is carried out using the value tree and the parameters *VL* and *VR*, which take into account the costs and the frequency of producing new variants for the customer.

2.4.2. Model of operating system kernel and its variability

Model of OS kernel is a collection of individual program fragments implementing the functions of the OS, e.g. Linux [3, 23, 25, 26].

Model of OS kernel is a set of artifacts and interfaces between them:

$M_{os} = (S_k, M_f, M_o, M_i)$, where

S_k – a set of fragments of OS code;

M_f – a set of features;

M_o – a set of dependencies between features,

M_i – a set of interface features (subset of M_f).

The variability model of the OS is identical to the PS model [25].
The OS defines a set of functions and their features. To generate some variant of OS we define the required set of functions and specify the set of interface features.

3. Managing variability of systems

Variability of systems depends on requirements, FM, architecture, documentation, tests, etc. In general, the variability can be implemented in both PS and FPS. In the case of PS variability includes documentation, functions and elements of any type. In FPS, the variability includes the sets of individual products.
The variability of the FPS is managed with:

- variation points;
- versions of the artifacts;
- predicate constraints for variation points.

The variability is managed by method of E.Deming, determined by the functions F1 - F4 (Fig.3) of the development of FPS [22-27]:

Fig. 3. Functions in the Deming cycle

F1 – operations and actions for preparation of artifacts (Act) [23];

F2 – planning of system construction from the artifacts (Plan at the levels of domain engineering and application engineering);

F3 – testing and verification of changes (Check);

F4 – update system FPS (Do).

Managing variability of FPS in accordance with the requirements *R* is performed by:

1) justification of the solution *F1* (requirement *R1*);
2) agreement on the implementation approach (requirement *R2*);
3) validating the correctness implementation (requirement *R3*);
4) tracking relationships between system characteristics at all development stages (requirement *R4*).

4. Verification of the model variability

The object of verification is a model of the characteristics of the FM and requirements for the development of a new system. Properties of objects, subject to verification of FM are described by means of Linear Temporal Logic (LTL) or a Computational Tree Logic (CTL). Main approaches to formal verification of object are based on deductive verification and model checking [22-29].

Model checking is only applicable to models with a finite number of states and consists in checking that the model conforms to its formal specifications. The specifications are described using the language of temporal logic and assertions. If there is a mismatch between the model and specifications then the counterexample is produced.
The model checking involves execution of the following actions:

1) Build a model of the functional and interface objects, which must have a small number of states.
2) The specification of the requirements in terms of temporal logic.
3) Verification of the model.

5. Assembling artifacts and RUC in FPS

The assembly of artifacts includes steps F1–F4 using the RUC, and storing them in the repository according to requirements and predicates for RUC [13-15]. The configuration management of the PS in the FPS includes:

- identification of configuration items and data;
- managing the process of changes of artifacts and products;
- changing the models of variability under the new requirements;
- assessment of the variability of the PS and the FPS.

For managing artifacts and their variability in the PS and FPS, so-called model configuration environment is created, which includes:

– building process of RUC and artifacts of the system;

– schema description of artifacts and database of requirements;

– architecture, a set of basic RUC and PS in repository.

– Configurator, combining the artifacts in PS and FPS

Managing of configuration environment includes collecting data for such standard operations like reporting and audit.
Reporting configuration – collecting and reporting all necessary information about the state of the development process of the PS.
Audit configuration – guarantee that the PS contains the functionality planned in accordance with the specifications including requirements, architecture and user documentation.
In the development of PS the term "assembly" refers to the process of source code transformation from artifacts or RUC, which can be done on a computer and converted into code to run. One of the steps of Configurator is compilation of the source code into intermediate code or into the machine code. Then the linking process replaces the addresses of functions by real addresses used in the program at run time.
Configuration build is based on (Fig.4):

- the engineering model for the development of elements, components, assemblies, reuses, assets, services, product, FPS, and development management to plan and coordinate this activity [25];
- the process model under development "to ensure re-use" (for reuse) and develop "the use of RUC" (with reuse); the model for controlling variability in the process of configuring the product from RUC.

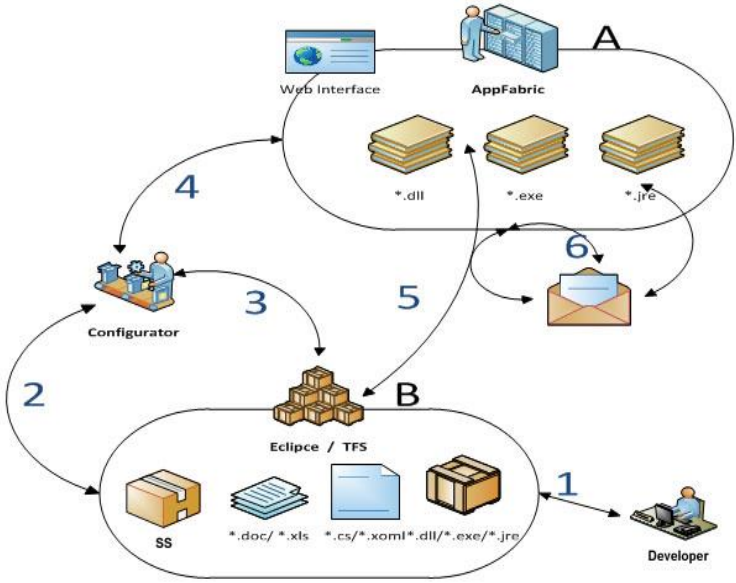


Fig. 4. General model of configurator in .Net

Organization of the development of the PS and FPS is based on the following axioms.

Axiom 1. The technology defines a cyclic sequence of software development processes and updating FPS.

Axiom 2. Each terminal characterization of OM is implemented by one and only one RUC.

6. Testing of FPS products

The structure of FPS includes RUC and test products (plans, test suite, test data, etc.) [25-28]. Test code is generated for testing individual PS and form a set of tests for the FPS. Testing method of FPS is based on requirements-based testing. It specifies the actions to manage testing on the basis of "requirements" with the help of tests to verify functional and interface objects. It determines the control degree of test objects and interfaces, and also the evaluation criterion for the quality of the FPS for the variants of the PS family of FPS. The scheme of testing of the FPS is given in Table 1.

Table 1. Basic schema of testing PS families

Types of testing test	**Testing objects**
Testing of software system architecture	All products, individual products, and functional elements
A set of tests of the family, adaptation as a set of FM to a specific product	All products are individual products of the family
Testing requirements (ScrenTED)	Products of the FPS self-test
Self-testing	Functional objects and interface
The application of metadata	Functional objects and interface
"Test print components" "testable beans" (improving productivity)	Functional objects and interface
Orientation to the service security and reliability (e.g. for Web applications)	Separate objects and interfaces
Automatic test generation for specifications in Boolean form	All products of the family
FCTA (Fault Contribution Tree Analysis)	Family products

Thus, this technique of testing of FPS [28] consists of three steps:
1) Testing artifacts, applications, PS, RUC and reducing the defects in the FPS.
2) Testing of FPS by means of tests.
3) Checking the degree of testing functional and interface objects of the FPS.
The test used offline and is common to test individual elements of the FPS.
In the last step the degree is defined by the quality of testing metric *KT*:
KT = 1, if tested operations are independent from each another;
KT = 0, if tested operations depend on the execution path and interoperability.
KT belongs to the segment [0; 1] and is calculated according to the following formula: $$KT = \frac{1}{n}\sum_{i=1}^{n} KTi$$

The final value KT of the FPS specifies the level of examination: *KT* = 1 if all objects are controlled and *KT* = 0 if not all objects are controlled and $0<KT<1$ means that the objects of FPS partially controlled.

The metrics of the control interface CI_i is calculated according to the formula:

$$CI = 1/n \sum_{i=1}^{n} CI_i$$

Where CI_i – the degree of correctness of data type conversion in the *i*-th interface object.
If $CI_i = 1$, the interface is fully controlled, $CI_i = 0$, the interface is not completely controlled;
$CI_i = (0, 1)$ – the interface is partially controlled.
The metric value of *CI* means: 1 – control of CI_i interface is complete; 0 - otherwise.
The test used offline and is common to test elements of the FPS.

testing for compliance with specified requirements to individual family members PS and the entire family FPS checks the degree of product testing. If this degree is high, then the manufactured product is delivered to the customer.

7. Conclusion

The basic fundamental concepts of modeling variability of PS and FPS are given in the original methods of K.Pohl according to FM in Product Family, K.Czarnecki in the space of problems and solutions with ready resources (reuses, assets, artifacts, RUC, etc.) and logic-algebraic approach OCM for modeling the FPS from functional and interface elements. We developed the theory of model definition of FPS from the ready resources for software for FM. The proposed variability model of FPS is based on the specified requirements of the FPS for solving optimization problems of planning of development processes and for evaluation of variability model. Methods for verification, testing, and executing of variants of PS and FPS were proposed.

References

[1]. Pohl K., Böckle G., van der Linden F. J. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, 2005. DOI: 10.1007/3-540-28901-1.

[2]. Bachmann F., Clements P. Variability in software product lines. CMU/SEI Technical Report CMU/SEI-2005-TR-012, 2005.

[3]. Lotufo R., She S., Berger T., Czarnecki K., Wąsowski A. Evolution of the Linux kernel variability model. Proc. of SPLC'10, LNCS 6287:136-150, Springer, 2010. DOI: 10.1007/978-3-642-15579-6_10.

[4]. Lavrischeva E.M., Grischenko V.N. Methods and tools for object-component programming // Cybernetics and System Analyses, 2003, №1, pp.39–55.
[5]. Kang K., Cohen S., Hess J., Novak W., Peterson S. Feature-oriented domain analysis (FODA) feasibility study. CMU/SEI Technical Report CMU/SEI-90-TR-21, 1990.
[6]. Berger T., She S., Lotufo R., Wąsowski A., Czarnecki K. A study of variability models and languages in the systems software domain. IEEE Transactions on Software Engineering, 39(12):1611-1640, 2013. DOI: 10.1109/TSE.2013.34.
[7]. Zippel R. et al. Kconfig language. https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.
[8]. Wang H., Li Y., Sun J., Zhang H., Pan J. A semantic web approach to feature modeling and verification. Proc. of Workshop on Semantic Web Enabled Software Engineering (SWESE'05), p. 44, 2005.
[9]. Lavrischeva E.M., Slabospitskaya O.O., Koval G.I., Kolesnik A.L. Theoretical Aspects of Variability Management in Product Lines Families. Vesnik KNU, series on maths and physics (1):151-158, 2011 (in Ukrainian).
[10]. Berger T. Variability mining with LEADT. DOI TSE 2014.
[11]. Lavrischeva, E.: Formal Fundamentals of Component Interoperability in Programming. In: Cybernetics and Systems Analysis, vol. 46, no. 4, pp. 639–652. Springer, Heidelberg (2010) http://link.springer.com/article/10.1007/s10559-010-9240-z
[12]. Lavrischeva E.M., Grischenko V.N. Assembling programming. K.: Basic foundation industry Software Products. - K.: Nauk.dumka, 2009.-372 p.
[13]. Ekaterina Lavrischeva, Andrey Stenyashin, AndriyKolesnyk, Object-Component Development of Application and Systems. Theory and Practice, Journal of Software Engineering and Applications, 2014, 7, Published Online August 2014 in SciReshttp://www.scirp.org/journal/jsea
[14]. Kolesnyk A., Slabospitskaya O. Tested Approach for Variability Management Enhancing in Software Product Line. – In: ICT in Education, Research and Industrial Applications: Integration and Knowledge, Proc. 8 –th Int. Conf. ICTERI 2012, CEUR –WS.org/Vol –848, ISSN 1613 –0073, urn:nbn:de:0074 –848-8. – P. 125 –133.
[15]. Kolesnyk A.L. Model and methods development families of variants of systems. – Autoref. Disser, KNU, 2013. –22p. (ukr.)
[16]. Cohen M.B., Gibbons P.B., Mugridge W.B., Colbourn C.J. Constructing test suites for interaction testing. Proc. of 25-th Intl. Conf. on Software Engineering, pp. 38-48. IEEE, 2003. DOI: 10.1109/ICSE.2003.1201186.
[17]. Lotufo R., She S., Berger T., Czarnecki K., Wąsowski A. Evolution of the Linux kernel variability model. Proc. of SPLC'10, LNCS 6287:136-150, Springer, 2010. DOI: 10.1007/978-3-642-15579-6_10.
[18]. C.Käster, A. Dreiling and K. Ostermann's ,Variability Mining with LEADT/- work is supported by ERC grant #203099
[19]. Grindal M., Offutt A.J., Andler S.F. Combination testing strategies: a survey. Software Testing, Verification, and Reliability, 15(3):167-199, 2005. DOI: 10.1002/stvr.319.
[20]. Lavrischeva E.M. Slabospitskya O.A. Approach to development object-component model family systems software products. Problems of Programming, 2013, №3, pp.14–26 (ukr.)
[21]. Lavrischeva E.M. Theory of object-components modeling of the programs systems. Preprint ISP RAS № 29, 2016, www.ispras.ru/preprints/docs/prep_29_2016.pdf.
[22]. Deming E. New economics for manufactures, governments and education, 1993.

[23]. Gruler A., Leucker M., Scheidemann K. Modeling and model checking software product lines. Proc. of IFIP Intl. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS), pp. 113-131. Springer, 2008. DOI: 10.1007/978-3-540-68863-1_8.
[24]. Ekaterina M. Lavrischeva. Assemblling Paradigms of Programming in Software Engineering, 2016, 9, pp.296-317, http://www.scrip.org/journal/jsea, http://dx.do.org/10.4236/ jsea.96021
[25]. Kuliamin V.V., Lavrischeva E.M., Mutilin V.S., Petrenko A.K. Verification and analysis of variable operating Systems.Trudy ISP RAN/Proc. ISP RAS, 23:359-370, 2012 (in Russian) Vol 28, Iss.3, pp.189-209. DOI: 10.15514/ISPRAS-2016-28(3)-12
[26]. Lavrischeva E.M., Petrenko A.K. Software Product Lines Modeling. Trudy ISP RAN/Proc. ISP RAS, 2016, vol 28. Iss. 6, pp. 180 -190. DOI: 10.15514/ISPRAS-2016-28(6)-4
[27]. C.Kästner, A. Dreiling, K. Ostermann, Variability Mining with LEADT, In Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE), pp. 157–166. 2009.
[28]. Korotun T.M. Methods and tools testing families system in resource-limited settings (ukr.), 2005, Autoref. Dis. IC NANU, 22 pages.
[29]. Lavrischeva E.M. Software Engineering. Paradigms, Technology, CASE-tools –M: Urait, 2016. – 280 p.

Проектирование моделей вариабельности для программных, операционных систем и их семейств

[1 2]Е.М. Лаврищева <lavr@ispras.ru>
[1]В.С. Мутилин <mutilin@ispras.ru>
[1]А.Г. Рыжов <ryzhov@ispras.ru>
[1]Институт системного программирования им. В.П. Иванникова РАН,
г. Москва, 109004, ул. А. Солженицына, 25
[2]Московский физико-технический институт,
Московская обл., г. Долгопрудный, 141701, ул. Институтская, 9

Аннотация. Сложность существующих систем и их сопровождения привела к созданию новой концепции вариабельности систем, определяемой с помощью модели характеристик (МХ). В статье мы рассматриваем подходы к формальному определению МХ и созданию на их основе вариантов программных систем (ПС), операционных систем (ОС) и их семейств. Мы рассмотрим методы создания ПС в линейке продуктов (ProductFamily/ProductLines), конвейере К.Чарнецки для сборки артефактов в пространстве проблем и решений, логико-математическое моделирование ПС из функциональных и интерфейсных элементов в объектно-компонентном методе (ОКМ), выделение функциональных элементов в ОС в МХ для генерации новых вариантов этой системы. Обсуждаются подходы формализации вариабельности существующих, новых ПС и их семейств. Определена новая концепция управления вариабельностью с помощью ОКМ. Предложены подходы к верификации МХ для ПС, ОС, их семейств и конфигурирования функциональных и интерфейсных объектов для получения новых вариантов системы. Изучены характеристики процесса тестирования

ПС, ОС и их семейств.

Ключевые слова: модель вариабельности; программная система; семейство систем; конфигурация; вариант; функциональный, интерфейсный элемент; требование; управление.

DOI: 10.15514/ISPRAS-2017-29(5)-6

For citation: Лаврищева Е.М., Мутилин В.С., Рыжов А.Г.. Проектирование моделей вариабельности для программных, операционных систем и их семейств. *Труды ИСП РАН*, том 29, вып. 5, 2017 г., стр. 93-110 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(5)-6

Литература

[1]. Pohl K., Böckle G., van der Linden F. J. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, 2005. DOI: 10.1007/3-540-28901-1.

[2]. Bachmann F., Clements P. Variability in software product lines. CMU/SEI Technical Report CMU/SEI-2005-TR-012, 2005.

[3]. Lotufo R., She S., Berger T., Czarnecki K., Wąsowski A. Evolution of the Linux kernel variability model. Proc. of SPLC'10, LNCS 6287:136-150, Springer, 2010. DOI: 10.1007/978-3-642-15579-6_10.

[4]. Лаврищева Е.М., Грищенко В.Н. Методы и средства объектно-компонентного программирования. Кибернетика и системный анализ. 2003.-№1, с. 39-55.

[5]. Kang K., Cohen S., Hess J., Novak W., Peterson S. Feature-oriented domain analysis (FODA) feasibility study. CMU/SEI Technical Report CMU/SEI-90-TR-21, 1990.

[6]. Berger T., She S., Lotufo R., Wąsowski A., Czarnecki K. A study of variability models and languages in the systems software domain. IEEE Transactions on Software Engineering, 39(12):1611-1640, 2013. DOI: 10.1109/TSE.2013.34.

[7]. Zippel R. et al. Kconfig language. https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

[8]. Wang H., Li Y., Sun J., Zhang H., Pan J. A semantic web approach to feature modeling and verification. Proc. of Workshop on Semantic Web Enabled Software Engineering (SWESE'05), p. 44, 2005.

[9]. Лаврищева Е.М., Слабоспицькая О.А., Коваль Г.И., Колесник А.А. Теоретические аспекты управления вариабельностью в семействах ПС. Весник КНУ, серия физ.–мат. наук, 2011, № 1, стр. 151-158.

[10]. Berger T. Variability mining with LEADT. DOI TSE 2014.

[11]. Lavrischeva, E.: Formal Fundamentals of Component Interoperability in Programming. In: Cybernetics and Systems Analysis, vol. 46, no. 4, pp. 639–652. Springer, Heidelberg (2010) http://link.springer.com/article/10.1007/s10559-010-9240-z

[12]. Лаврищева Е.М., Грищенко В.Н. Сборочное программирование. Основы индустрии программных продуктов. К.: Наук. Думка, 2009, 371 с.

[13]. Ekaterina Lavrischeva, Andrey Stenyashin, Andriy Kolesnyk, Object-Component Development of Application and Systems. Theory and Practice, Journal of Software

Engineering and Applications, 2014, 7, Published Online August 2014 in SciReshttp://www.scirp.org/journal/jsea

[14]. Kolesnyk A., Slabospitskaya O. Tested Approach for Variability Management Enhancing in Software Product Line. – In: ICT in Education, Research and Industrial Applications: Integration and Knowledge, Proc. 8 –th Int. Conf. ICTERI 2012, CEUR –WS.org/Vol –848, ISSN 1613 –0073, urn:nbn:de:0074 –848-8. – P. 125 –133.

[15]. Колесник А.Л. Модели и методы разработки семейств вариантных программных систем.-Автореф.- КНУ.- 2013. 22 с.

[16]. Cohen M.B., Gibbons P.B., Mugridge W.B., Colbourn C.J. Constructing test suites for interaction testing. Proc. of 25-th Intl. Conf. on Software Engineering, pp. 38-48. IEEE, 2003. DOI: 10.1109/ICSE.2003.1201186.

[17]. Lotufo R., She S., Berger T., Czarnecki K., Wąsowski A. Evolution of the Linux kernel variability model. Proc. of SPLC'10, LNCS 6287:136-150, Springer, 2010. DOI: 10.1007/978-3-642-15579-6_10.

[18]. C.Käster, A. Dreiling and K. Ostermann's ,Variability Mining with LEADT/- work is supported by ERC grant #203099

[19]. Grindal M., Offutt A.J., Andler S.F. Combination testing strategies: a survey. Software Testing, Verification, and Reliability, 15(3):167-199, 2005. DOI: 10.1002/stvr.319.

[20]. Лаврищева Е.М., Слабоспицкая О.А. Подход к построению объектно-компонентной модели семейства программных продуктов. Проблемы программирования, 2013, №3, стр. 14–26 (укр.).

[21]. Лаврищева Е.М. Теория объектно-компонентного моделирования программных систем. Препринт ИСП РАН № 29, 2016. http://www.ispras.ru/preprints/docs/prep_29_2016.pdf.

[22]. Deming E. New economics for manufactures, governments and education, 1993.

[23]. Gruler A., Leucker M., Scheidemann K. Modeling and model checking software product lines. Proc. of IFIP Intl. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS), pp. 113-131. Springer, 2008. DOI: 10.1007/978-3-540-68863-1_8.

[24]. Ekaterina M. Lavrischeva. Assemblling Paradigms of Programming in Software Engineering, 2016, 9, pp.296-317, http://www.scrip.org/journal/jsea, http://dx.do.org/10.4236/ jsea.96021

[25]. Кулямин В.В. Лаврищева Е.М., Мутилин В.С., Петренко А.К. Верификация и анализ вариабельных операционных систем. Труды ИСП РАН. Том 28. вып. 3, стр. 189-209. DOI: 10.15514/ISPRAS-2016-28(3)-12

[26]. Лаврищева Е.М. Петренко А.К. Моделирование семейств программных систем. Труды ИСП РАН, 2016, том 28. вып. 6, стр. 180 -190. DOI: 10.15514/ISPRAS-2016-28(6)-4

[27]. C. Kästner, A. Dreiling, K. Ostermann, Variability Mining with LEADT, In Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE), pp. 157–166. 2009.

[28]. Korotun T.M. Methods and tools testing families system in resource-limited settings (ukr.), 2005, Autoref. Dis. ICNANU, 22 pages.

[29]. Лаврищева Е.М. Программная инженерия. Парадигмы, Технологии, CASE-средства программирования.2 изд. М: Юрайт, 2016, 280 с.

Подход к определению достижимости программных дефектов, обнаруженных методом статического анализа, при помощи динамического символьного исполнения[1]

А.Ю. Герасимов <agerasimov@ispras.ru>
Л.В. Круглов <kruglov@ispras.ru>
М.К. Ермаков <mermakov@ispras.ru>
С.П. Вартанов <svartanov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Среди методов анализа программ на наличие дефектов выделяют методы статического и динамического анализа. В данной статье мы предлагаем комбинированный подход, заключающийся в применении динамического символьного исполнения для определения достижимости дефектов, найденных при помощи статического анализа. Предлагаемый подход является развитием ранее предложенного подхода определения достижимости определенной инструкции в программе методами динамического символьного исполнения, примененном последовательно для нескольких точек в программе, включающих точки инициализации дефекта, условные переходы в трассе дефекта и точку реализации дефекта. С начала производится статический анализ исполняемого кода программы с целью выделения путей исполнения, которые приводят к точке инициализации дефекта. Далее производится вычисление входных данных, приводящих к точке инициализации дефекта методом динамического символьного исполнения и прохождения базовых блоков, лежащих на трассе дефекта, включая точку реализации дефекта. Выбор наиболее перспективного пути для исполнения программы производится при помощи метрики минимального расстояния от пути исполнения на предыдущей итерации до следующей точки на трассе дефекта. Метрика вычисляется на основе путей в графе вызовов в программе, расширенного графом потока управления функций на путях исполнения, приводящих к реализации дефектов. Предлагаемый подход был проверен на нескольких программах с открытым исходным кодом из комплекта утилит командной строки операционной системы Debian Linux. В результате экспериментальной проверки было подтверждена возможность применения данного подхода к классификации дефектов, найденных при

[1] Исследование проводится в рамках научно-исследовательских работ Института системного программирования им. В.П. Иванникова РАН в 2014-2017 годах

помощи статического анализа программ. Также, были обнаружены некоторые ограничения, препятствующие внедрению данного подхода в промышленные инструменты анализа. Одним из направлений дальнейших исследований может быть выбрано исследование подходов к снятию этих ограничений.

Ключевые слова: статический анализ программ; динамический анализ программ.

DOI: 10.15514/ISPRAS-2017-29(5)-7

Для цитирования: Герасимов А.Ю., Круглов Л.В., Ермаков М.К., Вартанов С.П. Подход определения достижимости программных дефектов, обнаруженных методом статического анализа программ, при помощи динамического анализа. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 111-134. DOI: 10.15514/ISPRAS-2017-29(5)-7

1. Введение

Сегодня постоянно растёт сложность разрабатываемого программного обеспечения, в связи с чем задача автоматического обнаружения и воспроизведения программных дефектов становится весьма актуальной. По одной из возможных классификаций методов анализа программ все подходы можно разделить на две большие группы: методы статического анализа и методы динамического анализа.

Статический анализ производится без запуска программы на исполнение и как правило по некоторой семантической модели программы, построенной по исходному или исполняемому коду. Существует несколько приложений статического анализа программ, таких как сбор метрик [1] и построение выводов о качестве программного обеспечения на их основе, генерация тестовых сценариев [2]. Так или иначе методы статического анализа программ предназначены для обеспечения качества программных продуктов, в связи с чем методы обнаружения дефектов и уязвимостей безопасности в программах получили наиболее широкое распространение в индустрии программной инженерии [3, 4, 5]. Стоит отметить, что методы статического анализа недостаточно точны в обнаружении ошибок [6, 7], и связано это в первую очередь с обеспечением требований масштабируемости и производительности анализа [8, 9].

Динамический анализ производится либо в процессе исполнения программы (online-анализ) [10, 11, 12], либо после завершения работы программы (offline или post-mortem анализ) [13]. Среди методов динамического анализа программ стоит выделить динамическое символьное исполнение [14], также известное как конкретно-символьное (concolic – concrete and symbolic) [15], которое совмещает реальное исполнение программы с символьным исполнением [16]. Динамическое символьное исполнение позволяет применять техники исследования путей программы [17] и, добавляя предикаты безопасности к ограничениям пути (path constraint), проверять потенциально опасные операции на наличие реальных ошибок в программах. Динамическое символьное исполнения применяется для таких задач как генерация тестовых

покрытий, анализ утечек критических данных в программе, автоматическая генерация фильтров для входных данных с целью предотвращения эксплуатации уязвимостей в программах, обнаружение ошибок и уязвимостей в программах [18]. Наиболее серьезным ограничением для применения методов чистого динамического символьного исполнения для анализа программ является проблема экспоненциального взрыва количества путей для анализа (path explosion). Каждый условный переход в программе, зависящий от входных данных, потенциально увеличивает количество путей для анализа в два раза, что в конечном итоге приводит к резкому росту количества путей для анализа и невозможности исчерпывающего (exhaustive) анализа реальных программ.

В связи с указанными выше известными ограничениями статического анализа и динамического символьного исполнения, примененными в последнее время развиваются комбинированные подходы, позволяющие компенсировать недостатки каждого из подходов в отдельности. В работе [19], посвященной описанию инструмента Check'n'Crash, рассматривается подход совмещения статического анализа кода программ на языке Java с целью обнаружения критических ошибок времени исполнения и целенаправленной генерации тестовых сценариев для подтверждения найденных ошибок в процессе запуска сгенерированных тестов. В работе [20] описывается подход к улучшению производительности генерации тестов для ошибок, найденных в программах при помощи статического анализа, путём выделения срезов (slice), относящихся только к потоку данных, влияющих на найденную ошибку. В работе [21] описывается инструмент CONBOL, котором метод статического анализа используется для обнаружения ошибок в функциях программы и генерации тестовых сценариев для демонстрации найденных ошибок. В работе [22] рассматривается инструмент HVDS, который совмещает обнаружение подозрительных мест в программе методом статического анализа и генерацию входных данных для прохождения по пути, приводящем к найденным подозрительным местам. В работе [23], посвященной инструменту DSD-Crasher, который развивает идеи Check'n'Crash, описывается подход к обнаружению критических ошибок времени исполнения, приводящих к аварийному завершению программы путем совмещения статического анализа программ на языке Java, который получает ограничения на значения входных данных на основе извлечения инвариантов поведения программы из имеющихся тестов программы и автоматической генерации тестовых сценариев для проверки истинности найденных программных ошибок. В работе [24] предлагается совмещения статического анализа программы для генерации регулярных выражений, описывающих ограничения на входные данные программы, генетического алгоритма для генерации входных данных программы для обнаружения критических ошибок времени исполнения программы. В работе [25] предлагается метод совмещающий статический анализ на основе абстрактной интерпретации

программы и генерацию тестовых наборов для подтверждения дефектов в программе. В работе [26] предлагается подход к совмещению статического анализа программ для операционной системы Android с целью обнаружения последовательности событий операционной системы, приводящей к исполнению определенных частей кода программ, и динамического анализа программы направляемого этой информацией с целью генерации последовательности событий, приводящих к исполнению программы по критическим с точки зрения безопасности путям в программе (с динамической загрузкой кода или вызовами к методам операционной системы). В работе [27] рассматривается инструмент, совмещающий динамическое символьное исполнение программ на языке C#, которое в процессе построения тестового покрытия программы одновременно проверяет запрещенные состояния, заранее найденные в процессе статического анализа программы, обнаруживающего нарушение контрактов вызова функций.

В данной статье мы рассматриваем подход совмещения статического анализа исходного кода программ с целью обнаружения потенциальных дефектов, статического анализа бинарного кода программ с целью построения возможных путей до места потенциального дефекта и вычисления входных данных для достижения потенциальных дефектов с учётом прохождения трассы дефекта при помощи динамического символьного исполнения. Статья организованна следующим образом: в разделе 2 даётся краткий обзор понятий и подходов к анализу программ, в разделе 3 описывается предлагаемый нами подход к построению входных данных для подтверждения достижимости дефектов, в разделе 4 приводится описание и результаты экспериментальной проверки предложенного подхода, в разделе 5 делаются выводы и предлагаются дальнейшие направления исследований в области анализа программ на наличие дефектов.

2. Обзор базовых понятий

2.1 Дефекты с трассой исполнения

Среди дефектов, обнаруживаемых статическими анализа торами стоит выделить группу дефектов, которые называются дефектами с трассой исполнения или дефектами типа «инициализация-реализация». На Рис. 1 приведен пример программы с дефектом такого типа.

```
void * alloc_buffer(size_t size){
    return malloc(sizeof(int) * size);
}
void mlk(size_t number, int * input){
    int *buffer = alloc_buffer(number);
    memcpy(buffer, input, number);
    if(number < 2) {
```

```
            return; // Memory leak
        }
        ...
        // Processing of data
        ...
        free(buffer);
    }
```

Рис. 1. Пример фрагмента программы с дефектом типа «инициализация-реализация»

Fig. 1. Example of a program fragment with an initialization-implementation type defect

Ошибочная ситуация инициализируется в функции `alloc_buffer` путём выделения памяти. Далее в зависимости от вычисления условия `if(number < 2)`, которое в данном случае попадает как событие в трассу дефекта, происходит переход на оператор возврата из функции, на котором происходит реализация дефекта утечка памяти в связи с потерей значения локального указателя `buffer`.

Таким образом дефекты типа «инициализация-реализация» определяются следующими характеристиками:

- *точка инициализации* дефекта, где происходит инициализации ошибочной ситуации;
- *точка реализации* дефекта, где происходит ошибочная ситуация;
- *трасса* от точки инициализации до точки реализации ошибочной ситуации, если какие-либо специфические события зарегистрированы статическим анализатором между точками инициализации и реализации дефекта.

Разыменование нулевого указателя, разыменование неинициализированного указателя, использование непроверенных входных данных в уязвимой функции, утечка памяти, утечка ресурса, выход за границы буфера в памяти являются дефектами типа «инициализация-реализация» и обнаруживаются при помощи анализа, чувствительного к пути исполнения в программе.

2.2 Путь исполнения

Путь исполнения в программе представляет собой последовательность инструкций от точки входа до точки останова программы, исполненных процессором. Если поведение программы детерминировано, то путь исполнения может быть описан последовательностью условных переходов, при этом для каждого пути исполнения может существовать от нуля до нескольких различных наборов входных данных. При чём каждый из них уникально идентифицирует единственный путь исполнения. Для генерации

входных данных в процессе итеративного динамического символьного исполнения программы требуется собрать трассу исполнения, в виде инструкций, исполненных процессором, чтобы получить достаточно информации для определения нового пути исполнения программы.

На рис. 2 представлен пример программы, в которой дефект разыменования нулевого указателя реализуется, если в качестве первого аргумента командной строки будет передана строчка `'bad'`.

```
int main(int argc, char* argv[]){
    char* buffer = NULL;
    if(argv[1][0] == 'b')
        if(argv[1][1] == 'a')
            if(argv[1][2] == 'd')
                printf(”%s\n”, buffer);
    return 0;
}
```

Рис. 2. Пример пути исполнения, приводящего к дефекту в программе
Fig. 2. An example of the execution path leading to a defect in the program

Для подтверждения достижимости дефекта в программе методом итеративного динамического символьного исполнения необходимо сгенерировать такой набор входных данных, который приведет к исполнению программы по пути, содержащему точку инициализации дефекта, точки событий трассы дефекта и точку реализации дефекта.

2.3 Итеративное динамическое символьное исполнение

В процессе итеративного динамического символьного исполнения программа запускается множество раз и на каждой итерации запуска анализируется новый путь в программе. Это достигается применением метода чередования путей (path alternation), основанном на инвертировании направления перехода в одном из условных переходов, зависящем от входных данных. Путь в программе определяется системой формул, описывающей ограничения над областями памяти и регистрами, содержащими помеченные (tainted) данные, то есть данные полученные из внешних источников, таких как аргументы командной строки, стандартные потоки ввода вывода, файлы и других. Будем называть такую систему формул набором ограничений пути (path constraint).

Для построения системы формул ограничений пути требуется отслеживать поток операций над помеченными данными в программе в процессе её исполнения. Для этого требуется отслеживать все операции чтения из источников внешних данных программы и операции перезаписи ранее прочитанных помеченных значений до тех пор, пока они не будут перезаписаны непомеченными (внутренними) данными программы. Поскольку каждый путь в программе определяется множеством условных переходов, условия которых зависят от внешних данных программы, то они

также должны быть записаны в формулу ограничений пути. Новый путь в программе может быть получен инвертированием условий в исходной формуле ограничений пути. Предположим в трассе есть N условных переходов, зависящих от внешних данных программы. Соответственно N новых наборов входных данных для N путей в программе может быть сгенерировано по результатам исполнения по данному пути. Трасса номер i будет соответствовать i-*тому* условному переходу, все условия до условия i должны остаться неизменными, а условия после i должны быть исключены из формулы ограничений пути. Проверка системы формул ограничений на выполнимость позволит, в случае выполнимости, подобрать значения нового набора входных данных, соответствующего новому пути исполнения в программе.

На первом шаге итеративного символьного исполнения программа запускается с некоторым начальным значением набора входных данных (зерном) и строится начальная трасса исполнения программы. После того, как условия условных переходов, зависящих от входных данных, будут инвертированы и будут сгенерированы новые наборы входных данных, каждый из которых будет оценен при помощи некоторой эвристики, показывающей перспективность анализа программы по соответствующему набору пути исполнения. Каждому набору входных данных присваивается оценка на основе эвристики. Набор входных данных с лучшей оценкой выбирается для запуска программы на следующей итерации. Все остальные наборы сохраняются для использования на последующих итерациях. Когда условие в определенном условном переходе инвертировано, все условия условных переходов до него должны быть проигнорированы в процессе инвертирования на следующей итерации анализа при запуске на соответствующем наборе входных данных. Для этого инструменту анализа вместе с набором входных данных для анализируемой программы передается номер условного перехода, начиная с которого необходимо проводить инвертирование условного выражения для получения следующих наборов входных данных. Среди эвристик можно выделить следующие: случайный выбор, интересен наиболее глубокий путь, интересен путь с максимальным приростом проанализированных базовых блоков и другие.

3. Описание подхода

3.1 Обзор

В качестве входных данных для предлагаемого подхода предлагается использовать предупреждение о потенциальном дефекте в программе, сгенерированное инструментом статического анализа или аналитиком, а также исполняемый код программы, собранный из того же исходного кода, который был обработан статическим анализатором. Предупреждение может содержать

детальную информацию о дефекте, такую как точку инициализации, трассу и точку реализации дефекта. В вырожденном случае оно должно содержать точку реализации дефекта. Сама задача подтверждения достижимости дефекта может быть разделена на несколько подзадач.

Во-первых, происходит преобразование трассы дефекта в исходном коде в трассу дефекта в исполняемом коде. Такое преобразование требуется в связи с тем, что в процессе динамического символьного исполнения производится анализ исполняемого кода программы и использование информации об исходном коде на данном уровне не представляется целесообразным. В то же время наличие оптимизаций кода может серьезно изменить структуру исполняемого кода по сравнению с исходным кодом, в связи с чем в описываемом подходе предполагается отсутствие оптимизаций исполняемого кода.

Во-вторых, для определения достижимости дефекта типа «инициализация-реализация» требуется проанализировать все пути исполнения в программе, которые проходят через точку инициализации дефекта при чем максимально быстрым способом. Для решения этой задачи предлагается использовать итеративное динамическое символьное исполнение для генерации наборов входных данных и ввести новую эвристику определения наиболее перспективного пути для анализа, которая будет рассчитывать минимальное расстояние от пути исполнения программы на текущей итерации до точки инициализации дефекта с использованием графа потока управления анализируемой программы [28]. Данный граф строится перед началом динамического символьного исполнения методом статического анализа исполняемого кода.

После того, как точка инициализации дефекта достигнута итеративно используется направленное динамическое символьное исполнение для прохождения по трассе дефекта от точки инициализации дефекта, через точки событий в трассе дефекта до точки инициализации дефекта. Если удалось построить набор входных данных такой, что исполнение программы проходит через всю трассу дефекта, включая точки инициализации и реализации ошибочной ситуации, то считается, что достижимость дефекта подтверждена. В обратном случае итеративное динамическое символьное исполнение должно быть продолжено до тех пор, пока для всех известных путей, приводящих исполнение в точку инициализации дефекта, не будет проведен анализ достижимости дефекта.

3.2 Эвристика выбора кратчайшего пути

Для быстрого достижения точки инициализации дефекта предлагается использовать следующую эвристику: из всех возможных путей исполнения программы, проходящих через заданные точки, на каждом шаге итеративного динамического символьного исполнения выбирается такой путь, в котором количество условных переходов наименьшее. Кратчайший путь до точки

инициализации дефекта вычисляется на графе потока управления программы, полученного методом статического анализа исполняемого кода программы до начала итеративного динамического символьного исполнения программы.

Чтобы собрать всю требующуюся информацию, которая будет использоваться при расчете эвристики кратчайшего пути, достаточно построить сокращенный граф вызовов программы, включающий только те функции, через которые проходит путь исполнения до точки инициализации дефекта, через точки событий трассы дефекта и точку реализации дефекта. Этот граф мы строим методом обратной трассировки от точки инициализации дефекта. Сначала определяются все точки вызова функции, содержащей точку инициализации дефекта. Затем определяются точки вызова функций, вызывающих функцию, содержащую точку инициализации дефекта, и так до тех пор, пока не будет найдена функция, не вызывающаяся из других функций программы (например, главная функция программы, являющаяся точкой входа в программу). Если в процессе построения графа вызовов встречается рекурсия, то соответствующие функции пропускаются, как уже включенные в граф вызовов.

После построения неполного графа вызовов программы для каждой функции строится неполный граф потока управления функции, содержащий только те базовые блоки и условные переходы, которые лежат на путях, приводящих к точке инициализации дефекта. После построения такого сокращённого графа потока управления программы для каждого направления условного перехода в этом графе проставляется расстояние (количество условных переходов) до точки инициализации дефекта. Далее будем называть такой сокращенный и взвешенный граф потока управления программы графом достижения точки инициализации дефекта.

На рис. 3 изображен граф достижения точки инициализации дефекта в функции `target`. Расстояние от точки вызова функции target из функции transit до точки инициализации дефекта равно 0, поскольку мы исключаем ребра безусловных переходов на графе потока управления, расстояние от точки входа в функцию transit до точки инициализации дефекта равно 2, расстояние от точки входа в функцию main до точки инициализации дефекта равно 5 и 6 соответственно двум путям в функции main.

В процессе вычисления эвристики минимального расстояния мы принимаем во внимание только те базовые блоки, которые лежат на пути, ведущем к точке инициализации дефекта, потому что все остальные базовые блоки не доминируют над точкой инициализации дефекта и могут быть проигнорированы.

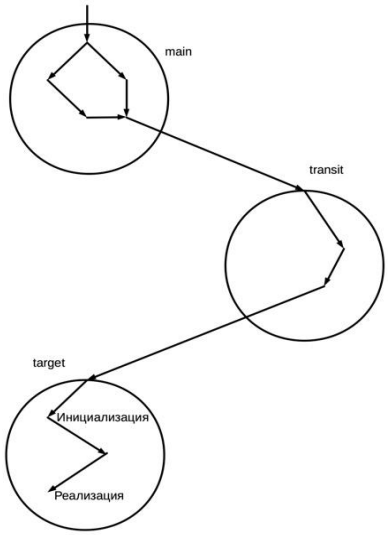


Рис. 3. Граф достижения точки инициализации дефекта
Fig. 3. The graph of reaching a point of defect initialization

Предположим, что на пути исполнения программы в процессе некоторой итерации есть три условных перехода *A*, *B* и *C*, из которых возможно достижение точки инициализации дефекта. Тогда будет выбран тот условный переход, альтернативная ветвь которого будет иметь наименьшую оценку среди других:

```
E = min(distance(C, T), distance (B,T), distance(A, T))
```

где, `T` – точка инициализации дефекта, а `distance(X,T)` функция расчёта расстояния от узла `X` до точки инициализации дефекта в графе достижения точки инициализации дефекта. Результатом вычисления эвристики для некоторого пути исполнения программы является расстояние от ближайшего к точке инициализации дефекта условного перехода на пути исполнения программы. Расстояние становится равным нулю, когда точка инициализации дефекта достигнута.

3.3 Трансформация пути дефекта

После того как точка инициализации дефекта достигнута, производится направленный анализ, который проводит исполнение программы по трассе дефекта. Для этого требуется преобразовать трассу дефекта, выраженную в терминах исходного кода в трассу дефекта в исполняемом коде, чтобы её можно было использовать в процессе итеративного динамического символьного исполнения. В рамках нашей реализации трансформация производилась для исходной трассы в программе на языке Си и исполняемого

кода, сгенерированного компиляторами GCC [29]. Реализация была протестирована для версий GCC 4.6.3 и 4.9.2 для платформы x86-64. Процесс трансформации для условного оператора описан ниже. Циклы, тернарный оператор, оператор-переключатель и другие в данной работе не рассматриваются и включены в список направлений дальнейшего исследования.

Исходя из того, что исполняемый код программы не подвергался оптимизации, последовательность условных переходов в исполняемом коде будет соответствовать последовательности конъюнктов в условных выражениях условных операторов в исходном коде.

Условный переход в исполняемом коде это либо переход на базовый блок, соответствующий `then`-ветке или базовый блок, соответствующий `else`-ветке в исходном коде программы. Возможные варианты в исполняемом коде для условного оператора показаны на рис. 4.

```
if(A)
{
    // then-ветка
}
else
{
    // else-ветка
}
```

`; Вариант 1: переход`
`; на then-ветку`
 	
`4004fc: cmp $0x0, -`
`0x3(%rbp)`
`400500: jne 400515`
`…`
`400515: ; then-ветка`
	`; Вариант 2: переход`
	`; на else-ветку`
	
	`4004fc: cmp $0x0, -`
	`0x3(%rbp)`
	`400500: je 400515`
	`…`
	`400515: ; else-ветка`

Рис 4. Пример вариантов генерации кода для условного оператора
Fig. 4. Example of code generation options for a conditional statement

Правила генерации кода для сложных условий с конъюнкциями, дизъюнкциями и отрицаниями приведены на рис. 5.

A && B	• переход на then-ветку: вариант 2 для A и вариант 1 для B; • переход на else-ветку: вариант 2 для A и вариант 2 для B
A || B	• переход на then-ветку: вариант 1 для A и вариант 1 для

	B; • переход на else-ветку: вариант 1 для A и вариант 2 для B
!A	• переход на then-ветку: вариант 2 для A; • переход на else-ветку: вариант 1 для A.

Рис 5. Варианты генерации неоптимизированного кода для условного оператора в компиляторе GCC версии 4.6.3 и 4.9.2

Переход на `then`-ветку или `else`-ветку зависит от операций внутри `then`- или `else`- блоков и формы условного выражения. Например, было обнаружено, что:

- переход на then-блок генерируется если в соответствующем блоке исходный код содержит операцию безусловного перехода: `goto`, `break` или `continue`;
- в то же время, переход на `else`-ветку генерируется если в условном операторе нет `else`-ветки и содержится внутри `then`-ветки вышестоящего условного оператора (при этом не обязательно непосредственного вышестоящего);
- переход на `else`-ветку происходит во всех остальных случаях.

После того как трансформация пути из исходного кода в исполняемый код произведена может оказаться, что в исполняемом коде несколько путей соответствуют одному пути в исходном коде. Например, на рис. 6 приведена дизъюнкция `A || B` которая должна быть вычислена в true, которой в исполняемом коде будет соответствует два пути исполнения:

- путь, соответствующий условию `A` – истинно;
- путь, соответствующий условию `A` – ложно, `B` – истинно.

```
if(A || B) { ... } else { ... }
```

```
4004f6: cmp $0x0, -0x4(%rbp)
4004fa: jne 400502                    ; A
4004fc: cmp $0x0, -0x3(%rbp)
400500: je 400515                     ; B
400502: ; then-ветка
…
400515: ; else-ветка
```

Рис. 5. Пример ситуации, где один путь в исходном коде соответствует двум путям в исполняемом коде

Fig. 5. A case where one path in the source code corresponds to two paths in executable code

Поэтому, трансформация одного пути в терминах исходного кода может привести к нескольким путям исполнения в исполняемом коде и могут существовать несколько наборов входных данных, удовлетворяющих одному сообщению о дефекте.

3.4 Воспроизведение пути инициализация-реализация

Когда точка инициализации потенциального дефекта достигнута, происходит применение направленного динамического символьного исполнения для исполнения пути, на котором лежат события трассы потенциального дефекта. Это позволяет сократить время достижения точки реализации потенциального дефекта. В процессе направленного динамического символьного исполнения по трассе дефекта собираются условия прохождения пути для проверки выполнимости результирующего пути и генерации входных данных и воспроизведения генерации входных данных, которые позволят провести исполнение по трассе дефекта и показать его достижимость.

Стоит отметить, что модель отслеживания потока входных данных, описанная в пункте 2.3, эффективна, но не полна: существуют неявно помеченные данные, которые модель игнорирует, учитывая только явные зависимости по входным данным. Для генерации входных данных необходимо учитывать поток явно помеченных данных, но условие на пути, которое необходимо инвертировать может зависеть как от помеченных, так и от непомеченных данных. В связи с этим инвертирование условий, зависящих от помеченных и от непомеченных данных необходимо производить раздельно. Инвертирование помеченных условных переходов приводит к генерации нового набора входных данных и исполнению нового пути. Однако, инвертирование непомеченных условий бесполезно, так как в модели нет связи между непомеченными условными переходами и входными данными, в связи с чем инвертирование непомеченного условия не приведет к генерации нового набора входных данных. Если трасса дефекта содержит событие, связанное с условным переходом, не зависящим от помеченных данных, результат направленного анализа может оказаться некорректным. При этом возможно три ситуации:

- условный переход не зависит от входных данных, но должен быть инвертирован;
- условный переход неявно зависит от входных данных, но условие не должно быть инвертировано;
- условный переход неявно зависит от входных данных, но условие не должно быть инвертировано.

В первом случае можно заключить, что данный путь от инициализации до реализации дефекта несовместен. Второй случай соответствует ситуации, когда условие уловного перехода в трассе дефекта соответствует актуальному

значению и путь оказывается совместным и, таким образом, достижимость дефекта подтверждается. В третьем случае возможны следующие варианты:

- инвертированное условие не совместно с предусловием пути и результат направленного динамического символьного исполнения некорректен;
- реализация дефекта на самом деле не зависит от инвертированного условия, которое добавлено в трассу инструментом статического анализа по ошибке, и результат динамического символьного исполнения корректен.

Поэтому требуется ручная проверка дефекта и сгенерированных входных данных, если непомеченный условный переход был инвертирован в процессе динамического символьного исполнения.

Результатом выполненного направленного динамического символьного исполнения будет система формул, описывающая ограничения пути в терминах данных, которыми оперирует программа. Если условие условного перехода инвертируется, тогда новая формула должна быть добавлена в систему

```
tainted_condition_var != old_condition
```

где `tainted_condition_var` – символьная переменная, содержащая условие условного перехода, а `old_condition` – значение условия до инвертирования. Если новая система формул неразрешима, то путь исполнения, соответствующий инвертированному условному переходу невыполним и требуется анализ следующего пути из списка путей, достигающих точки инициализации дефекта.

4. Оценка предложенного подхода

Предложенный подход к определению достижимости программных дефектов, найденных методами статического анализа исходного кода, был реализован на основе инструмента динамического символьного исполнения программ *Avalanche* [30], разработанного в ИСП РАН. В инструмент *Avalanche* были внесены следующие изменения:

- добавлен алгоритм расчёта эвристики оценки пути на основе минимального расстояния до интересующей точки в программе;
- изменен модуль распространения помеченных данных с целью реализации алгоритма направленного динамического символьного исполнения;
- реализован модуль извлечения сокращенного графа вызовов и сокращенного графа потока управления функций, который представляет собой разборщик дизассемблированного представления программы, полученного при помощи инструмента `objdump` [31];

- реализован алгоритм трансформации трассы дефекта на основе обработчика дерева абстрактного синтаксиса, которое предоставляется инструментом статического анализа исходного кода программы.

Реализация была протестирована на наборе программ с открытым исходным кодом из комплекта поставки Debian Linux:

a2ps – утилита конвертации файлов в различных форматах в PostScript документ

bbe – утилита для редактирования двоичных файлов

bc – интерпретатор алгебраических выражений с неограниченной точностью

byacc – генератор LALR(1) парсеров

bzip2 – утилита для сжатия файлов

cabextract – инструмент распаковки Microsoft cabinet файлов (.cab)

ccrypt – утилита кодирования и раскодирования файлов и потоков

chrpath – утилита редактирования путей к библиотекам в ELF-файлах

cpio – утилита сжатия и извлечения архивов в различных форматах

discount – программа трансформации текстовых файлов в формат Markdown

Табл. 1 содержит данные по предупреждениям о дефектах на проектах, найденных референсным инструментом статического анализа исходного кода программ на языке Си. Предупреждения были размечены на истинные (T), ложные (F) и подтвержденные (C) методом направленного динамического символьного исполнения. Разметка предупреждений о дефектах в программе на истинные и ложные произведена вручную. Всего было классифицировано 348 предупреждений о дефектах, 113 из них признаны ложными, 235 – истинными. Референсный инструмент статического анализа исходного кода программ был настроен на поиск дефектов типа «инициализация-реализация» следующих типов:

- разыменование нулевого указателя (NPD);
- выход за границы буфера в памяти (BOF);
- утечка ресурса (LEAK);
- использование входных данных в уязвимых функциях без проверки (TAINT);

- использование неинициализированных значений (UNINIT).

Табл. 1 Результаты экспериментальной проверки подхода. F – ложные предупреждения, T – истинные предупреждения, C – подтвержденные предупреждения
Table. 1 Results of experimental verification of the approach. F - false warnings, T - true warnings, C - confirmed warnings

Project	NPD			BOF			LEAK			TAINT			UNINIT			TOTAL		
	F	**T**	**C**	**F**	**T**	**C**	**F**	**T**	**C**	**F**	**T**	**C**	**F**	**T**	**C**	**F**	**T**	**C**
a2ps	65	33	2*	5	2		2	22	11		7	2				72	64	15
bbe		2			10			1									13	0
bc							1	6	4		1	1				1	7	5
byacc				10				1						2		10	3	0
bzip					1			14			9			1			25	0
cabextract					2			3			1	1					6	1
ccrypt							7	15	2		1					7	16	2
cgrpath								9			10	3					19	3
cpio	19	22					2	18	2		4		1	1		22	45	2
discount		37					1				1					1	38	0
Total	84	94	2*	15	15	0	13	89	19	0	34	7	1	4	0	113	235	28

Предложенный подход был применен к каждому из предупреждений о дефекте от инструмента статического анализа исходного кода программ. Инструмент направленного динамического символьного исполнения запускался с 30-ти минутным ограничением по времени. Всего для 28 предупреждений о дефектах была подтверждена достижимость. Ни для одного из дефектов типа использование неинициализированного значения и выход за границы буфера не была подтверждена достижимость. Стоит отметить, что два подтвержденных дефекта типа разыменование нулевого указателя оказались подтверждены ложно. Связано это с имеющимися ограничениями в реализации предложенного подхода: отсутствием добавления в систему формул ограничений пути предиката безопасности. В обоих случаях ложного подтверждения дефекта требуется добавление ограничения вида:

```
variable == NULL
```

которое соответствовало бы строчке исходного кода:

```
variable = someFunction(arguments);
```

где функция `someFunction` ни на одном из выполнимых путей не возвращает `NULL`. Это означает, что достижимость точки инициализации дефекта, прохождение по трассе дефекта и достижимость точки реализации дефекта ещё не означает подтверждение наличия дефекта в программе. Пока

не реализована функциональность добавления предиката безопасности для проверки реализации дефекта, данное ограничение подхода будет ложно подтверждать достижимость нереализуемых дефектов.

Для остальных предупреждений о дефектах, которые классифицированы как истинные, достижимость была подтверждена. Таким образом 26 дефектов были подтверждены верно. Для 9 предупреждений о дефектах потребовалась ручная проверка в связи с наличием в трассе события, связанного с непомеченным условием в условном переходе. Все они были классифицированы как ложные.

5. Заключение

В данной статье описан подход для подтверждения при помощи динамического символьного исполнения достижимости дефектов типа инициализация-реализация, найденных при помощи статического анализа исходного кода. Экспериментальная проверка реализации подхода показала применимость данного подхода для подтверждения достижимости дефектов на наборе реальных программ с открытым исходным кодом.

По результатам реализации подхода и экспериментальной проверки можно выделить несколько направлений для дальнейших исследований с целью улучшения применимости предложенного подхода на практике:

- исследование алгоритмов минимально достаточной помеченности потока данных программы, с учётом неявных зависимостей по данным и по управлению;
- исследование и реализация подхода определения предиката безопасности для различных дефектов с целью подтверждения дефекта, а не только его достижимости, методами итеративного динамического символьного исполнения;
- поддержка других типов дефектов в программах.

Список литературы

[1]. Vogelsang A., Fehnker A., Huuck R., Reif W. Software Metrics in Static Program Analysis. ICFEM'10 Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering, Shanghai, Shina, November 17-19, 2010, pp. 485-500

[2]. Kim Y., Kim Y., Kim T., Lee G., Jang Y., Kim M. Automated unit testing of large industrial embedded software using concolic testing. ACE'13 Proceedings of the 28yh IEEE/ACM International Conference on Automated Software Engineering, Silicaon Valley, CA, USA, November 11-15, 2013, pp. 519-528

[3]. Xie Y., Chou A., Engler D. ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors. ESEC/FSE-11 Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International

Symposium on Foundations of Software Engineering, Helsinki, Finland, September 01-05, 2003, pp. 327-336

[4]. Bessey A., Block K., Chelf B, Chow A., Fulton B., Hallem S., Henri-Gros C., Kamsky A., McPeak S., Engler D. A few billion lines of code later: using static analysis to find bugs in the real world. Communications od ACM, vol. 53, issue 2, February 2010, pp. 66-75

[5]. Иванников В.П., Белеванцев А.А., Бородин А.Е., Игнатьев В.Н., Журихин Д.М., Аветисян А.И., Леонов М.И. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, вып. 1, 2014, стр. 231-250. DOI: 10.15514/ISPRAS-2014-26(1)-7

[6]. Engler D., Chelf B., Chou A., Hallen S. Checking system rules using system-specific, programmer-written compiler extensions. OSDI'00 Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, vol. 4, article No.1, San-Diego, CA, USA, October 22-25, 2000

[7]. Johnson B., Song Y., Murphy-Hill E., Bowdidge R., Why don't software developers use static analysis tools to find bugs? ICSE'13 Proceedings of the 2013 International conference on Software Engineering. San Francisco, CA, USA, May 18-26, 2013

[8]. Christakis M., Müller P., Wüstholz An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzers. VMCAI'15 International Workshop on Verification, Model Checking and Abstract Interpretation, Springer, 2015, pp. 336-354

[9]. Livshits B., Sridharan M., Smaragdakis Y., Lhoták O., Amaral J.N., Chang B.-Y. E., Guyer S.Z., Khedker U.P., Møhler A., Vardoulakis D. In defence of soundness: a manifesto. Communications of ACM, vol. 58, no. 2, February 2015

[10]. Cadar C., Dunbar D., Endger D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems. OSDI'08 Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation, San Diego, CA, USA, December 08-10, 2008, pp. 209-224

[11]. Averginos T., Cha S.K., Revert A., Schwartz E.J., Woo M., Brumley D. Automatic Exploit Generation. Communications of the ACM, volume 57, issue 2, February 2014, pp. 74-84

[12]. Chipunov V., Kuznetsov V., Candea G. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems (TOCS) – Special Issue APLOS 2011, article no. 2, volume 30, issue 1, February 2012

[13]. Manevich R., Sridharan M., Adams S., Das M., Yang Z. PSE: explaining program failures via post-mortem static analysis. SIGSOFT'04/FSE-12 Proceedings of the 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering, Newport Beach, NY, USA, 2004, pp. 63-72

[14]. Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M.G., Liang Z., Newsome J., Poosankam P., Saxena P. BitBlaze: a New Approach to Computer Security via Binary Analysis. ICISS'08 Proceedings of the 4th international Conference on Information Systems Security, Hydarabad, India, December 16-20, 2008, pp. 1-25

[15]. Sen K., Marinov D., Agha G. CUTE: a concolic unit testing engine for C. ESEC/FSE-13 Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, September 05-09, 2005, pp. 263-272

[16]. King J.C. Symbolic Execution and Program Testing. Communications of the ACM, volume 19, issue 7, 1976, pp. 385-394

[17]. Cadar C., Ganesh V., Pawlowski P., Dill D.L., Engler D.R. EXE: automatically generating inputs of death. CCS'06 Proceedings of the 13th ACM Conference on Computer and Communications Security, Alexandria, Virginia, USA, October 30 - November 03, 2006, pp. 322-335

[18]. Schwartz E.J., Averginos T., Brumley D. All You Ever Wanted to Know About Dynamic Tait Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). SP'10 Proceedings of the 2010 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 16-19, 2010, pp. 317-331

[19]. Csallner C., Smaragdakis Y. Check'N'Crash: combining static checking and testing. ICSE'05 Proceedings of the 27th international conference on software engineering, St. Louis, MO, USA, May 15-21, 2005, pp. 422-431

[20]. Chebaro O., Kosmatov N., Giorgetti A., Julliand J. Programs slicing enhances a verification technique combining static and dynamic analysis. SAC'12 Proceedings of the 27th Annual ACM Symposium of Applied Computing, Trento, Italy, March 26-30, 2012, pp. 1284-1291

[21]. Kim T., Park J., Kulinda I., Jang Y. Concolic Testing Framework for Industrial Embedded Software. APSEC'14 Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference, volume 2, Jeju, South Korea, December 01-04, 2014, pp. 7-10

[22]. Hanna A., Ling H.Z., Yang X., Debbabi M. A synergy between static and dynamic analysis or the detection of software security vulnerabilities. OTM'09 Proceedings of the Confederated International Congress, CoopIS, DOA, IS and ADBASE 2009 on the Move to Meaningful Internet Systems: part II. Vilamoura, Portugal, November 01-06, 2009, pp. 815-832

[23]. Csallner C, Smaragdakis Y. DSD-Crasher: a hybrid analysis tool for bug finding. IISTA'06 Proceedings of the 2006 International Symposium on Software Testing and Analysis. Portland, Maine, USA, July 17-20, 2006, pp. 245-254

[24]. Artho C, BIere A. Combined Static and Dynamic Analysis. Electronic Notes in Theoretical Computer Science (ENTCS), volume 131, May, 2005, pp. 3-14

[25]. Chebaro o., Kostomarov N., Giorgetti A., Julliand J. Combining static analysis and test generation for C program debugging. TAP'10 Proceedings of the 4th International Conference on Tests and Proofs. Málaga, Spain, July 01-02, 2010, pp. 94-100

[26]. Schütte J., Fedler R., Tetze D. ConDroid: targeted dynamic analysis of Android Applications. AINA'15 Proceedings of IEEE 26th international Conference on Advanced Information Networking and Applications, Gwangui, South Korea, March 24-27, 2015

[27]. Ge X., Taneja K., Xie T., Tillmann N. DyTa: dynamic symbolic execution guided with static verification results. ICSE'11 Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, Mau 21-28, 2011, pp. 992-994

[28]. Герасимов А.Ю., Круглов Л.В. Вычисление входных данных для достижения определенной функции в программе методом итеративного динамического анализа. Труды ИСП РАН, том 28, выпуск 5, 2016, стр. 159-174. DOI: 10.15514/ISPRAS-2016-28(5)-10

[29]. Stallman R. M. Using the GNU Compiler Collection: a GNU Manual for GCC Version 4.3.3. Free Software Foundation Inc., May, 2004

[30]. Исаев И.К., Сидоров Д.В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование, 2010, №4, стр. 1-16

[31]. GNU binutils [HTML] http://www.gnu.org/software/binutils, accessed 01.11.2017

An approach of reachability determination for static analysis defects with help of dynamic symbolic execution

A.Y. Gerasimov <agerasimov@ispras.ru>
L.V. Kruglov <kruglov@ispras.ru>
M.K. Ermakov <mermakov@ispras.ru>
S.P. Vartanov <svartanov@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Historically program analysis methods are divided into two groups – static program analysis methods and dynamic program analysis methods. In this paper, we present a combined approach which allows to determine reachability for defects found by static program analysis techniques through applying dynamic symbolic execution for a program. This approach is an extension of our previously proposed approach for determining the reachability of specific program instructions using dynamic symbolic execution. We focus on several points in the program which include a defect initialisation point, a defect realisation point, and additional intermediate conditional jumps related to the defect in question. Our approach can be described as follows. First of all, we perform static analysis of program executable code to gather information on execution paths which guide dynamic symbolic execution to the point of defect initialisation. Next, we perform concolic execution in order to obtain an input data set to reach the defect initialisation point as well as the defect realisation point through intermediate conditional jumps. Concolic execution is guided by minimizing the distance from a previous path to the next defect trace point when selecting execution paths. The distance metric is calculated using an extended graph of the program combining its call graph and portions of its control flow graph that include all the paths through which the defect realisation point can be reached. We have evaluated our approach using several open source command line programs from Linux Debian. The evaluation confirms that the proposed approach can be used for classification of defects found by static program analysis. However, we have found some limitations, which prevent deploying this approach to industrial program analysis tools. Mitigation of these limitations serves as one of the possible directions for future research.

Keywords: static program analysis; dynamic program analysis.

DOI: 10.15514/ISPRAS-2017-29(5)-7

For citation: Gerasimov A.Y., Kruglov L.V., Ermakov M.K., Vartanov S.P. An approach of reachability confirmation for static analysis defects with help of dynamic symbolic execution. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 5, 2017. pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-7

References

[1]. Vogelsang A., Fehnker A., Huuck R., Reif W. Software Metrics in Static Program Analysis. ICFEM'10 Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering, Shanghai, Shina, November 17-19, 2010, pp. 485-500

[2]. Kim Y., Kim Y., Kim T., Lee G., Jang Y., Kim M. Automated unit testing of large industrial embedded software using concolic testing. ACE'13 Proceedings of the 28yh IEEE/ACM International Conference on Automated Software Engineering, Silicaon Valley, CA, USA, November 11-15, 2013, pp. 519-528

[3]. Xie Y., Chou A., Engler D. ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors. ESEC/FSE-11 Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Helsinki, Finland, September 01-05, 2003, pp. 327-336

[4]. Bessey A., Block K., Chelf B, Chow A., Fulton B., Hallem S., Henri-Gros C., Kamsky A., McPeak S., Engler D. A few billion lines of code later: using static analysis to find bugs in the real world. Communications od ACM, vol. 53, issue 2, February 2010, pp. 66-75

[5]. Ivannikov V.P., Belevantsev A.A., Borodin A.E., Ignat'ev V.N., Zhurikhin D.M., Avetisyan A.I., Leonov M.I. 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. DOI: 10.15514/ISPRAS-2014-26(1)-7

[6]. Engler D., Chelf B., Chou A., Hallen S. Checking system rules using system-specific, programmer-written compiler extensions. OSDI'00 Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, vol. 4, article No.1, San-Diego, CA, USA, October 22-25, 2000

[7]. Johnson B., Song Y., Murphy-Hill E., Bowdidge R., Why don't software developers use static analysis tools to find bugs? ICSE'13 Proceedings of the 2013 International conference on Software Engineering. San Francisco, CA, USA, May 18-26, 2013

[8]. Christakis M., Müller P., Wüstholz An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzers. VMCAI'15 International Workshop on Verification, Model Checking and Abstract Interpretation, Springer, 2015, pp. 336-354

[9]. Livshits B., Sridharan M., Smaragdakis Y., Lhoták O., Amaral J.N., Chang B.-Y. E., Guyer S.Z., Khedker U.P., Møhler A., Vardoulakis D. In defence of soundness: a manifesto. Communications of ACM, vol. 58, no. 2, February 2015

[10]. Cadar C., Dunbar D., Endger D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems. OSDI'08 Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation, San Diego, CA, USA, December 08-10, 2008, pp. 209-224

[11]. Averginos T., Cha S.K., Revert A., Schwartz E.J., Woo M., Brumley D. Automatic Exploit Generation. Communications of the ACM, volume 57, issue 2, February 2014, pp. 74-84

[12]. Chipunov V., Kuznetsov V., Candea G. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems (TOCS) – Special Issue APLOS 2011, article no. 2, volume 30, issue 1, February 2012

[13]. Manevich R., Sridharan M., Adams S., Das M., Yang Z. PSE: explaining program failures via post-mortem static analysis. SIGSOFT'04/FSE-12 Proceedings of the 12th

ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering, Newport Beach, NY, USA, 2004, pp. 63-72

[14]. Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M.G., Liang Z., Newsome J., Poosankam P., Saxena P. BitBlaze: a New Approach to Computer Security via Binary Analysis. ICISS'08 Proceedings of the 4th international Conference on Information Systems Security, Hydarabad, India, December 16-20, 2008, pp. 1-25

[15]. Sen K., Marinov D., Agha G. CUTE: a concolic unit testing engine for C. ESEC/FSE-13 Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, September 05-09, 2005, pp. 263-272

[16]. King J.C. Symbolic Execution and Program Testing. Communications of the ACM, volume 19, issue 7, 1976, pp. 385-394

[17]. Cadar C., Ganesh V., Pawlowski P., Dill D.L., Engler D.R. EXE: automatically generating inputs of death. CCS'06 Proceedings of the 13th ACM Conference on Computer and Communications Security, Alexandria, Virginia, USA, October 30 - November 03, 2006, pp. 322-335

[18]. Schwartz E.J., Averginos T., Brumley D. All You Ever Wanted to Know About Dynamic Tait Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). SP'10 Proceedings of the 2010 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 16-19, 2010, pp. 317-331

[19]. Csallner C., Smaragdakis Y. Check'N'Crash: combining static checking and testing. ICSE'05 Proceedings of the 27th international conference on software engineering, St. Louis, MO, USA, May 15-21, 2005, pp. 422-431

[20]. Chebaro O., Kosmatov N., Giorgetti A., Julliand J. Programs slicing enhances a verification technique combining static and dynamic analysis. SAC'12 Proceedings of the 27th Annual ACM Symposium of Applied Computing, Trento, Italy, March 26-30, 2012, pp. 1284-1291

[21]. Kim T., Park J., Kulinda I., Jang Y. Concolic Testing Framework for Industrial Embedded Software. APSEC'14 Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference, volume 2, Jeju, South Korea, December 01-04, 2014, pp. 7-10

[22]. Hanna A., Ling H.Z., Yang X., Debbabi M. A synergy between static and dynamic analysis or the detection of software security vulnerabilities. OTM'09 Proceedings of the Confederated International Congress, CoopIS, DOA, IS and ADBASE 2009 on the Move to Meaningful Internet Systems: part II. Vilamoura, Portugal, November 01-06, 2009, pp. 815-832

[23]. Csallner C, Smaragdakis Y. DSD-Crasher: a hybrid analysis tool for bug finding. IISTA'06 Proceedings of the 2006 International Symposium on Software Testing and Analysis. Portland, Maine, USA, July 17-20, 2006, pp. 245-254

[24]. Artho C, BIere A. Combined Static and Dynamic Analysis. Electronic Notes in Theoretical Computer Science (ENTCS), volume 131, May, 2005, pp. 3-14

[25]. Chebaro o., Kostomarov N., Giorgetti A., Julliand J. Combining static analysis and test generation for C program debugging. TAP'10 Proceedings of the 4th International Conference on Tests and Proofs. Málaga, Spain, July 01-02, 2010, pp. 94-100

[26]. Schütte J., Fedler R., Tetze D. ConDroid: targeted dynamic analysis of Android Applications. AINA'15 Proceedings of IEEE 26th international Conference on Advanced Information Networking and Applications, Gwangui, South Korea, March 24-27, 2015

[27]. Ge X., Taneja K., Xie T., Tillmann N. DyTa: dynamic symbolic execution guided with static verification results. ICSE'11 Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, Mau 21-28, 2011, pp. 992-994

[28]. Gerasimov A.Y., Kruglov L.V. Input data generation for reaching specific function in program by iterative dynamic analysis method. Trudy ISP RAN/Proc. ISP RAS, vol, 28, issue 5, 2016, pp. 159-174. DOI: 10.15514/ISPRAS-2016-28(5)-10

[29]. Stallman R. M. Using the GNU Compiler Collection: a GNU Manual for GCC Version 4.3.3. Free Software Foundation Inc., May, 2004

[30]. Isaev I.K., Sidorov D.V. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. Programming and Computer Software, 2010, vol. 36, No. 4, pp. 225-236. DOI: 10.1134/S0361768810040055

[31]. GNU binutils [HTML] http://www.gnu.org/software/binutils, accessed 01.11.2017

Логика первого порядка для задания требований к безопасному программному коду

А.В. Козачок <a.kozachok@academ.msk.rsnet.ru>
Академия Федеральной службы охраны РФ,
302034, Россия, г. Орёл, ул. Приборостроительная, д. 35

Аннотация. В настоящее время вопросу защиты информации при проектировании и эксплуатации объектов критической информационной инфраструктуры уделяется особое внимание. Одним из распространенных подходов к обеспечению безопасности информации, обрабатываемой на объектах, при этом является создание изолированной программной среды. Безопасность среды обуславливается ее неизменностью. Однако эволюционное развитие систем обработки информации порождает необходимость запуска в данной среде новых компонентов и программного обеспечения при условии выполнения требований по безопасности. Наиболее важным при этом является вопрос доверия к новому программному коду. Данная работа посвящена разработке формального логического языка описания функциональных требований к программному коду, который позволит в дальнейшем предъявлять требования на этапе статического анализа и контролировать их выполнение в динамике.

Ключевые слова: формальный логический язык; моделирование; процесс; вредоносная программа; проверка моделей; автомат безопасности

DOI: 10.15514/ISPRAS-2017-29(5)-8

Для цитирования: Козачок А.В. Логика первого порядка для задания требований к безопасному программному коду. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 135-148. DOI: 10.15514/ISPRAS-2017-29(5)-8

1. Введение

В последние несколько лет особое внимание в исследованиях, посвященных защите информации, уделяется вопросу, касающемуся обеспечения информационной безопасности объектов критической информационной инфраструктуры (КИИ). Подтверждением данного факта является разработанный и внесенный на рассмотрение федеральный закон "О безопасности критической информационной инфраструктуры Российской Федерации" 6 декабря 2016 года. Особое внимание при этом уделяется защите объектов КИИ от компьютерных атак и деструктивных программных

воздействий [1]. Предметом рассмотрения данной статьи является защита объектов КИИ от деструктивных программных воздействий.

Зачастую возможность реализации данных угроз обуславливается наличием доступа в глобальную сеть Интернет объектов КИИ. При этом существующие системы и средства защиты информации не обеспечивают гарантированной защиты. Так, например, исследование, проведенное компанией AV-Comparatives, показало, что применяемые в современных антивирусных средствах механизмы обнаружения позволяют достичь уровня эвристического обнаружения 0,974 ("Avast Internet Security"), но при этом остаются необнаруженными 468 образцов вредоносного программного обеспечения [2].

Одним из возможных подходов к защите от деструктивных программных воздействий является использование изолированной программной среды, которая является доверенной и безопасной при условии сохранения ее неизменности. Однако эволюционное развитие систем сбора и обработки информации, а также наличие доступа объектов КИИ в глобальную сеть Интернет порождает необходимость запуска в данной среде новых компонентов и программного обеспечения, которые могут нарушить ее целостность и безопасность. Наиболее важным при этом является вопрос доверия к новому контенту и программному коду.

Одним из возможных вариантов построения безопасной среды с возможностью доверия к приходящему контенту является применение системы безопасного исполнения программного кода [3]. Предлагаемая система является расширенной композицией двух активно развивающихся подходов к обнаружению деструктивных программных воздействий, а именно: применения методов формальной верификации модели исследуемой программы [4–7] и использование автомата безопасности для контроля за функционированием исследуемой программы в реальном времени [8–12].

В основе системы безопасного исполнения программного кода лежит предположение о том, что если безопасность программного кода при априорно известных функциональных требованиях не подтверждена, то ее использование запрещается. Настоящее исследование посвящено разработке формального языка описания функциональных требований к программному коду для применения его в разрабатываемой системе безопасного исполнения программного кода.

2. Краткий обзор исследований в области формальной верификации моделей программ для обнаружения вредоносного кода

Идея применения метода формальной верификации "Model checking" при решении задачи обнаружения деструктивных программных воздействий состоит в том, чтобы построить формальную (математическую) модель вредоносной программы, которая отражает (моделирует) ее возможное

поведение в операционной системе. Разрешенное поведение программы при этом задается в виде спецификации. На основе этой спецификации и модели исполняемого файла с использованием метода "Model checking" принимается решение о возможности использования анализируемой программы.

Впервые данный метод при решении задачи по обнаружению вредоносного кода был применен в 2005 году в работе Киндера [4]. Группа авторов предложила анализировать поведение программ и на основе "Model checking" принимать решение о его схожести с поведением вредоносных программ. Предложенный ими подход заключается в задании спецификаций с помощью формул темпоральной логики для каждого из классов деструктивных программ, представители которого имели схожее поведение, однако существенно отличались по бинарному представлению, вследствие чего не могли быть обнаружены сигнатурным методом. Каждый исследуемый бинарный файл автоматически преобразовывался в модель, заданную на языке верификатора. На основе этой модели верификатор определял соответствует ли она одной из множества заданных спецификаций, соответствующих семействам вредоносных программ. Для сокращения записи спецификаций авторами была введена логика CTPL (Computation Tree Predicate Logic), являющаяся расширением известной логики CTL. Достоинством предложенного подхода является возможность обнаружения семейств вредоносных программ. Ограничением является то, что он не учитывает работу исследуемой программы со стеком, а также необходимость ручного задания спецификации поведения для каждого класса вредоносного кода.

В 2012 году Фу Сонг и Тассир Туили предложили использовать метод "Model checking" для обнаружения вредоносных программ с учетом их поведения и взаимодействия со стеком [5]. Для описания модели поведения вредоносного кода авторами была введена логика SCTPL (Stack Computation Tree Predicate Logic), позволяющая учитывать работу со стеком. Применение данного подхода позволило существенно повысить точность обнаружения вредоносных программ. В результате развития данного подхода авторами была предложена логика SLTPL (Stack Linear Tree Predicate Logic) [6].

3. Описание системы безопасного исполнения программного кода

Отличительная особенность разрабатываемой системы заключается в извлечении информации о поведении исследуемой программы, как на этапе хранения, так и на этапе исполнения (рис. 1). Полученная информация сравнивается со спецификацией поведения в соответствии с заданными функциональными требованиями. В случае их соответствия исследуемая программа признается безопасной.

Для реализации предложенного подхода на практике необходимо решение следующих частных задач:

- анализ (извлечение и преобразование информации из исследуемой программы в вид, пригодный для дальнейшей обработки – спецификации);
- синтез (построение модели безопасного исполнения программного кода в соответствии с заданными функциональными требованиями);
- верификация (алгоритм, получающий на вход спецификацию анализируемой программы и принимающий решение о ее соответствии модели безопасного исполнения программного кода);
- контроль исполнения (реализация монитора исполнения программы, позволяющего перехватывать все сигналы взаимодействия процесса с операционной системой, отслеживать соответствие состояния программы заданной конфигурации и завершить целевую программу в случае необходимости).

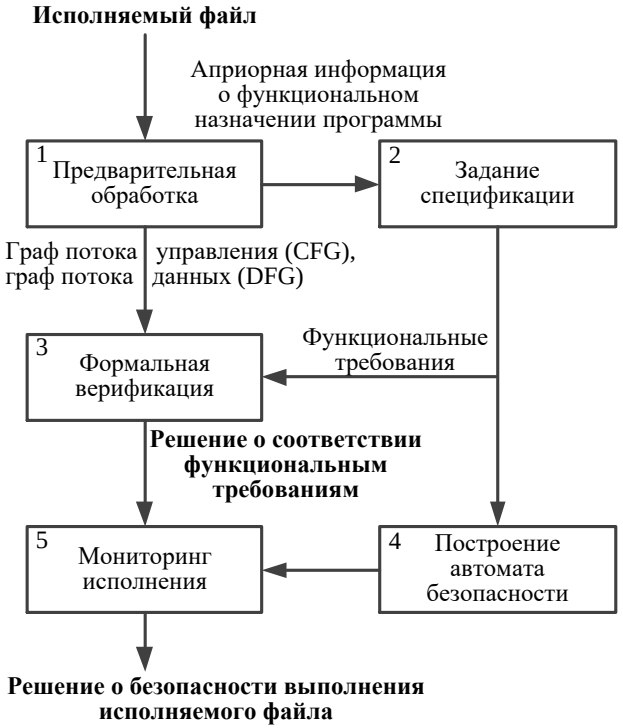


Рис. 1. Модель системы безопасного исполнения программного кода
Fig. 1. Secure code execution system model

На вход блока 1 подается исследуемый исполняемый файл, безопасность которого необходимо установить. В данном блоке происходит проверка наличия в коде программы конструкций самомодификации (включая упаковщики) и механизмов защиты кода от анализа. Выполнение этих требований является необходимым условием для дальнейшего исследования исполняемого файла. В случае их наличия файл признается небезопасным и его дальнейшая проверка прекращается. Далее происходит преобразование исполняемого файла из бинарного представления в последовательность команд языка ассемблера и данных, на их основе строится граф потока управления (Control Flow Graph) и граф потока данных (Data Flow Graph). Также на этом этапе осуществляется сбор и систематизация априорных сведений о функциональном предназначении программы, которые подаются на вход блока 2.

В блоке "Задания спецификации" на основе априорных сведений о функциональном предназначении программы формируется перечень ограничений на ее функциональные возможности, выполнение которых необходимо для безопасного использования программы. Данный перечень включает в себя функциональные требования, выполнение которых может быть обеспечено в рамках работы системы безопасного исполнения программного кода. Они могут быть разделены на группы в зависимости от категории исследуемой программы. Выходом данного блока являются формализованные функциональные ограничения на работу программы в виде формулы темпоральной логики и конфигурации автомата безопасности.

В блоке 3 осуществляется процесс формальной верификации модели исполняемого файла, построенной на основе графов потоков управления и данных, на соответствие функциональным требованиям статического этапа проверки с помощью метода "Model checking". Требования по корректности безопасного поведения при этом описываются в виде спецификации, которая отражает рамки разрешенного поведения программы. Поскольку происходит именно математическая верификация, решение о согласованности, то есть соответствии возможного поведения требуемому, является корректным. Алгоритмы проверки моделей, как правило, базируются на исчерпывающей достижимости всего множества состояний модели [13].

Таким образом, для каждого состояния проверяется, удовлетворяет ли оно заданным в спецификации требованиям. В самой простой форме алгоритмы проверки моделей позволяют дать ответ на вопрос о достижимости заданных состояний. В таком случае необходимо определить все запрещенные состояния, достижение которых является небезопасным, и выяснить, существует ли такая последовательность их смены, которая приводит к одному из запрещенных. Если такая последовательность существует, то принимается решение о запрете использования исследуемой программы. Стоит отметить, что исчерпывающая достижимость множества состояний гарантируется ввиду конечности числа состояний модели [14]. В случае

рассогласованности модели программы и спецификации безопасности исполняемый файл признается небезопасным и его дальнейшее исследование прекращается. Выходом данного блока является решение о безопасности использования данной программы по результатам верификации на статическом этапе проверки.

Для контроля над выполнением функциональных ограничений во время работы программы предлагается использование системы, схожей с системой предотвращения вторжений на уровне компьютера. Монитор исполнения запускается параллельно с исполняемой программой и перехватывает все, совершаемые ею, системные вызовы. Изначально монитор исполнения загружает модель разрешенного поведения и устанавливает начальное состояние. Каждый вызов системной функции сопоставляется с моделью разрешенного поведения и происходит переход в новое состояние или, в случае отсутствия такого перехода, подается команда на завершение процесса. В основу монитора исполнения положен автомат безопасности (Security Automata) [8], который строится, как правило, на основе автомата со стеком. Входными символами автомата является множество событий функционирования процесса, а конфигурация автомата определяет множество разрешенных операций для каждого состояния.

В блоке 4 происходит преобразование функциональных требований к программе в конфигурацию конечного автомата со стеком. В блоке 5 производится непрерывный мониторинг функционирования программы в рамках заданной модели безопасного исполнения. Выходом данного блока является решение о безопасности выполнения исполняемого файла.

4. Формальный логический язык описания функциональных требований

Базовыми понятиями при рассмотрении работы операционной системы (ОС) являются процесс и ресурс. Согласно [15] процесс – это контейнер для набора ресурсов, используемых при выполнении экземпляра программы. К основным видам ресурсов ОС относятся следующие элементы [16]:

- процессорное время;
- оперативная память;
- внешняя память;
- устройства ввода-вывода.

В основе построения предлагаемой системы безопасного исполнения программного кода лежит модель описания функционирования процесса в операционной системе. К субъектам относятся процессы, совершающие действия по отношению к объектам. Объектами являются ресурсы ОС и процессы, в отношении которых совершаются действия другими субъектами:

- "процесс" (*p*);

- "оперативная память" (*m*);
- "внешняя память" (*e*);
- "периферийные устройства" (*d*);
- "сетевая подсистема" (*n*).

Для осуществления доступа к ресурсу процесс выполняет соответствующую функцию ОС, то есть подает заявку на выполнение некоторых действий. ОС на основе внутренних механизмов распределения ресурсов, а также на основе политики безопасности, принимает решение о доступе исполняемого кода данного процесса к запрашиваемому ресурсу.

Прикладные программы в процессе работы обладают полным доступом к своему виртуальному адресному пространству для выполнения операций чтения и записи. Для ввода или вывода данных за пределы своего адресного пространства прикладной программе необходимо вызывать соответствующие функции ОС, если она имеет соответствующие привилегии для осуществления таких операций. К таким функциям ОС можно отнести операции чтения, записи, запуска или завершения процесса, выделения дополнительной области памяти, ее освобождения и др.

Анализ исследований в области формальной верификации показал, что существующие подходы к описанию спецификаций при решении задачи обнаружения вредоносных программ не являются универсальными. Так как часть из них ориентирована на команды языка ассемблера, а часть – на API-функции.

Поэтому предлагается следующий формальный логический язык для задания функциональных требований с возможностью однозначного перехода к формулам темпоральной логики для дальнейшей верификации по моделям.

В соответствии с определением логики первого порядка [17] необходимо задать следующие подмножества:

$$FormSpec = Func \cup \Pr ed \cup Var \cup Log \cup Aux.$$

При этом множество функциональных символов будет включать в себя следующие операции:

$$Func = \{create, open, delete, read, write\},$$

где *create* – операция создания объекта, *open* – операция открытия объекта, *delete* – операция удаления (завершения) объекта, *read* – операция чтения из объекта, *write* – операция записи в объект.

Множество предикатных символов включает в себя базовые предикаты темпоральной логики CTL* [18] и предикат проверки безопасности:

$$\Pr ed = \{IsSecure, AX, AF, AG, AU,$$
$$AR, EX, EF, EG, EU, ER, EC\},$$

где *IsSecure* – предикат проверки безопасности текущего состояния относительно трассы исполнения в целом, A – квантор всеобщности,

указывающий на то, что данное свойство выполнено для всех путей, *E* – квантор существования, указывающий на то, что данное свойство выполняется для некоторого пути, *X* – унарный оператор, указывающий на то, что данное свойство выполняется на следующем состоянии текущего пути, *G* – унарный оператор, указывающий на то, что данное свойство выполняется на каждом состоянии текущего пути, *F* – унарный оператор, указывающий на то, что данное свойство выполняется в некотором состоянии в будущем, *U* – бинарный оператор, указывающий на то, что первое свойство выполняется для всех состояний пути, предшествующих состоянию, где выполняется второе свойство, *R* – бинарный оператор, указывающий на то, что второе свойство выполняется для всех состояний, следующих до состояния, в котором выполняется первое свойство, *C* – унарный оператор, указывающий на то, что данное свойство выполняется в текущем состоянии текущего пути (дополнительно введен авторами).

Множество символов предметных переменных включается в себя следующие элементы:

$$Var = \{p, m, n, e, d, cat\},$$

где *p,m,n,e,d* – объекты над которыми совершаются действия, *cat* – индекс категории объекта (субъекта) (табл. 1).

Табл. 1. Категории объектов и субъектов.
Table 1. Categories of subjects and objects.

Обозначение	Описание
Субъект "Процесс" (*p*)	
1	системный процесс
2	привилегированный процесс
3	пользовательский процесс
Объект "Оперативная память" (*m*)	
1	адресное пространство системного процесса
2	адресное пространство другого процесса
3	собственное адресное пространство процесса
Объект "Внешняя память" (*e*)	
1	исполняемые файлы
2	системные каталоги и конфигурация системы
3	файлы и каталоги других пользователей
4	системные библиотеки
5	собственные файлы и каталоги
Объект «Периферийные устройства» (d)	
1	устройства вывода
2	устройства ввода
Объект "Сетевая подсистема" (*n*)	
1	сервисы узлов глобальной сети
2	сервисы узлов локальной сети

3	локальные сетевые сервисы

Множество логических символов включает в себя следующие элементы:

$$Log = \{\neg, \wedge, \vee, \rightarrow, \exists, \forall\},$$

где $\neg$ – символ логического отрицания, $\wedge$ – символ конъюнкции, $\vee$ – символ дизъюнкции, $\rightarrow$ – символ импликации, $\exists$ – квантор существования, $\forall$ – квантор всеобщности.

Множество вспомогательных символов включает в себя следующие элементы:

$$Aux = \{, ()\}.$$

5. Базис функциональных требований, обеспечивающих безопасное исполнение программного кода

На основе предложенного формального логического языка *FormSpec* были сформулированы базовые правила (формулы) безопасного исполнения программного кода для каждого из функциональных символов. Символ * означает объект (субъект) любой категории из числа возможных для данного класса.

Для операции создания объекта:

- $\neg$ *EF create (p,*,p,*)* – запрет на создание дочерних процессов;
- *EF create (p,*,m,3)* – выделение памяти только в своем адресном пространстве процесса;
- *EF create (p,*,e,5)* – разрешено создание новых файлов (каталогов) только в каталоге текущего процесса;
- $\neg$ *EF create (p,*,n,*)* – запрет на создание сетевых соединений;
- $\neg$ *EF create (p,*,d,*)* – запрет на создание устройств (драйверов).

Для операции открытия объекта:

- $\neg$ *EF open (p,*,p,*)* – запрет на открытие процессов;
- *EF open (p,*,e,4)* $\vee$ *EF open (p,*,e,5)* – разрешено открытие системных библиотек и файлов, содержащихся в текущем каталоге процесса;
- $\neg$ *EF open (p,*,d,*)* – запрет на открытие устройств.

Для операции удаления (завершения) объекта:

- *EF delete* (p_i,*,p_i,*) – процесс может завершить свою работу;
- *EC open (p,*,*e_j*,5)* $\wedge$ *EF delete(p,*,*e_j*,5)* – процесс может удалять файлы им созданные.

Для операции чтения из объекта:

- $\neg$ *EF read(p,*,p,*)* – запрет на получение информации о процессах;
- *EF read(p,*,m,3)* – разрешено чтение адресного пространства своего процесса;

- $EC\ open(p,*,e_j,4) \wedge EF\ read(p,*,e_j,4)$ – разрешено чтение из системных библиотек;
- $(EC\ open(p,*,e_j,5) \vee EC\ create(p,*,e_j,5)) \wedge EF\ read(p,*,e_j,5)$ – разрешено чтение файлов открытых (созданных) процессом;
- $\neg EF\ read(p,*,n,*)$ – запрет на работу с сетью;
- $\neg EF\ read(p,*,d,*)$ – запрет на работу с устройствами.

Для операции записи в объект:

- $EF\ write(p,*,m,3)$ – разрешена запись в адресное пространство своего процесса;
- $(EC\ open(p,*,e_j,5) \vee EC\ create(p,*,e_j,5)) \wedge EF\ write(p,*,e_j,5)$ – разрешено чтение файлов открытых (созданных) процессом;
- $\neg EF\ write(p,*,n,*)$ – запрет на работу с сетью;
- $\neg EF\ write(p,*,d,*)$ – запрет на работу с устройствами.

Следует отметить, что представленный базис можно рассматривать как аксиоматический, так как его выполнение обеспечивает безопасность выполнения программного кода (выполнение предиката *IsSecure*) в аспекте защиты от вредоносного кода. Также ограничения, вводимые им на предмет взаимодействия с сетевой и файловой подсистемами, возможно преодолеть за счет введения ограничений на последовательность выполняемых действий и изоляции возможных информационных контуров.

6. Обсуждение результатов

Одним из вариантов практического приложения предложенного формального логического языка описания функциональных требований к программному коду является формализация угроз из "Банка данных угроз безопасности информации" ФСТЭК [19].

"Угроза изменения системных и глобальных переменных" нарушителем может быть реализована за счет применения вредоносного программного обеспечения, которое может привести к совершению опосредованного деструктивного воздействия на определенные программы или систему в целом. Для нейтрализации данной угрозы необходимо задать следующее правило: "Запрет процессам 3 категории на совершение изменений системных и глобальных переменных", выраженное следующем образом:

$$\neg EF\ (create\ (p,3,e,2) \vee write\ (p,3,e,2)) \quad (1)$$

Выражение (1) формально означает, что процессы 3 категории не могут осуществлять создание или модификацию системных каталогов и файлов конфигурации. Этим же правилом можно предотвратить "угрозу несанкционированного редактирования реестра".

Суть "угрозы несанкционированного копирования защищаемой информации" заключается в получении злоумышленником копии защищаемой информации

другого пользователя и дальнейшем выводе ее за пределы системы. Для ограничения последовательности таких действий необходимо ввести следующее правило:

$$\neg EF(EC\ read\ (p,*,e,3) \wedge (EF\ create\ (p,*,e,5) \vee EF\ write\ (p,*,e,5) \vee \vee EF\ write\ (p,*,d,1) \vee EF\ write\ (p,*,n,1)) \quad (2)$$

Из выражения (2) следует, что процессу любой категории запрещается последовательно сначала считывать некоторую информацию из файлов других пользователей, а затем осуществлять ее запись в файлы в собственном каталоге или отправлять на устройства вывода или в сеть.

Для "угрозы перехвата вводимой и выводимой на периферийные устройства информации" можно задать правило, ограничивающее прямое взаимодействие процессов 3 категории и устройств ввода:

$$\neg EF\ read(p,3,d,2) \quad (3)$$

Выражение (3) запрещает непосредственный доступ к считыванию информации с устройств ввода в обход существующих в операционной системе механизмов.

Ряд приведенных примеров подтверждает состоятельность и полноту возможностей описания существующих угроз на предложенном формальном логическом языке. Их учет при работе системы безопасного исполнения кода позволит исключить возможность реализации заданных угроз.

7. Заключение

Предложенный формальный логический язык описания функциональных требований, позволяющий описать поведение процесса без конкретизации операций или элементарных действий (на высоком уровне абстракции), и в обобщенной математической форме выразить субъектно-объектные отношения процессов с ресурсами ОС различных категорий., задает основу построения системы безопасного исполнения программного кода, которая позволит доверять новому программному коду и не нарушать целостность изолированной программной среды.

Направлением дальнейших исследований является построение полного набора правил безопасного исполнения программного кода с использованием введенного формального логического языка, позволяющего устранить ограничения, введенные аксиоматическим базисом.

Список литературы

[1]. Проект федерального закона от 06.12.2016 № 9198-П10 "О безопасности критической информационной инфраструктуры Российской Федерации". URL: http://asozd2.duma.gov.ru/main.nsf/(SpravkaNew)?OpenAgent&RN=47571-7&02 (дата обращения 14.03.2017).

[2]. Козачок А.В., Кочетков Е.В., Татаринов А.М. Обоснование возможности построения эвристического механизма распознавания вредоносных программ на основе статического анализа исполняемых файлов. Вестник компьютерных и информационных технологий, 2017, №. 3, с. 50-56. DOI: 10.14489/vkit.2017.03. pp.050-056.
[3]. Козачок А.В., Кочетков Е.В. Обоснование возможности применения верификации программ для обнаружения вредоносного кода. Вопросы кибербезопасности. 2016. № 3 (16). С. 25-32.
[4]. Kinder J. et al. Detecting malicious code by model checking. International Conference on Detection of Intrusions and Malware and Vulnerability Assessment. Springer Berlin Heidelberg, 2005, pp. 174-187.
[5]. Song F., Touili T. Efficient malware detection using model-checking. International Symposium on Formal Methods. Springer Berlin Heidelberg, 2012, pp. 418-433.
[6]. Song F., Touili T. PoMMaDe: pushdown model-checking for malware detection. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013, pp. 607-610.
[7]. Jasiul B., Szpyrka M., Śliwa J. Formal Specification of Malware Models in the Form of Colored Petri Nets. Computer Science and its Applications. Springer Berlin Heidelberg, 2015, pp. 475-482.
[8]. Schneider F.B. Enforceable security policies. ACM Transactions on Information and System Security (TISSEC), 2000, vol. 3, no. 1, pp. 30-50.
[9]. Feng H.H. et al. Formalizing sensitivity in static analysis for intrusion detection. Proc. IEEE Symposium on Security and Privacy, 2004, pp. 194-208.
[10]. Basin D. et al. Enforceable security policies revisited. ACM Transactions on Information and System Security (TISSEC), 2013, vol. 16, no. 1, pp. 3-8.
[11]. Feng H.H. et al. Anomaly detection using call stack information. Proc. IEEE Symposium on Security and Privacy, 2003, pp. 62-75.
[12]. Basin D., Klaedtke F., Zălinescu E. Algorithms for monitoring real-time properties. International Conference on Runtime Verification. Springer Berlin Heidelberg, 2011, pp. 260-275.
[13]. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. Москва, 2002, изд-во МЦНМО, 416 с.
[14]. Вельдер С.Э., Лукин М.А., Шалыто А.А., Яминов Б.Р. Верификация автоматных программ. Санкт-Петербург, 2011, изд-во Наука, 244 с.
[15]. Руссинович М., Соломон Д. Внутреннее устройство Microsoft Windows. 6-е изд. Санкт-Петербург, 2013, изд-во Питер, 800 с.
[16]. Гордеев А.В. Операционные системы: по направлению подгот. "Информатика и вычисл. техника". Санкт-Петербург, 2009, изд-во Питер, 416 с.
[17]. Коротков М.А., Степанов Е.О Основы формальных логических языков. Санкт-Петербург, 2003, изд-во СПб ГИТМО (ТУ), 84 с.
[18]. Hafer T., Thomas W. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. International Colloquium on Automata, Languages and Programming. Springer Berlin Heidelberg, 1987, pp. 269-279.
[19]. ФСТЭК "Банк данных угроз безопасности информации" URL: http://bdu.fstec.ru/ (дата обращения: 14.03.17).

First order logic to set requirements for secure code execution

A.V. Kozachok < a.kozachok@academ.msk.rsnet.ru >
The Academy of the Federal Guard Service of the Russian Federation,
34, Priborostroitelnaya st., Oryol, 302034, Russia

Abstract. Currently the problem of information security during designing and exploiting the objects of critical information infrastructure is paid special attention to. One of the most common approaches to providing information security, processed on the objects, is creating isolated programming environment. The environment security is determined by its invariability. However, the evolutional development of data processing systems gives rise to the necessity of implementing the new components and software in this environment under condition that security requirements are satisfied. The most important requirement consists in trust in the new programming code. The given paper is devoted to developing formal logical language of description of functional requirements for programming code, allowing to make further demands at the stage of static analysis and to control their implementation in dynamics.

Keywords: formal logical language, modeling, process, malware, model checking, security automata.

DOI: 10.15514/ISPRAS-2017-29(5)-8

For citation: Kozachok A.V. First order logic to set requirements for secure code execution. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 5, 2017. pp. 135-148 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-8

References

[1]. Federal law draft no. 9198-P10 "On the security of the critical information infrastructure of the Russian Federation". URL: http://asozd2.duma.gov.ru/main.nsf/(SpravkaNew)/OpenAgent&RN=47571-7&02 (in Russian).

[2]. Kozachok A.V., Kochetkov E.V, Tatarinov A.M. Construction Heuristic Malware Detection Mechanism Based on Static Executable File Analysis Possibility Proof. Vestnik komp'juternyh i informacionnyh tehnologij [Herald of Computer and Information Technologies], 2017, no. 3, pp. 50-56. DOI: 10.14489/vkit.2017.03. pp. 050-056 (in Russian).

[3]. Kozachok A.V., Kochetkov E.V. Using program verification for detecting malware. Voprosy kiberbezopasnosti [Cybersecurity issues], 2016, no. 3, vol. 16, pp. 25-32 (in Russian).

[4]. Kinder J. et al. Detecting malicious code by model checking. International Conference on Detection of Intrusions and Malware and Vulnerability Assessment. Springer Berlin Heidelberg, 2005, pp. 174-187.

[5]. Song F., Touili T. Efficient malware detection using model-checking. International Symposium on Formal Methods. Springer Berlin Heidelberg, 2012, pp. 418-433.

[6]. Song F., Touili T. PoMMaDe: pushdown model-checking for malware detection. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013, pp. 607-610.
[7]. Jasiul B., Szpyrka M., Śliwa J. Formal Specification of Malware Models in the Form of Colored Petri Nets. Computer Science and its Applications. Springer Berlin Heidelberg, 2015, pp. 475-482.
[8]. Schneider F.B. Enforceable security policies. ACM Transactions on Information and System Security (TISSEC), 2000, vol. 3, no. 1, pp. 30-50.
[9]. Feng H.H. et al. Formalizing sensitivity in static analysis for intrusion detection. Proc. IEEE Symposium on Security and Privacy, 2004, pp. 194-208.
[10]. Basin D. et al. Enforceable security policies revisited. ACM Transactions on Information and System Security (TISSEC), 2013, vol. 16, no. 1, pp. 3-8.
[11]. Feng H.H. et al. Anomaly detection using call stack information. Proc. IEEE Symposium on Security and Privacy, 2003, pp. 62-75.
[12]. Basin D., Klaedtke F., Zălinescu E. Algorithms for monitoring real-time properties. International Conference on Runtime Verification. Springer Berlin Heidelberg, 2011, pp. 260-275.
[13]. Klark Je., Gramberg O., Peled D. Program models verification: Model Checking. Moscow, MCNMO, 2002, 416 p (in Russian).
[14]. Vel'der S.Je., Lukin M.A., Shalyto A.A., Jaminov B.R. Automata program verififcation. St. Petersburg, Nauka, 2011, 244 p (in Russian).
[15]. Russinovich M. E., Solomon D. A., Ionescu A. Windows internals. Pearson Education, 2012, 800 p.
[16]. Gordeev A.V. Operating systems. Izdatel'skij dom “Piter”, 2009, 412 p (in Russian).
[17]. Korotkov M.A., Stepanov E.O Fundamentals of formal logical languages. St. Petersburg, SPb GITMO (TU), 2003, 84 p (in Russian).
[18]. Hafer T., Thomas W. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. International Colloquium on Automata, Languages and Programming. Springer Berlin Heidelberg, 1987, pp. 269-279.
[19]. Federal Service for Technical and Export Control. Information security threats database. URL: http://bdu.fstec.ru (in Russian).

Обещающая компиляция в ARMv8.3

[1,2]*А.В.Подкопаев <apodkopaev@2009.spbu.ru>*
[3] *О.Лахав <orilahav@tau.ac.il>*
[4]*В.Вафеядис <viktor@mpi-sws.org>*
[1] *Санкт-Петербургский Государственный Университет,*
199034, Россия, Санкт-Петербург, Университетская набережная, д. 7–9
[2]*JetBrains Research, 199034, Россия, Санкт-Петербург,*
Университетская набережная, д. 7–9–11/5А
[3] *Тель-Авивский Университет, 39040, Израиль, Тель-Авив 69978*
[4]*Институт им. Макса Планка: Программные Системы,*
67663, Германия, Кайзерслаутерн, ул. Пауль-Эрлих G26

Аннотация. Поведение многопоточных программ не может быть промоделировано попеременным последовательным исполнением различных потоков на одном вычислительном узле. На данный момент полное и корректное описание поведения многопоточных программ является открытой теоретической проблемой. Одним из перспективных решений этой проблемы является „обещающая“ модель памяти. Для того, чтобы некоторая модель могла быть использована в стандарте некоторого промышленного языка программирования, должна быть доказана корректность компиляции из этой модели в модель памяти целевой процессорной архитектуры. Данная статья представляет доказательство корректности компиляции из подмножества обещающей модели в модели памяти процессора ARMv8.3. Главной идеей доказательства является введение промежуточного операционного варианта модели ARMv8.3, поведение которого может быть симулировано обещающей моделью.

Ключевые слова: многопоточность; корректность компиляции; слабые модели памяти.

DOI: 10.15514/ISPRAS-2017-29(5)-9

Для цитирования: Подкопаев А.В., Лахав О., Вафеядис В. Обещающая компиляция в ARMv8.3. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 149-164. DOI: 10.15514/ISPRAS-2017-29(5)-9

1. Введение

Распространенным заблуждением является уверенность, что поведение многопоточных программ может быть описано попеременным исполнением инструкций потоков одним процессором, то есть в рамках модели *последовательной консистентности* [1]. В качестве контрпримера рассмотрим следующую программу:

a := [*x*];	‖	b := [*y*];	**(LB)**
[*y*] := 1;	‖	[*x*] := 1;	

В этой программе каждый поток сначала выполняет чтение из некоторой локации (*x* или *y*), а потом записывает значение 1 в другую локацию. Пусть изначально в локации *x* и *y* записан 0. Тогда попеременным исполнением инструкций потоков можно получить три следующих результата: 1) a = 0, b = 0; 2) a = 0, b = 1; 3) a = 1, b = 0. Тем не менее, данная программа, будучи запущенной на процессорах семейств Power или ARM, может завершиться с результатом a = 1, b = 1. Такой результат не может быть получен в модели последовательной консистентности, так как требует выполнения обеих операций записи до любого чтения.

Поведение многопоточной программы, выходящее за пределы последовательной консистентности, называют *слабым поведением* (weak behavior). Слабое поведение является результатом применения оптимизаций со стороны компилятора и/или процессора. Так в случае программы **LB**, компилятор может переставить инструкции первого потока, например, как часть планирования кода [2], или процессор может буферизировать чтение во втором потоке и выполнить инструкцию [*x*] := 1 до завершения исполнения инструкции чтения. В обоих случаях результат a = 1, b = 1 станет возможным.

В современных системах используют оптимизации, которые таким странным образом меняют семантику многопоточной программы, потому что, во-первых, они ускоряют выполнение программ, а, во-вторых, для большинства многопоточных систем справедливо, что слабое поведение не проявляется у программ с корректными блокировками, т. е. у программ без гонок по памяти[1].

В связи с тем, что слабое поведение программ бывает очень сложным, и требуется гарантировать надёжность прикладному программисту, научное сообщество в сотрудничестве с индустрией разрабатывает формальные семантики для систем со слабыми поведениями, или *слабые модели памяти* (weak memory models). Слабые модели существуют для таких распространенных процессорных платформ, как x86 [3], Power [4,5], ARM [6,7], а также для языков программирования C/C++ [8] и Java [9].

Отметим, что, с одной стороны, прикладному программисту хотелось бы иметь как можно больше гарантий, что означает уменьшение числа возможных слабых поведений. С другой стороны, модель памяти языка должна предоставлять возможность эффективной компиляции в целевые платформы, что ведёт к увеличению числа слабых поведений.

На данный момент не существует стандартизованной модели памяти промышленного языка программирования, которая удовлетворяет всем

1 В программе имеется гонка по памяти (data race), если в ней есть два конкурентных обращения к одной и той же ячейке памяти, при этом как минимум одно из обращений является записью в ячейку [10].

требованиям. Так известно, что модель памяти Java не допускает некоторые оптимизации, которые используются даже в официальном компиляторе Java [11], а модель памяти C/C++11 допускает т. н. значения из воздуха (out-of-thin-air values) [12]. Тем не менее, недавно было представлено перспективное решение для данной проблемы – «обещающая» модель памяти [13].

Для того, чтобы обещающая модель памяти смогла войти в стандарт промышленного языка программирования, про модель должно быть доказано несколько утверждений, в том числе возможность её эффективной и корректной компиляции. Данная статья посвящена корректности компиляции из обещающей модели памяти в аксиоматическую модель памяти процессора ARMv8.3 [7].

Основным результатом работы является доказательство корректности компиляции из подмножества обещающей модели памяти без сертификации [13] в модель ARMv8.3. Рассмотренное подмножество обещающей модели состоит из расслабленных (relaxed) операций чтения и записи, а также высвобождающих (release) и приобретающих (acquire) барьеров памяти. Кроме того, в доказательстве используется новый подход, который потенциально может быть использован для других подобных доказательств: вводится промежуточная операционная семантика, которая осуществляет обход исполнения аксиоматической семантики (в нашем случае, ARMv8.3), при этом данный обход может быть симулирован обещающей моделью. В отличии от метода, использованного ранее в [13], наш подход вводит меньше ограничений на целевую платформу и не опирается на представимость целевой модели как набора оптимизаций над более строгой моделью [14].

Статья организована следующим образом. Мы вводим понятие корректности компиляции и описываем существующие работы в соответствующей области (раздел 2), обсуждаем базовые концепции модели памяти ARMv8.3 и обещающей модели на примерах (разделы 3 и 4), описываем общую структуру доказательства (раздел 5). В разделе 6 подводятся итоги и приводятся возможные направления для дальнейшей работы.

Отметим, что в данной статье приводится лишь верхнеуровневое описание доказательства. Подробное формальное доказательство приведено в [15].

2. Корректность компиляции. Существующие работы

Выберем два языка – HL (high level) и LL (low level) – а также схему компиляции *comp* из HL в LL, т. е. функцию, преобразующую программу на языке HL в язык LL. Схема компиляции *comp* считается корректной, если для любой программы *Prog* и её скомпилированной версии *comp(Prog)* выполняется следующее: любое поведение, разрешенное для *comp(Prog)* в рамках модели языка LL, также разрешено для *Prog* в рамках модели HL.

На данный момент существует множество работ, посвящённых корректности компиляции. Наиболее известными являются [16] и [17], представляющие

верифицированные компиляторы для языков C и CakeML соответственно. Данные компиляторы сравнимы с промышленными по качеству генерируемого кода. Существенным ограничением данных работ является то, что они рассматривают только однопоточные программы. Существует и расширение работы [16] для случая слабой памяти [18], но в [18] рассматривается только достаточно простая и ограниченная модель памяти TSO (total store order).

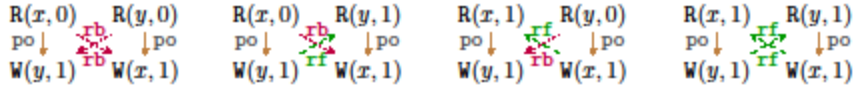
Имеются работы, посвящённые компиляции из одной слабой модели памяти в другую [8, 13, 19, 20, 21]. В них отсутствует полноценная компиляция промышленного языка программирования и рассматривается лишь подмножество языка, имеющее отношение к многопоточности. Как следствие, исходный и целевой языки обычно считаются совпадающими с точностью до модификаторов на операциях записи, чтения и барьеров памяти, которые имеют отношение к многопоточности. Этим же путём следует и представляемая работа.

Наиболее близкими к нашей работе являются [13] и [21]. В [13] доказывается корректность компиляции из обещающей модели в модели процессоров x86-TSO [3] и Power [4]. При этом авторы [13] использовали результат [14], который утверждает, что все исполнения, возможные в рамках моделей x86-TSO [3] и Power [4], могут быть представлены как некоторое число оптимизаций над исполнением в более строгих моделях, которые не нуждаются в механизме обещаний. Далее авторы доказывают, что упомянутые оптимизации являются корректными в рамках обещающей модели, т. е. семантика оптимизированной программы является подмножеством семантики изначальной программы, и показывают корректность компиляции для упрощенных моделей. К сожалению, модель памяти ARMv8.3 [7] не может быть представлена как набор упомянутых оптимизаций над более строгой моделью памяти. Поэтому в нашей работе нам пришлось использовать другой подход.

В работе [21] показана корректность компиляции из обещающей модели в модель памяти ARMv8 POP [6]. В работе использован метод *запаздывающей симуляции*, который является специальной формой индукции по исполнению программы в целевой машине. Данный подход возможен, т. к. обе модели представлены *операционно*, т. е. в терминах некоторой абстрактной машины. Поскольку модель памяти процессора ARMv8.3 задана *аксиоматически*, т.е. она представляет семантику программы как множество графов, где каждый конкретный граф является одним из возможных исполнений программы, то доказательство с помощью симуляции напрямую не построить. Чтобы преодолеть данное ограничение и все-таки использовать симуляцию, мы вводим операционную семантику обхода графа, представляющего исполнение в модели ARMv8.3. Данный обход может быть напрямую симулирован абстрактной машиной, представляющей обещающую модель памяти.

3. Описание модели памяти ARMv8.3 на примерах

В этом разделе мы рассмотрим модель памяти процессора ARMv8.3 на примере программы **LB** из раздела 1 и вариации этой программы, для которой запрещено слабое поведение.



а) б) в) г)

*Рис. 1. Возможные исполнения программы **LB** в модели ARMv8.3*
*Fig. 1. Executions of program **LB**, which are allowed by model ARMv8.3*

Поскольку модель ARMv8.3 является аксиоматической, то семантика программы в этой модели представляется множеством графов, где вершины представляют *события*, т. е. операции над памятью, такие как чтение, запись или барьер памяти. На рис. 1 мы можем видеть все четыре возможных исполнения программы **LB** в рамках этой модели. Исполнения а), б) и в) соответствуют результатам (a = 0, b = 0), (a = 0, b = 1) и (a =1, b = 0), в то время как исполнение г) является слабым и представляет результат (a = 1, b = 1). Обратим внимание на исполнение б) и разберемся с обозначениями.

В случае исполнения б) в графе четыре операции, из которых две являются чтением (R(*x*, 0), R(*y*, 1)), а две – записью (W(*y*, 1), W(*x*, 1)). Также в каждом графе присутствуют инициализирующие записи, но в рис. 1 мы их опустили для краткости. Рёбрами в графе являются различные отношения между операциями. Так рёбра, подписанные po (program order), обозначают программный порядок между операциями, ребро rf (read from) ведёт в операцию чтения из соответствующей операции записи, а ребро rb (read before) – из операции чтения в операцию записи, когда чтение происходит из «более ранней» записи. Для определения «более ранних» записей используется отношение co (coherence order), которое является полным порядком на записях в одну локацию, но поскольку на рис. 1 нет двух записей в одну локацию, то отношение co не представлено.

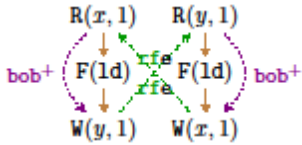
Для того, чтобы запретить результат (a = 1, b = 1), в программу могут быть вставлены *барьеры памяти*, которые запрещают процессору выполнение инструкций не по порядку. В рамках архитектуры ARMv8 существует несколько различных видов барьеров, в том числе полный барьер памяти, имеющий модификатор **SY**, и барьер на чтение, имеющий модификатор **LD**, который является ослабленной версией полного барьера. Существуют и другие модификаторы, но они не используется в схеме компиляции из обещающей модели памяти, рассмотренной в этой работе.

В случае программы **LB** достаточно вставить барьер на чтение в оба потока между операциями, чтобы запретить слабое поведение:

a := [*x*]; ‖ b := [*y*]; ‖

fence(**LD**);	fence(**LD**);		**(LB-LD)**
[*y*] := 1;	[*x*] := 1;		

Рассмотрим, каким образом модель памяти ARMv8.3 запрещает поведение (a = 1, b = 1) для программы **LB-LD**. Ниже представлен соответствующий граф исполнения:



В этом графе есть несколько изменений по сравнению с графом на рис. 1, г). Во-первых, в графе появились вершины, соответствующие барьерам памяти. Во-вторых, появились ребра, соответствующие отношению bob+, которое является транзитивным замыканием для bob. Отношение bob (barrier order before) связывает операцию чтения с операцией барьера по чтению, если барьер следует за чтением в программном порядке, и любую операцию, следующую за барьером. В-третьих, подписи отношения rf были заменены на подписи rfe, где e (external) обозначает, что данное ребро связывает события из разных потоков.

В результате в графе присутствует цикл из рёбер отношений bob и rfe. Такие циклы запрещены в модели памяти ARMv8.3, поэтому поведение (a = 1, b = 1) в целом не является *ARM-согласованным*. Под ARM-согласованным поведением мы понимаем граф, который не противоречит аксиомам модели ARMv8.3. Полный список аксиом приведён в [7].

4. Описание обещающей модели памяти на примерах

В этом разделе мы, аналогично разделу 3, рассмотрим обещающую модель памяти на примере программы **LB**, а также на двух вариациях этой программы, которые содержат барьеры памяти.

Обещающая модель памяти задана операционно, т. е. в терминах некоторой абстрактной машины, которую дальше мы будет называть *обещающей машиной*. Состояние этой машины включает *память* – множество сообщений, каждое из которых соответствует какой-то записи в локацию. Изначально память (M) содержит по инициализирующему сообщению на каждую локацию.

Каждое сообщение в памяти состоит из трёх компонент: целевой локации, значения и *метки времени.* Метки времени служат для того, чтобы упорядочивать сообщения, которые относятся к одной и той же локации, и задаются положительными рациональными числами.

Так, перед исполнением программы **LB**, память выглядит следующим образом:

$$M = \{<x:0@0_\tau>,\ <y:0@0_\tau>\},$$

где 0 – это записанное значение, а 0_τ – метка времени.

Исполнение программы в обещающей машине представляется как некоторое попеременное исполнение потоков. На каждом конкретном шаге активный поток может сделать одно из двух действий: либо выполнить следующую в программе инструкцию, либо *пообещать* сделать запись в память.

Для того, чтобы получить результат (a = 1, b = 1), в программе **LB** один из потоков должен начать свою работу с обещания. Пусть это будет левый поток. После того, как левый поток обещает сделать запись $<y:1@2_\tau>$, соответствующее сообщение добавляется во множество обещаний, еще не выполненных левым потоком, а также в память. После этого память содержит три сообщения:

$$M = \{<x:0@0_\tau>,\ <y:0@0_\tau>,\ <y:1@2_\tau>\}.$$

Теперь это сообщение является доступным для чтения другим потокам, поскольку оно находится в памяти, но не для левого потока, т. к. оно находится в его множестве еще не выполненных обещаний. После этого, правый поток может быть исполнен в обычном порядке.

Первым действием он производит чтение из нового сообщения в памяти, а вторым – записывает новое сообщение в локацию *x*. Теперь память содержит четыре сообщения:

$$M = \{<x:0@0_\tau>,\ <y:0@0_\tau>,\ <y:1@2_\tau>, <x:1@1_\tau>\}.$$

Далее левый поток читает из этого сообщения и в конце выполняет своё обещание, исполняя инструкцию записи в локацию *y*.

Точно так же, как и в случае модели ARMv8.3, для того, чтобы запретить результат (a = 1, b = 1), в программу могут быть добавлены барьеры памяти. Виды барьеров в обещающей семантике отличаются, т. к. обещающая модель памяти придерживается синтаксиса, описанного в стандартах C/C++11.

В рассматриваемом нами подмножестве обещающей машины есть два типа барьеров: *высвобождающий* (rel, release) и *приобретающий* (acq, acquire). При эффективной компиляции программы из синтаксиса обещающей модели в синтаксис ARMv8 высвобождающий барьер переходит в полный барьер памяти ARMv8, а приобретающий – в барьер по чтению.

Программа, результатом компиляции которой является **LB-LD**, выглядит так[2]:

a := [*x*];	‖	b := [*y*];	‖
fence(**ACQ**);	‖	fence(**ACQ**);	‖
[*y*] := 1;	‖	[*x*] := 1;	‖

(LB-ACQ)

2 Заметим, что для обоих программ используется одна и та же нотация, несмотря на то, что программа **LB-LD** рассматривается как программа на целевом языке компиляции, а программа **LB-ACQ** – на исходном языке.

Несмотря на то, что после компиляции данная программа не может иметь слабое поведение (a = 1, b = 1), сама программа **LB-ACQ** в рамках обещающей модели имеет данное поведение. Это не противоречит корректности компиляции, т. к. последняя утверждает, что любое поведение скомпилированной программы должно быть поведением изначальной программы, при этом обратное не утверждается.

Для того, чтобы запретить результат (a = 1, b = 1), нужно использовать высвобождающие барьеры:

a := [*x*];	‖	b := [*y*];	‖ **(LB-REL)**
fence(**REL**);	‖	fence(**REL**);	‖
[*y*] := 1;	‖	[*x*] := 1;	‖

Данный тип барьера запрещает потоку делать обещания через него, т. е. в момент исполнения самого барьера множество ещё не выполненных обещаний данного потока должно быть пусто. Поэтому при исполнении программы как минимум одно из чтений будет выполнено до того, как в памяти появится хотя бы одно сообщение со значением 1.

При компиляции высвобождающие барьеры переходят в полные барьеры, которые так же, как и барьеры по чтению, гарантируют отсутствие слабого поведения для соответствующей программы в рамках модели ARMv8.3.

5. Доказательство корректности компиляции

Основным результатом данной статьи является доказательство следующей теоремы.

Теорема 1. Для любой программы *Prog*, результата её компиляции *Prog*ARM и ARM-согласованного исполнения *G* программы *Prog*ARM существует исполнение *Prog* обещающей машиной т.ч. финальное состояние памяти машины совпадает с состоянием памяти *G*.

Под финальным состоянием памяти обещающей машины имеется ввиду подмножество сообщений, каждое из которых имеет максимальную метку времени среди сообщений, относящихся к той же локации. Состоянием памяти исполнения *G* считается множество событий-максимумов по отношению co, которое является аналогом меток времени в ARM-согласованных исполнениях.

5.1. Структура доказательства

Как видно из разделов 3 и 4, ARMv8.3 и обещающая модели заданы в совершенно разных стилях. Нужно это различие преодолеть для того, чтобы иметь возможность доказать теорему 1.

Наше доказательство базируется на введении промежуточной операционной семантики, которая совершает обход графа ARM-согласованного исполнения в специальном порядке, гарантирующим возможность обещающей машине

исполнять соответствующие инструкции одновременно с обходом. Имея упомянутую промежуточную семантику, мы можем показать, что обещающая машина может симулировать семантику обхода.

Метод симуляции является прямой индукцией по исполнению целевой машины. Первым шагом метода является определение отношения симуляции, которое тесно связывает состояния машин обоих моделей. Вторым шагом доказывается, что если целевая машина находится в состоянии *t*, а исходная — в состоянии *s*, и эти состояния связаны отношением симуляции, тогда для любого состояния *t'*, в которое может перейти целевая машина, существует состояние *s'*, в которое может перейти исходная машина, т.ч. *t'* и *s'* опять связаны отношением симуляции.

5.2. Обход ARM-согласованных исполнений

Данный обход мы представляем в виде операционной семантики некоторой абстрактной машины. Состояние машины состоит из пары множеств *(C, I)* и называется *конфигурацией обхода*. При этом пара *(C, I)* считается корректной конфигурацией обхода ARM-согласованного исполнения *G*, если выполняется утверждения, представленные ниже.

1. *C* и *I* являются подмножествами событий в графе *G*.
2. *C* является замкнутым по префиксу отношения po (программного порядка), т. е. если некоторое событие *e* принадлежит *C*, то и любое событие *u*, т.ч. из *u* есть po-ребро в *e*, тоже принадлежит *C*.
3. Все события из множества *I* являются событиями записи, и все события записи из множества *C* также являются элементами *I*.
4. Все события, которые являются инициализирующими записями в локации, являются элементами *C*.

Далее множество *C* мы будем называть множеством *покрытых* элементов (covered), а множество *I* – множеством *выпущенных* элементов (issued).

У абстрактной машины, соответствующей обходу исполнения, есть два правила перехода из одной конфигурацию в другую: *покрытие события* и *выпуск события*.

Правило покрытия позволяет для некоторого события *e* перейти из конфигурации *(C, I)* в конфигурацию *(C* ∪{*e*}*, I)*. При этом для события *e* должны выполняться условия, представленные ниже.

1. Все события, po-предшествующие *e*, должны принадлежать *C*.
2. Событие *e* не принадлежит C.

3. Если событие *e* является операцией чтения, то событие записи *w*, из которого читает *e* (т. е. между *w* и *e* есть ребро отношения rf), должно быть элементом множества *I*.

4. Если событие *e* является операцией записи, то оно должно быть элементом множества *I*.

Правило выпуска добавляет событие *e* во множество выпущенных, т. е. соответствует переходу из *(C, I)* в *(C, I* ∪{*e*}*)*. Событие *e* должно удовлетворять следующим ограничениям:

1. Событие *e* является операцией записи.
2. Событие *e* не является элементом множества *I*.
3. Все po-предшествующие барьеры памяти должны быть покрыты.
4. Все записи других потоков, от которых зависит событие *e*, являются выпущенными[3].

Так как оба правила только наращивают компоненты конфигурации, а любое исполнение мы считаем конечным графом, то для того, чтобы доказать наличие полного обхода ARM-согласованного исполнения *G*, достаточно показать, что из любой конфигурации *(C, I)*, т.ч. *C* не совпадает со множеством событий *G*, существует переход к новой конфигурации.

Лемма 2. Пусть *(C, I)* является корректной конфигурацией ARM-согласованного исполнения *G*, и *C* не совпадает со множеством событий *G*. Тогда существуют *C`* и *I`*, т.ч. есть правило перехода из *(C, I)* в *(C`, I`)*.

Идея доказательства. Если *C* не совпадает со множеством событий исполнения *G*, то множество не покрытых событий, т.ч. для каждого события из множества все po-предшествующие ему события покрыты, не пусто. Обозначим данное множество *Next*. Если в *Next* есть элемент, удовлетворяющий остальным требованиям правила покрытия, то существует соответствующий переход в новую конфигурацию. Если в этом множестве есть событие записи, то оно может быть выпущено согласно другому правилу обхода. Иначе все элементы *Next* соответствуют операциям чтения из ещё не выпущенных событий. В этом случае можно показать, что либо существует запись, которая может быть выпущена, либо в графе исполнения *G* существует цикл, который противоречит требованиям ARM-согласованности [7].

При симуляции обещающей машиной семантики обхода шаг обхода, являющийся покрытием события, будет соответствовать исполнению инструкции в обещающей машине (выполнение инструкции чтения, барьера памяти или сделанного ранее обещания), тогда как выпуск события будет соответствовать обещанию сообщения соответствующим потоком.

5.3. Симуляция обхода ARMv8.3 исполнения обещающей машиной

Как было замечено выше, доказательство корректности компиляции через симуляцию является индукционным доказательством по исполнению целевой

3 Формально, событие *e* является зависимым от записи другого потока *w*, если существует путь в графе исполнения *G* от *w* к *e* такой, что первое ребро данного пути является ребром отношения rfe, а все остальные ребра являются ребрами отношения dob (dependency ordered before), определение которого приведено в [7].

машины. Важным элементом такого доказательства является отношение симуляции на состояниях соответствующих машин. Это отношение должно выполняться для начальных состояний, что соответствует базе индукции, а также должно сохраняться при индукционном переходе. Из того, что отношение выполняется для финального состояния целевой машины, должно следовать заключение корректности компиляции, в нашем случае совпадение состояний памяти обещающей машины и ARM-согласованного исполнения.

Отношение симуляции *Inv* (инвариант, invariant) между состоянием обещающей машины *S* и корректной конфигурацией обхода *(C, I)* ARM-согласованного исполнения *G* выполняется, если память обещающей машины соответствует множеству выпущенных событий *I*, и для каждого события, которое может быть покрыто в текущей конфигурации, существует соответствующая инструкция в программе[4].

Более подробно, должны выполняться три следующих условия:

1. Для любого события записи *w* из множества выпущенных событий *I* существует сообщение *m* в памяти *S*, т.ч. *m* относится к той же локации, что и *w*, записывает то же значение и метка времени сообщения совпадает с порядковым номером *w* в отношении со (coherence order). При этом, если *w* также является покрытым (т. е. является элементом *C*), то сообщение *m* является выполненным, иначе – *m* еще не выполненное обещание.
2. Аналогично, для любого сообщения *m* из памяти *S* существует соответствующее ему событие во множестве выпущенных событий *I*.
3. Для цепочки событий каждого потока в исполнении *G* существует соответствующая серия переходов в графе потока управления программы. Кроме того, каждый поток обещающей машины уже выполнил часть цепочки, которая относится к покрытым событиям из *C*.

Имея отношение симуляции, нужно показать, что оно сохраняется при индукционном переходе. Из него напрямую следует утверждение теоремы 1, т. е. корректность компиляции.

Лемма 3. Пусть выполняется отношение симуляции *Inv(S, C, I, G)* и существует переход из конфигурации *(C, I)* в конфигурацию *(C`, I`)*, то существует шаг обещающей машины из состояния *S* в такое состояние *S`*, что выполняется *Inv(S`, C`, I`, G)*.

Идея доказательства. Переход из конфигурации *(C, I)* в конфигурацию *(C`, I`)* может быть либо покрытием события, либо выпуском события. Рассмотрим данные варианты и начнем с покрытия.

4 В отношении симуляции присутствуют и другие требования, относящиеся к компонентам обещающей машины, которые были опущены в данной статье для краткости.

Если переход в $(C`, I`)$ является покрытием некоторого события e, то $C` = C \cup \{e\}$ и $I` = I$. Согласно отношению симуляции, поток обещающей машины, к которому относится событие e, может сделать соответствующий переход. Если e является операцией чтения, то согласно требованию на покрываемость события оно читает из события записи, которое уже выпущено, т. е. является элементом I. Согласно отношению симуляции, в памяти обещающей машины есть соответствующее сообщение, и оно может быть прочитано при переходе от S к $S`$. Если $e$ является операцией записи, то оно является выпущенным согласно условию на покрытие, а соответствующее сообщение находится в памяти обещающей машины, т.о. при переходе от $S$ к $S`$ обещающая машина выполняет связанное обещание. Если e является операцией барьера, то, согласно условию выпускаемости события, не существует po-последующего события записи, которое выпущено. Из этого следует, что соответствующий поток обещающей машины не имеет невыполненных обещаний и может выполнить следующую инструкцию барьера памяти. Таким образом, обещающая машина может симулировать переход покрытия.

Если переход в $(C`, I`)$ является выпуском некоторого события записи w, то $C` = C$ и $I` = I \cup \{w\}$. Согласно требованию на выпускаемость события w, все po-предшествующие барьеры памяти покрыты, а значит, по отношению симуляции сообщение, соответствующее w, может быть обещано в рамках обещающей машины. Отношение симуляции при этом переходе, очевидно, сохранится. □

6. Заключение

Мы доказали корректность компиляции из подмножества обещающей модели памяти в модель памяти ARMv8.3. Для этого мы использовали новый метод обхода графов исполнений, что позволило нам базировать доказательство на симуляции. Данный метод накладывает меньше ограничений на целевую платформу по сравнению с методом, использованном для доказательства корректности компиляции в модели x86-TSO и Power в [13]. Последнее означает, что предложенный метод потенциально проще переиспользовать в последующих доказательствах корректности из обещающей модели памяти.

В нашей работе было рассмотрено базовое подмножество обещающей машины, состоящее из расслабленных чтений и записей, а также приобретающих и высвобождающих барьеров памяти. Нашей следующей задачей является расширение доказательства до полной обещающей модели. Для этого нужно будет рассмотреть более строгие варианты чтений и записей, инструкции атомарного чтения и записи (read-modify-write), частным случаем которых является операция чтения с возможным обменом (CAS, compare-and-swap), и полные барьеры памяти (SC fences). Далее мы опишем, какие модификации в доказательстве нужны для поддержки упомянутых инструкций.

Для того, чтобы добавить более строгое чтение, а именно приобретающее чтение (acquire read), в доказательство не нужно вносить никаких изменений. Это следует из того, что для обещающей модели памяти была доказана корректность оптимизации, которая заменяет приобретающее чтение на приобретающий барьер памяти и расслабленное чтение, а данные инструкции рассмотрены в нашем доказательстве.

Для поддержки более строго варианта инструкции записи (высвобождающей записи, release write) нужно показать, что в обходе ARM-согласованного исполнения при выпуске события, соответствующего высвобождающей записи, это событие может быть сразу же покрыто. Данное утверждение нужно, поскольку обещающая машина должна одновременно пообещать и выполнить обещание, относящееся к высвобождающей записи. При этом само утверждение может быть тривиальным образом показано.

Аналогичное утверждение о покрытии сразу двух событий должно быть доказано и для поддержки инструкций атомарного чтения и записи (CAS), т. к. в модели ARMv8.3 такие инструкции представляются двумя непосредственно следующими друг за другом событиями, тогда как в обещающей модели памяти атомарная инструкция выполняется за один шаг.

Полные барьеры памяти (SC fences) в обещающей модели памяти требуют некоторого фиксированного полного порядка на их исполнении. Для поддержки таких барьеров в доказательстве нужно расширить отношение симуляции, чтобы в нем появилась глобальная компонента, отражающая данный порядок. Также в обход ARM-согласованного исполнения нужно добавить ограничение на покрытие полных барьеров, гарантирующее их покрытие в том же порядке, в котором они могут быть исполнены в обещающей модели памяти.

Список литературы

[1]. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.

[2]. Aho A.V., Sethi R., Ullman J.D., Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[3]. Sewell P., Sarkar S., Owens S., Zappa Nardelli F., and Myreen M. O. x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors. Commun. ACM, 53(7):89–97, 2010. doi:10.1145/1785414.1785443.

[4]. Alglave J., Maranget L., and Tautschnig M.. Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst., 36(2):7:1–7:74, 2014. doi:10.1145/2627752.

[5]. Sarkar S., Sewell P., Alglave J., Maranget L., and Williams D. Understanding POWER multiprocessors. In PLDI 2011, pages 175–186. ACM, 2011. doi:10.1145/1993498. 1993520.

[6]. Flur S., Gray K. E., Pulte C., Sarkar S., Sezgin A., Maranget L., Deacon W., and Sewell P. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In POPL 2016, pages 608–621. ACM, 2016. doi:10.1145/2837614.2837615.

[7]. Pulte C., Flur S., Deacon W., French J., Sarkar S., and Sewell P. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. URL: http://www.cl.cam.ac.uk/~pes20/armv8-mca/armv8-mca-draft.pdf. July 2017.

[8]. Batty M., Owens S., Sarkar S., Sewell P., and Weber T. Mathematizing C++ concurrency. In POPL 2011, pages 55–66. ACM, 2011. doi:10.1145/1925844.1926394.

[9]. Manson J., Pugh W., and Adve S. V. The Java memory model. In POPL 2005, pages 378–391. ACM, 2005. doi:10.1145/1040305.1040336.

[10]. Трифанов В.Ю. Динамическое обнаружение состояний гонки в многопоточных Java-программах. Диссертация на соискание ученой степени кандидата технических наук. СПбГУ ИТМО, 2013, 112 стр.

[11]. Ševčík J. and Aspinall D. On Validity of Program Transformations in the Java Memory Model. In *Proceedings of the 22nd European conference on Object-Oriented Programming*(ECOOP '08), Jan Vitek (Ed.). Springer-Verlag, Berlin, Heidelberg, 27-51. doi: 10.1007/978-3-540-70592-5_3.

[12]. Boehm H.-J. and Demsky B. Outlawing ghosts: Avoiding out-of-thin-air results. In MSPC 2014, pages 7:1–7:6. ACM, 2014. doi:10.1145/2618128.2618134.

[13]. Kang J., Hur C.-K., Lahav O., Vafeiadis V., and Dreyer D. A promising semantics for relaxed-memory concurrency. In POPL 2017. ACM, 2017. doi:10.1145/3009837. 3009850.

[14]. Lahav O. and Vafeiadis V. Explaining relaxed memory models with program transformations. FM 2016. Springer, 2016. doi:10.1007/978-3-319-48989-6_29.

[15]. Обещающая компиляция в ARMv8.3. Формальное доказательство. Октябрь 2017. URL: https://podkopaev.net/armpromise

[16]. Leroy X. A formally verified compiler back-end. J. Autom. Reasoning, 43(4):363–446, 2009. doi:10.1007/s10817-009-9155-4.

[17]. Kumar R., Myreen M. O., Norrish M., and Owens S.. CakeML: A verified implementation of ML. In POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 179–191. ACM Press, January 2014. doi:10.1145/2535838.2535841.

[18]. Sevcík J., Vafeiadis V., Zappa Nardelli F., Jagannathan S., and Sewell P. Relaxed-memory concurrency and verified compilation. In Proceedings of the 38th ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 43–54, 2011. doi:10.1145/1926385.1926393.

[19]. Batty M., Owens S., Sarkar S., Sewell P., and Weber T. Mathematizing C++ concurrency: The post-Rapperswil model. technical report n3132, iso iec jtc1/sc22/wg21. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3132.pdf.

[20]. Batty M., Memarian K., Nienhuis K, Pichon-Pharabod J., and Sewell P. The problem of programming language concurrency semantics. In ESOP, volume 9032 of LNCS, pages 283–307. Springer, 2015. doi:10.1007/978-3-662-46669-8_12.

[21]. Podkopaev A., Lahav O., and Vafeiadis V. Promising compilation to ARMv8 POP. In ECOOP 2017. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.22.

Promising Compilation to ARMv8.3

[1,2] *A.V.Podkopaev <apodkopaev@2009.spbu.ru>*
[3] *O.Lahav <orilahav@tau.ac.il>*
[4] *V.Vafeiadis <viktor@mpi-sws.org>*
[1] *St. Petersburg University,*
199034, Russia, St. Petersburg, Universitetskaya emb. 7–9
[2] *JetBrains Research,*
199034, Russia, St. Petersburg, Universitetskaya emb. 7–9-11/5A
[3] *Tel-Aviv University, 39040, Israel, Tel Aviv 69978*
[4] *Max Planck Institute for Software Systems,*
67663, Germany, Kaiserslautern, Paul-Ehrlish str. G26

Abstract. Concurrent programs have behaviors, which cannot be explained by interleaving execution of their threads on a single processing unit due to optimizations, which are performed by modern compilers and CPUs. How to correctly and completely define a semantics of a programming language, which accounts for the behaviors, is an open research problem. There is an auspicious attempt to solve the problem – promising memory model. To show that the model might be used as a part of an industrial language standard, it is necessary to prove correctness of compilation from the model to memory models of target processor architectures. In this paper, we present a proof of compilation correctness from a subset of promising memory model to an axiomatic ARMv8.3 memory model. The subset contains relaxed memory accesses and release and acquire fences. The proof is based on a novel approach of an execution graph traversal.

Keywords: concurrency; compilation correctness; weak memory models.

DOI: 10.15514/ISPRAS-2017-29(5)-9

For citation: Podkopaev A.V., Lahav O., Vafeiadis V. Promising Compilation to ARMv8.3. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 149-164 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-9

References

[1]. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.
[2]. Aho A.V., Sethi R., Ullman J.D., Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.
[3]. Sewell P., Sarkar S., Owens S., Zappa Nardelli F., and Myreen M. O. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM, 53(7):89–97, 2010. doi:10.1145/1785414.1785443.
[4]. Alglave J., Maranget L., and Tautschnig M.. Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst., 36(2):7:1–7:74, 2014. doi:10.1145/2627752.

[5]. Sarkar S., Sewell P., Alglave J., Maranget L., and Williams D. Understanding POWER multiprocessors. In PLDI 2011, pages 175–186. ACM, 2011. doi:10.1145/1993498.1993520.

[6]. Flur S., Gray K. E., Pulte C., Sarkar S., Sezgin A., Maranget L., Deacon W., and Sewell P. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In POPL 2016, pages 608–621. ACM, 2016. doi:10.1145/2837614.2837615.

[7]. Pulte C., Flur S., Deacon W., French J., Sarkar S., and Sewell P. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. URL: http://www.cl.cam.ac.uk/~pes20/armv8-mca/armv8-mca-draft.pdf. July 2017.

[8]. Batty M., Owens S., Sarkar S., Sewell P., and Weber T. Mathematizing C++ concurrency. In POPL 2011, pages 55–66. ACM, 2011. doi:10.1145/1925844.1926394.

[9]. Manson J., Pugh W., and Adve S. V. The Java memory model. In POPL 2005, pages 378–391. ACM, 2005. doi:10.1145/1040305.1040336.

[10]. Trifanov V.Yu. Dynamic detection of race conditions in multithreaded Java programs. Dissertation, ITMO University, 2013, p. 112 (in Russian).

[11]. Ševčík J. and Aspinall D. On Validity of Program Transformations in the Java Memory Model. In Proceedings of the 22nd European conference on Object-Oriented Programming(ECOOP '08), Jan Vitek (Ed.). Springer-Verlag, Berlin, Heidelberg, 27-51. doi: 10.1007/978-3-540-70592-5_3.

[12]. Boehm H.-J. and Demsky B. Outlawing ghosts: Avoiding out-of-thin-air results. In MSPC 2014, pages 7:1–7:6. ACM, 2014. doi:10.1145/2618128.2618134.

[13]. Kang J., Hur C.-K., Lahav O., Vafeiadis V., and Dreyer D. A promising semantics for relaxed-memory concurrency. In POPL 2017. ACM, 2017. doi:10.1145/3009837.3009850.

[14]. Lahav O. and Vafeiadis V. Explaining relaxed memory models with program transformations. FM 2016. Springer, 2016. doi:10.1007/978-3-319-48989-6_29.

[15]. [Promising compilation to ARMv8.3. A formal proof]. October 2017. URL: https://podkopaev.net/armpromise

[16]. Leroy X. A formally verified compiler back-end. J. Autom. Reasoning, 43(4):363–446, 2009. doi:10.1007/s10817-009-9155-4.

[17]. Kumar R., Myreen M. O., Norrish M., and Owens S.. CakeML: A verified implementation of ML. In POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 179–191. ACM Press, January 2014. doi:10.1145/2535838.2535841.

[18]. Sevcík J., Vafeiadis V., Zappa Nardelli F., Jagannathan S., and Sewell P. Relaxed-memory concurrency and verified compilation. In Proceedings of the 38th ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 43–54, 2011. doi:10.1145/1926385.1926393.

[19]. Batty M., Owens S., Sarkar S., Sewell P., and Weber T. Mathematizing C++ concurrency: The post-Rapperswil model. technical report n3132, iso iec jtc1/sc22/wg21. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3132.pdf.

[20]. Batty M., Memarian K., Nienhuis K, Pichon-Pharabod J., and Sewell P. The problem of programming language concurrency semantics. In ESOP, volume 9032 of LNCS, pages 283–307. Springer, 2015. doi:10.1007/978-3-662-46669-8_12.

[21]. Podkopaev A., Lahav O., and Vafeiadis V. Promising compilation to ARMv8 POP. In ECOOP 2017. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.22.

Инструментальная поддержка создания и трансформации функционально-потоковых параллельных программ

[1]А.И.Легалов <legalov@mail.ru>
[1]В.С.Васильев <rrrFer@mail.ru>
[1]И.В.Матковский <alpha900i@mail.ru>
[1]М.С.Ушакова <ksv@akadem.ru>
[1]Сибирский федеральный университет,
660041, Российская Федерация, г. Красноярск, пр. Свободный, 79

Аннотация. В работе рассмотрен нетрадиционный подход к созданию параллельных программ, их анализу и трансформации с использованием языка функционально-потокового параллельного программирования, обеспечивающего написание программ без учёта ресурсных ограничений, что позволяет изначально ориентироваться на алгоритмы с максимальным параллелизмом. Разработанные инструментальные средства обеспечивают трансляцию, выполнение, отладку, оптимизацию и верификацию функционально-потоковых параллельных программ. Выполнение разработанных программ осуществляется путём «сжатия» параллелизма с учётом ограниченных ресурсов реальных вычислительных систем. Вычислительный процесс рассматривается как наложение управляющего графа, определяющего организацию вычислений, на информационный граф, описывающий функциональные особенности реализуемого алгоритма. Возможно использование различных стратегий управления путём модификации управляющих графов. Предложенные инструменты обеспечивают формирование промежуточных представлений, на основе которых возможны дальнейшие преобразования исходных программ в программы для реальных архитектур параллельных вычислительных систем.

Ключевые слова: архитектурно-независимое параллельное программирование; функционально-потоковое параллельное программирование; трансформация программ; средства разработки программ; информационный граф, управляющий граф.

DOI: 10.15514/ISPRAS-2017-29(5)-10

Для цитирования: Легалов А.И., Васильев В.С., Матковский И.В., Ушакова М.С. Инструментальная поддержка создания и трансформации функционально-потоковых параллельных программ. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 165-184. DOI: 10.15514/ISPRAS-2017-29(5)-10

1. Введение

Параллельное программирование давно перестало использоваться только для высокопроизводительных вычислений. Вместе с тем, подходы к разработке параллельных программ мало изменились по сравнению с 80-ми годами прошлого века. Как и раньше, наиболее применяемым является написание кода с учётом особенностей целевой вычислительной системы. При переносе программы на другую архитектуру приходится либо полностью её переписывать, либо подвергать существенной модификации. Попытки перейти на нетрадиционные параллельные вычислительные системы (ПВС), как и ранее, подвергаются критике, основной идеей которой является несовместимость создаваемых для них программ с уже существующими. Хотя о такой несовместимости можно говорить практически для всех вновь разрабатываемых параллельных программ [1, 2].

Широкое применение императивного подхода вносит в разработку параллельных программ свои сложности. Программист явно или неявно формирует отношения между программными объектами, которые можно охарактеризовать следующим образом [3]:

- отношения между данными (информационные отношения), или информационный граф программы, задающий связь между обрабатываемыми данными и операциями;
- отношения управления, определяющие порядок выполнения операций и функций программы;
- отношения между вычислительными ресурсами (памятью, процессорными узлами), в которых должны выполняться операции, описываемые в программе.

В большинстве случаев разработчику программы приходится явно учитывать взаимосвязь между этими отношениями, добиваясь того, чтобы не возникали логические противоречия, ведущие к ошибочному выполнению. Для современных параллельных программ это труднодостижимо, что затрудняет отладку и верификацию, несмотря на применение специализированных инструментальных средств, поддерживающих, например, формальную верификацию с использованием проверки моделей (model checking) [2, 4]. При переходе на другую архитектуру весь процесс разработки и отладки практически приходится повторять заново.

Помимо этого, наличие зависимости от конкретных архитектур, сочетающих параллелизм с последовательным программированием, не способствует внедрению истинно параллельных алгоритмов на начальных стадиях их разработки. Это ведёт к искажению исходного параллелизма задачи, зависимости от ресурсных ограничений, которые при последующем распараллеливании не всегда позволяют прийти к более эффективным решениям.

Несмотря на это, ключевым направлением в параллельном программировании является разработка архитектурно-зависимых параллельных программ. Существуют различные подходы, сильно отличающиеся по идеологии поддержания параллелизма. К наиболее популярным среди них следует отнести: распараллеливание на основе передачи сообщений [5], многопоточное и многоядерное программирование для систем с общей памятью [6], применение графических ускорителей [7], а также сочетание всех трех подходов в различных комбинациях при программировании для гибридных и распределенных архитектур [8-11].

Несмотря на то, что концепция неограниченного параллелизма в настоящее время не пользуется популярностью при разработке реальных параллельных программ [12], у нее имеются определенные перспективы в качестве основы для формирования систем программирования, обеспечивающих последующую трансформацию в ресурсно-ограниченные и архитектурно-зависимые параллельные программы. Поэтому актуальной является задача создания языковых и инструментальных средств, обеспечивающих построение параллельных программ, не связанных на начальных этапах разработки с архитектурами реальных ПВС, выбор которых мог бы осуществляться уже после проведения процессов верификации, тестирования и отладки.

2. Особенности модели вычислений и языка функционально-потокового параллельного программирования

Основными идеями предлагаемого подхода, определяющими концепцию архитектурно-независимого параллельного программирования, являются исключение ресурсных ограничений и неявное управление вычислениями. Предполагается, что виртуальная машина, имеющаяся в распоряжении программиста, обладает неограниченными вычислительными ресурсами, а язык программирования позволяет описывать только информационные зависимости между выполняемыми функциями. Взаимодействие между функциями при этом осуществляется по готовности данных, что позволяет создавать программы с максимально возможным параллелизмом, сжатие которого к конкретным вычислительным ресурсам и условиям эксплуатации будет происходить после верификации и отладки написанных программ на уровне исходного представления. Это позволяет повысить эффективность процесса разработки параллельных программ. Например, можно создать единую библиотеку функций, адаптируемую к различным архитектурам вычислительных систем.

Для реализации системы, ориентированной на создание архитектурно-независимых параллельных программ, необходимо обеспечить соответствующую поддержку на уровне модели вычислений. Лежащая в

основе языка программирования модель функционально-потоковых параллельных вычислений [13] задается тройкой:

$$\mathbf{M = (G, P, S_0)},$$

где **G** - ациклический ориентированный граф, определяющий информационную структуру программы (ее информационный граф), **P** - набор правил, определяющих динамику функционирования модели (механизм формирования разметки), $\mathbf{S_0}$ - начальная разметка.

Информационный граф

$$\mathbf{G = (V, A)},$$

где **V** - множество вершин, определяющих программо-формирующие операторы, а **A** - множество дуг, задающих пути передачи информации между ними.

Имеется только один оператор, реализующий функцию: оператор интерпретации. Он принимает два аргумента, один из которых является значением функции, а другой – исходными данными, подлежащими обработке (рис. 1).

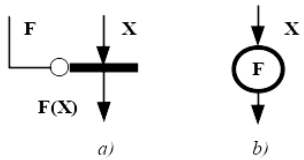


Рис. 1. Графическое обозначение оператора интерпретации (a – общая форма, b – в случае, когда значение функции известно)
Fig. 1. Graphical designation of the interpretation operator (a is the general form, b is in the case when the value of the function is known)

На выходе формируется результат. Для достижения более наглядного текстового представления программ с управлением по гот)овности данных оператор имеет префиксную и постфиксную форму записи, что позволяет выражение **F(X)** представить двумя способами:

X:F или **F^X**.

Функции могут являться изначально заданными элементарными операциями или создаваться программистом. Аксиоматика модели и её алгебра преобразований определяют базовые варианты реализации этого оператора.

Остальные операторы модели являются программо-формирующими, порождая связи между функциями интерпретации за счёт группировки порождаемых ими данных в различные структуры (списков), базовый набор которых представлен на рис. 2 и включает: копирование данных, формирование констант, создание списка данных из независимых величин, формирование параллельного списка, список задержанных вычислений.

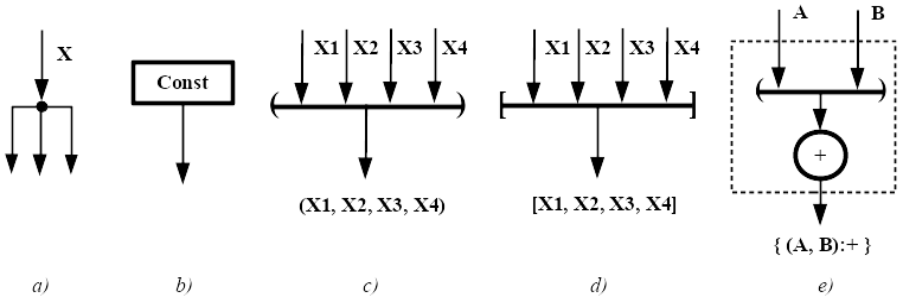


Рис. 2. Графическое обозначение программо-формирующих операторов (a – копирование данных, b – задание константы, c – группировка в список данных, d – группировка в параллельный список, e – создание задержанного списка)
Fig. 2. Graphical designation of program-forming operators (a - copying data, b – constant assignment, c – grouping in the data list, d - grouping in the parallel list, e - creating the delayed list)

Оператор копирования (рис. 2a) обеспечивает размножение данных. В языке определяется через именование значения и дальнейшее использование введенного обозначения в требуемых точках программы. Используются как префиксное, так и постфиксное именование связей:

значение >> имя, или **имя << значение**.

Константный оператор не имеет входов (рис. 2b). У него есть только один выход, на котором постоянно находится разметка, определяющая предписанное значение. В языковом представлении константный программо-формирующий оператор задается значением соответствующего типа.

Оператор группировки в список данных (рис. 2c) имеет несколько входов и один выход. Он обеспечивает структуризацию и упорядочение данных, поступающих по дугам из различных источников. Порядок элементов определяется номерами входов, каждому из которых соответствует натуральное число в диапазоне от **1** до **N**. В текстовом виде задается ограничением элементов списка круглыми скобками «**(**» и «**)**». Например:

(X1, X2, X3, X4).

Оператор создания параллельного списка (рис. 2d) группирует элементы, аналогично списку данных. Однако на выходе предполагается наличие множественной связи, кратность которой определяется количеством входов в оператор. При выполнении оператора интерпретации осуществляется параллельная обработка всех аргументов одной функцией:

[x, y, z]:sin ≡ [x:sin, y:sin, z:sin].

Аналогично при наличии списка функций все они параллельно обрабатывают один и тот же аргумент:

(x, 0):[<, =, >] ≡ [(x, 0):<, (x, 0):=, (x, 0):>].

В целом алгебра преобразований языка описывает различные более общие эквивалентные преобразования параллельных списков.

Оператор группировки в задержанный список (рис. 2е) задается вершиной, содержащей допустимый вычисляемый подграф (находится внутри контура, выделенного пунктиром), в котором возможно несколько входов и выходов. Входы связаны с дугами, определяющими поступление аргументов, а выход определяет выдаваемый из подграфа результат (в общем случае в виде параллельного списка). Особенностью такой группировки является то, что ограниченные данным оператором вершины, не могут выполняться, даже при наличии всех аргументов. Их активизация возможна только при снятии задержки (раскрытии контура), когда ограниченный подграф становится частью всего вычисляемого графа. В языке список задержанных вычислений задается охватом соответствующих операторов фигурными скобками «**{**» и «**}**». Раскрытие происходит, когда задержанный список оказывается одним из аргументов оператора интерпретации. Использование задержанного списка определяет формирование условных ветвлений в программе.

Применение программо-формирующих операторов и алгебры преобразований заимствованы из работы [14] и сопоставимо с представленными в ней функциональными формами. Однако в данном случае именно разнообразие этих операторов является основной идеей по повышению гибкости при написании параллельных программ и реализации различных нетрадиционных идей к построению параллельных алгоритмов.

В качестве примера приведено описание функции нахождения факториала с расщеплением диапазона от 1 до **n** пополам и параллельным вычислением для левой и правой ветвей. Головная функция **parfact** осуществляет проверку начального значения аргумента для определения хода вычислений. Эта проверка на равенство нулю осуществляется операциями меньше, равно и больше одновременно в соответствии с алгеброй эквивалентных преобразований языка:

```
parfact<< funcdef n {
  selector<< (n,0):(<,=,>):?;
  selector^ (-1, 1, {(n,1):mult}):. >>return
}
```

В результате сравнения аргумента **n** с нулевым значением возвращается список данных с булевскими величинами, определяющими истинность каждого из проверяемых условий. Встроенная функция «**?**» формирует на выходе параллельный список с номерами элементов, принявшими истинные значения в ходе проведенной проверки. В данном случае это будет одно из трех значений, которое и выступает в качестве селектора одного из вариантов дальнейшего выполнения данной функции. Например, при **n = 5** формирование селектора будет осуществляться следующим образом:

```
(5,0):(<,=,>):? => (false, false, true):? => 3
```

В операции интерпретации сформированное целочисленное значение выступает в роли индекса, по которому из списка данных выбирается соответствующий элемент. При значении **n > 0** из списка с альтернативными параметрами выбирается задержанный список, который раскрывается оператором интерпретации, содержащим в качестве функции пустое значение, обозначенное точкой:

```
{(n,1):mult}:. => (n,1):mult
```

После раскрытия задержанного списка запускается функция mult, вычисляющая произведение чисел от 1 до **n** с использованием рекурсивно-параллельного алгоритма, расщепляющего диапазон до элементарных операций умножения с последующей сверткой частичных произведений:

```
mult<< funcdef pair {
  selector<< pair:(<,=,>):?;
  selector^ (
    -1,
    {pair:1},
    {([(pair:1,((pair:+,2):div>>half,1):+),
       (half, pair:2)
      ]:mult):*
    }
  ):. >>return
}
```

Первоначально, как и в предшествующей функции, осуществляется проверка диапазона. Сформированный селектор позволяет выбрать из списка данных одно из трех значений: -1, если диапазон задан некорректно; одно значение, если оба числа равны; задержанный список, осуществляющий дробление диапазона пополам и одновременное рекурсивное вычисление произведения для полученных интервалов. Встроенная функция **div** обеспечивает при этом целочисленное деление, позволяя тем самым найти середину диапазона. Информационный граф данной функции представлен на рис. 3.

3. Инструментальная поддержка функционально-потокового параллельного программирования

Для использования подхода разработан ряд инструментальных средств, обеспечивающих поддержку функционально-потоковой парадигмы параллельного программирования. Разработанный язык, обеспечивает представление параллелизма на уровне элементарных операций, при котором каждая функция описывает только информационный граф алгоритма без каких-либо управляющих связей. Транслятор преобразует исходный текст функции в промежуточное представление, которое используется для оптимизации существующих зависимостей по различным критериям, а также для построения на его основе управляющего графа, задающего порядок выполнения в соответствии с выбранной стратегией управления

вычислениями [3]. Трансформация управляющего графа и его оптимизация позволяют получать стратегии, отличающиеся от управления по готовности данных и учитывающие различные ограничения, свойственные, например, реальным вычислительным системам.

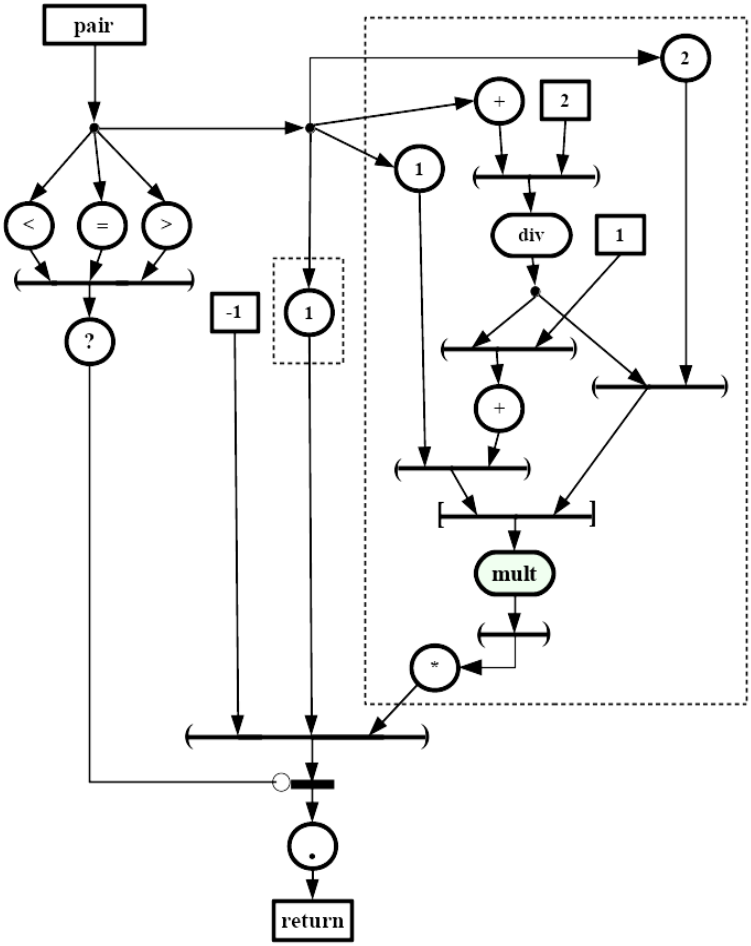


Рис. 3. Графическое представление функции перемножения в заданном диапазоне положительных чисел

Fig. 3. Graphical representation of multiply function for given range of positive numbers

Общая схема, отображающая различные варианты использования предлагаемых инструментальных средств, приведена на рис. 4. В рамках создаваемой среды выделяются следующие подсистемы:

- транслятор с языка функционально-потокового параллельного

программирования в промежуточное представление, называемое реверсивным информационным графом (РИГ);

- генератор управляющего графа (УГ), формирующий граф управления вычислениями;
- событийная машина, обеспечивающая выполнение функционально-потоковых параллельных (ФПП) программ в автоматическом и отладочном режимах, использующая в качестве программы РИГ и УГ;
- средства оптимизации реверсивного информационного графа;
- средства оптимизации управляющего графа;
- средства формальной верификации ФПП программ;
- средства преобразования ФПП программ в программы для других архитектур ПВС.

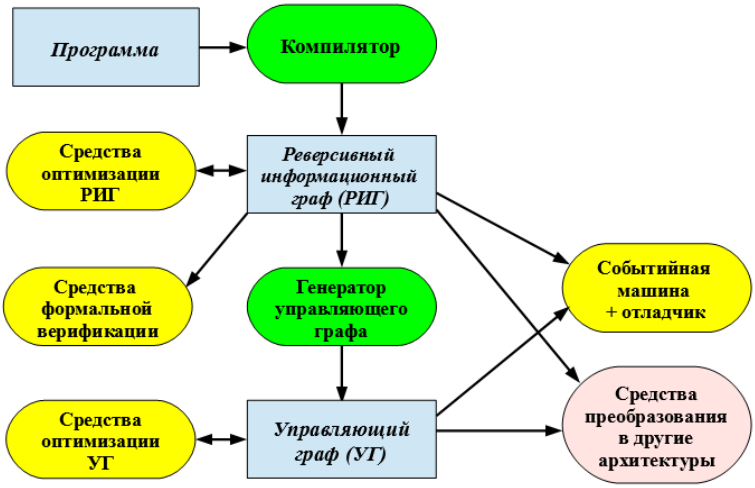


Рис. 4. Состав инструментальных средств, поддерживающих функционально-потоковое параллельное программирование
Fig. 4. System of tools for functional-dataflow parallel programming support

4. Трансляция функционально-потоковых параллельных программ

Транслятор ориентирован на обработку текстовых файлов, каждый из которых может содержать множество функций, написанных на языке ФПП программирования «Пифагор» [13]. Для каждой функции в памяти компьютера порождается РИГ, который сохраняется в репозитории функций в

текстовой форме. Выбор текстового представления для описания РИГ обусловлен тем, что формирование на его основе внутреннего представления в памяти компьютерной системы может быть легко выполнено с помощью простых транслирующих программ. Помимо этого разработчик может легко читать и анализировать оттранслированные функции, рассматривая данную форму как аналог языка ассемблера.

Формируемое для каждой функции в процессе синтаксического анализа промежуточное представление определяет свой РИГ как набор взаимосвязанных структур данных. Перед завершением трансляции эти представления сохраняются в текстовых файлах и могут использоваться внешними программами. В частности, функция вычисления факториала **parfact** преобразуется в следующее представление РИГ:

```
External
    0   parfact
    1   mult
Local
    0   0
    1   -1
    2   1
    3   1
    4   {1}7
id  delay   operation   links
0   0       arg
1   0       (---)       0 loc:0
2   0       (---)       < = >
3   0       :           1 2
4   0       :           3 ?
5   1       (---)       0 loc:3
6   1       :           5 ext:1
7   1       [---]       6
8   0       (---)       loc:1 loc:2 loc:4
9   0       :           8 4
10  0       :           9 .
11  0       return      10
```

При наличии внешних функций и констант данное представление содержит на них ссылки (ссылка на функцию **mult**). Помимо этого в нем задаются внутренние константы, а также реверсивный информационный граф заданной функции. Область описаний внешних ссылок начинается с ключевого слова ***External*** и содержит список строк, в каждой из которых задан дескриптор (номер) внешней ссылки и её имя. На нулевой позиции всегда располагается ссылка на транслируемую функцию. Это позволяет использовать обращение функции к самой себе в случае рекурсивных вызовов.

Область локальных констант начинается с ключевого слова ***Local*** и содержит список констант, используемых в ходе выполнения функции. Каждой константе соответствует свой дескриптор (номер), который при выполнении

программы обеспечивает доступ к соответствующим данным. Наряду с числовыми и символьными данными в этой области хранятся константы, определяющие параметры задержанных списков. Каждая из таких констант хранит номер задержанного списка, а также дескриптор узла РИГ, возвращающего вычисленное значение из данного задержанного списка (подобный узел существует в любом задержанном списке).

Описание РИГ содержит список вершин, их связей с локальными константами и внешними ссылками. Данная область начинается с заголовка, описывающего содержание каждой строки:

id delay operation links

Столбец ***id*** задаёт дескриптор (номер) вершины РИГ. В столбце ***delay*** указывается номер задержанного списка, в котором расположена соответствующая вершина. Если вершина РИГ не находится в задержанном списке, в данный столбец заносится ноль. В столбце ***operation*** размещается операция, выполняемая в соответствующей вершине РИГ. Столбец ***links*** указывает на связи вершин. В нем задаётся список ссылок на источники данных для текущей вершины. Источниками информации могут быть: внешние ссылки, локальные константы, предопределённые символы и узлы РИГ. Каждый из этих источников данных идентифицируется в качестве элемента списка связей с использованием своего варианта представления. Различие в описании связей в дальнейшем определяет обращение к разным областям памяти в вычислителе.

Указание на внешнюю ссылку задаётся в следующем формате:

ext:<символическое_имя_внешней_ссылки>

Для локальной константы используется формат:

loc:<значение_константы>

Предопределённые символы, обычно связанные со знаками различных операций, задаются своими значениями, например: +, -, * и так далее. Если источником операндов служит другая вершина РИГ, то в качестве связи задаётся целое число, равное дескриптору этой вершины.

Помимо этого в дополнительных файлах содержится информация для отладки программы, связывающая узлы РИГ с исходным текстом функции. Также, при наличии типов формируется файл, в котором описываются типы данных, порождаемые в каждом из узлов РИГ.

5. Формирование управляющего графа

Реверсивный информационный граф, порождённый транслятором, позволяет построить управляющий граф, определяющий выполнение функции. Для этого предназначена специальная утилита, которая формирует УГ, задающий управление вершинами РИГ по готовности данных. УГ сохраняется в

репозитории в текстовом виде. Для ранее приведённой функции вычисления факториала он будет выглядеть следующим образом:

```
parfact
id  delay   automat     inode   links
0   0       arg,0       0       links:1,1;5,1;
1   0       (---),0     1       links:3,1;
2   0       (---),0     2       links:3,2;
3   0       :,0         3       links:4,1;
4   0       :,0         4       links:9,2;
5   1       (---),0     5       links:6,1;
6   1       :,0         6       links:7,1;
7   1       [---],0     7
8   0       (---),0     8       links:9,1;
9   0       :,0         9       links:10,1;
10  0       :,0         10      links:11,1;
11  0       return,0    11
Signals: (Number, Node, Input)
0   1   2
1   2   1
2   2   2
3   2   3
4   4   2
5   5   2
6   6   2
7   8   1
8   8   2
9   8   3
10  10  2
Dynamic links: (Number, Delay list, Node)
0   1   7
```

Первая строка формируемого представления служит для идентификации УГ. В ней задаётся имя выполняемой функции. Следующая строка носит справочный характер, описывая названия столбцов управляющего графа, узлы которого располагаются в виде списка строк непосредственно за ней. Столбец ***id*** используется для задания дескриптора (номера) узла УГ. В столбце ***delay*** указывается номер задержанного списка, в который входит тот или иной узел ИГ. Если узел не входит ни в один из имеющихся задержанных списков, то в данном поле стоит значение, равное нулю. В столбце ***automat*** задаётся тип автомата, используемого для управления узлом, и через запятую указывается его начальное состояние. Обычно при формировании УГ автоматы находятся в начальном (нулевом) состоянии. Их состояния могут изменяться в ходе оптимизации управляющего графа с использованием соответствующих утилит. В столбце ***inode*** указывается узел РИГ, непосредственно связанный с данной вершиной УГ. При первоначальной генерации УГ количество его вершин совпадает с числом вершин РИГ, и вершины обоих графов находятся в однозначном соответствии. Однако в ходе оптимизации УГ, возможна

модификация этого соответствия. Может измениться как состояние автомата, размещённого в вершине УГ, так и его тип. Столбец ***links*** указывает на список управляющих связей, направленных от источников управляющих сигналов к их приёмникам. В отличие от РИГ, в УГ направление связей определяется таким образом, чтобы каждый источник данных инициировал выполнение процессов в связанных с ним приёмниках. Описание каждой связи управляющего графа содержит номер узла-приёмника и номер входа в нем.

Область управляющих сигналов следует после описания вершин УГ, отделяясь от него заголовком:

Signals: (Number, Node, Input)

Представленные в УГ сигналы обеспечивают начальную инициализацию вычислений, определяя готовность к использованию констант РИГ. Именно эти сигналы, наряду с сигналом, информирующим о появлении аргумента функции, определяют начало вычислений. Каждый сигнал имеет свой дескриптор (номер) и задаёт узел-приёмник, а также номер входа в этом узле.

Для перемещения между узлами РИГ в ходе вычислений нераскрытых задержанных списков необходима дополнительная управляющая информация. Она задаётся в области задержанных связей. Описание каждой связи содержит номер задержанного списка и узел РИГ, являющийся источником результата, порождаемого раскрытым задержанным списком. Данная область начинается с заголовка:

Dynamic links: (Number, Delay list, Node)

6. Параллельная событийная машина

Выполнение функционально-потоковых параллельных программ на текущем этапе осуществляется специальным интерпретатором (событийной машиной), состоящим из множества событийных процессоров (СП), управляемых менеджером событийной машины. Каждый из этих процессоров (рис. 5) осуществляет обработку только одной функции, запускаемой в отдельном потоке. Выполнение операций внутри функции в настоящий момент осуществляется последовательно за счёт изменения состояния вершин УГ, которые инициируют вычисления в вершинах РИГ.

Функционирование СП осуществляется следующим образом. Исходные сигналы, фиксирующие протекание в системе различных событий и определяемые начальной разметкой УГ, загружаются в очередь, из которой передаются обработчику в соответствии с дисциплиной обслуживания. В простейшем случае это может быть дисциплина FIFO. Обработчик управляющих сигналов анализирует поступившее событие и выбирает указанный в нем узел управляющего графа. На основе анализа состояния узла УГ он может обратиться к связанной с ним вершине информационного графа за кодом выполняемой операции. В случае, когда операция обработки данных должна быть выполнена, происходит обращение к обработчику узлов РИГ,

который осуществляет требуемые функциональные преобразования и сохраняет промежуточные результаты. После обработки данных управляющий узел переходит в новое состояние и при необходимости формирует сигнал, передаваемый следующему узлу, который поступает в очередь управляющих сигналов.

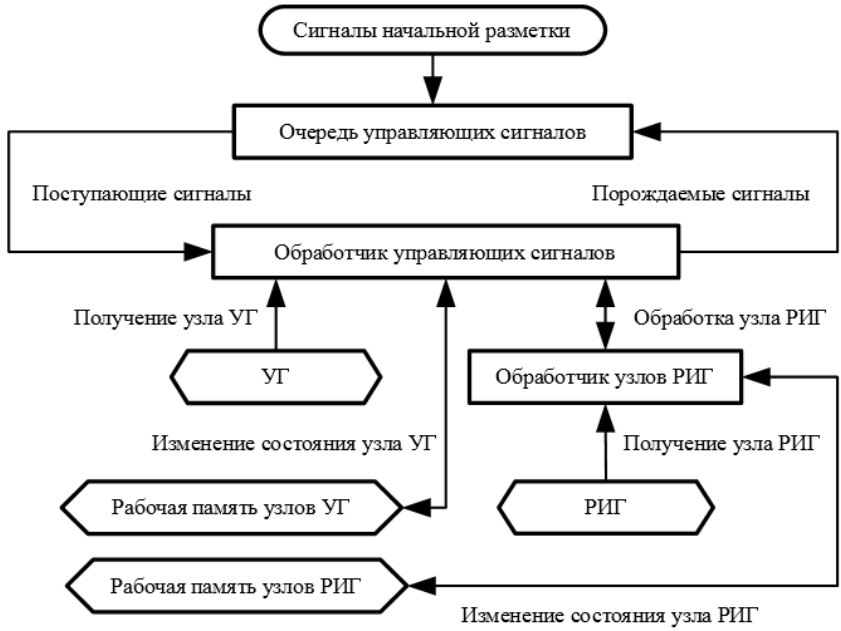


Рис. 5. Обобщённая структура событийного процессора
Fig. 5. Event processor's general structure

Перед запуском событийной машины производится компоновка программы из отдельных функций, размещённых в репозитории. Компоновщик проверяет наличие всех элементов, перечисленных в разделе внешних ссылок РИГ. В том случае, если какой-то из этих элементов отсутствует, интерпретация объявляется невозможной. Для каждого из найденных элементов также производится компоновка. Все найденные в процессе компоновки РИГ и УГ сохраняются в памяти, откуда легко могут быть извлечены, при необходимости, по имени. Таблицу загруженных компоновщиком РИГ и УГ хранит менеджер событийной машины. Процесс интерпретации начинается с создания первого (начального) СП. Ему передаются ссылки на РИГ и УГ той функции, с которой выполнение должно начаться. Процессор сохраняет данные в рабочей памяти узлов РИГ, а состояния автоматов в рабочей памяти узлов УГ, после чего начинает анализ управляющего графа. Автоматы, связанные с вершинами УГ, которым в РИГ соответствуют константы, будут изначально находиться в состоянии порождения новых событий. Эти события,

следуя по связям УГ, активируют принимающие их автоматы. Процесс передачи событий по связям продолжается до тех пор, пока не будет обработана вершина, соответствующая в РИГ вершине «***return***» (в этом случае функция считается выполненной), или пока очередь событий не опустеет. В последнем случае СП перейдёт в спящий режим, предварительно оповестив об этом менеджер событийной машины. Данная ситуация возможна, когда обработаны все внутренние сигналы и не пришли управляющие сигналы, сигнализирующие о получении результатов в вызываемых функциях.

7. Оптимизация функционально-потоковых параллельных программ

Основные методы оптимизации, разработанные в настоящее время, затрагивают преобразование промежуточных представлений функционально-потоковых параллельных программ. Они направлены на изменение информационного и управляющего графов. Проводимые преобразования во многом аналогичны методам, используемым при оптимизации исходных текстов программ и их промежуточных представлений в других языках программирования, и предназначены для решения схожих задач. Специфика функционально-потоковой модели параллельных вычислений накладывает свои особенности на реализацию этих методов. Она обусловлена алгеброй эквивалентных преобразований модели, реализованной в языке: информационный и управляющий графы могут изменяться независимо друг от друга. В ходе оптимизации необходимо обеспечивать согласованность РИГ и УГ, однако, для многих задач достаточно обработки РИГ. В таких случаях оптимизация управляющего графа должна осуществляться после трансформации информационного графа и построении на его основе нового УГ. Следует отметить, что разрабатываемые в настоящее время утилиты не затрагивают распределение реальных вычислительных ресурсов. В настоящий момент реализованы следующие методы.

1. Удаление неиспользуемого кода, то есть если вычисления не влияют на получение результата, то их можно удалить из функции. Соответствующая утилита проходит по реверсивному информационному графу от вершины возврата из функции и собирает все используемые вычисления, остальные — удаляются. Специфика данного преобразования определяется особенностью функционального языка, в котором каждая функция должна завершаться возвратом результатов.

2. Оптимизация в повторяющихся вычислениях. Традиционно в компиляторах подобное преобразование выполняется для циклов. В рассматриваемом случае аналогичным образом осуществляется преобразование рекурсии и имеющихся в модели и языке параллельных списков. В частности:

- при оптимизации в рекурсивной функции, вычисления, не зависящие по данным от изменяемых параметров рекурсии, выносятся во

вспомогательную функцию, а результат передаётся в основную функцию в качестве дополнительного параметра;

- при оптимизации параллельных списков, анализируется функция, применяемая к списку, и из неё в вызывающую функцию выносятся вычисления, не зависящие от аргумента [15].

3. Непосредственная (inline) подстановка простых функций. Если функция достаточно мала, то доля полезных вычислений в ней может быть невелика относительно издержек на вызов функции. В связи с этим тело функции вставляется вместо ее вызова.

4. Удаление повторяющегося кода. Если подграфы информационного графа выполняют одинаковые действия над одними и теми же аргументами и при этом находятся в одном или иерархически вложенных задержанных списках, то их можно слить, удалив тем самым избыточные вычисления. В реализации этого преобразования из РИГ сначала устраняются повторяющиеся константы. Тогда, вершины являются «повторяющимися» только в том случае, если выполняют одну и ту же операцию над одними и теми же узлами.

5. Оптимизация на основе эквивалентных преобразований, определяемых алгеброй преобразований модели функционально-потоковых параллельных вычислений. В РИГ выполняется поиск конструкций определённого вида и их трансформация в эквивалентную, но более простую для вычисления форму. В частности, модель допускает следующие эквивалентные преобразования:

- упрощение одноэлементного параллельного списка;
- свертку в один непосредственно вложенных параллельных списков;
- предварительное упрощение для списков, размер которых известен при компиляции.

6. Удаление избыточных управляющих зависимостей. Реверсивный информационный граф описывает зависимости по данным, а управляющий строится на его основе в соответствии со стратегией управления по готовности данных. Однако в ряде случаев часть построенных управляющих связей является избыточной. Поэтому их удаление не скажется на порядке выполнения программы.

7. Порождение управляющего графа, реализующего последовательный обход вершин РИГ. В данном случае осуществляется формирование вершин управляющего графа обеспечивающих запуск вершин информационного графа без дополнительного анализа готовности данных.

8. Формальная верификация ФПП программ

Наличие в программе только информационных зависимостей и отсутствие ресурсных ограничений позволяют облегчить формальную верификацию. Основными задачами в данном направлении работ являются: исследование

специфики применения формальных методов доказательства корректности; разработка инструментальных средств, упрощающих верификацию.

Акцент сделан на доказательство корректности программы с использованием дедуктивного анализа на основе исчисления Хоара [16]. Тройка Хоара представляется как информационный граф программы, к входной и выходной дуге которого привязаны формулы на языке спецификации (предусловие и постусловие). Процесс доказательства корректности программы заключается в разметке дуг информационного графа формулами на языке спецификации, в модификации графа и его свёртке. В результате, получается несколько информационных графов, у которых все дуги размечены. Каждый из полностью размеченных графов может быть преобразован в формулу на языке логики. Тождественная истинность всех полученных формул свидетельствует о корректности программы [17].

Процесс доказательства достаточно трудоёмок, так как требует рассмотрения большого количества различных вариантов графов и преобразований. Поэтому разработаны основные концепции архитектуры инструментального средства для поддержки формальной верификации программ на языке ФПП программирования [18]. Система получает на вход информационный граф программы и формулы предусловия и постусловия на языке спецификации. Она находит неразмеченные дуги графа и помогает с выбором аксиом и теорем, необходимых для их разметки. Весь процесс доказательства представляется в виде дерева, каждый узел которого является частично размеченным графом. Построение дерева завершается, когда все его листья содержат полностью размеченные информационные графы программы. После этого для каждого графа из листа генерируется формула на языке логики. Если все формулы тождественно истинны, то программа корректна.

9. Заключение

Разработанные инструментальные средства позволяют создавать архитектурно-независимые параллельные программы, на которые впоследствии можно накладывать различные стратегии управления вычислениями без изменения логики функционирования. Ничто не мешает проводить предварительную оптимизацию информационного графа, осуществлять тестирование и отладку. Последующие преобразования промежуточных представлений в программы для реальных вычислительных ресурсов могут проводиться на уже отлаженном коде с применением формализованных подходов, что позволит повысить надёжность программ. Можно также проводить дополнительные оптимизации, например, повышение эффективности использования памяти. Но все это будет делаться после того, как основная проблема разработки, ручное создание корректно функционирующей программы, уже решена. Вместе с тем следует отметить, что в настоящий момент решена только первая часть задачи, связанная с

выполнением программ на эмуляторе событийной машины. Переход к преобразованию на реальные вычислительные системы планируется на следующем этапе работы.

Исследование выполняется при финансовой поддержке РФФИ в рамках научного проекта № 17-07-00288.

Литература

[1]. Matloff Norman. Parallel Computing for Data Science With Examples in R, C++ and CUDA. CRC Press. Taylor & Francis Group, 2016.
[2]. McKenney Paul E. Is Parallel Programming Hard, And, If So, What Can You Do About It? 2014.
[3]. Легалов А.И. Об управлении вычислениями в параллельных системах и языках программирования. Научный вестник НГТУ, № 3 (18), 2004, стр. 63-72.
[4]. Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010, 560 с.
[5]. Корнеев В.Д. Параллельное программирование в MPI. Новосибирск, Изд-во ИВМмМГ СО РАН, 2002, 215 с.
[6]. Эхтер Ш., Роберте Дж. Многоядерное программирование. СПб., Питер, 2010, 316 с.
[7]. Cheng John, Grossman Max, McKercher Ty. Professional CUDA ® C Programming. John Wiley & Sons, Inc., Indianapolis, Indiana, 2014.
[8]. Tay Raymond. OpenCL Parallel Programming Development Cookbook. Published by Packt Publishing Ltd., 2013
[9]. Lastovetsky Alexey L. Parallel Computing on Heterogeneous Networks. John Willey & Sons, 2003
[10]. Grid Computing – Technology and Applications, Widespread Coverage and New Horizons. Edited by Soha Maad. Published by InTech Janeza Trdine 9, 51000 Rijeka, Croatia, 2012.
[11]. Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa. Heterogeneous Computing with OpenCL. Advanced Micro Devices, Inc. Published by Elsevier Inc. 2013
[12]. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб., БХВПетербург, 2002, 608 с.
[13]. А.И. Легалов. Функциональный язык для создания архитектурно-независимых параллельных программ. Вычислительные технологии, № 1 (10), 2005, стр. 71-89
[14]. Backus J. Can programming be liberated from von Neuman style? A functional stile and its algebra of programs CACM. 1978, Vol. 21, N 8, pp. 613–641
[15]. Васильев В.С., Рыженко И.Н., Матковский И.В. Оптимизация параллельных списков функционально-потокового языка программирования «Пифагор». Системы. Методы. Технологии. № 3 (23), 2014, стр. 102-107.
[16]. Hoare, C. A. R. An axiomatic basis for computer programming. Communications of the ACM, 1969, Vol. 10, No. 12, pp. 576–585. DOI: 10.1145/363235.363259
[17]. Kropacheva M., Legalov A. Formal Verification of Programs in the Pifagor Language. Parallel Computing Technologies, 12th International Confernce PACT September-October, 2013. – St. Petersburg, Russia. Lecture Notes in Computer Science 7979, Springer, 2013, pp. 80-89

[18]. Ushakova M.S., Legalov A.I. Automation of Formal Verification of Programs in the Pifagor Language. Modeling and Analysis of Information Systems, 2015, 22(4) pp. 578-589. DOI:10.18255/1818-1015-2015-4-578-589

Support tools for creation and transformation of functional-dataflow parallel programs

A.I.Legalov <legalov@mail.ru>
V.S.Vasilyev <rrrFer@mail.ru>
I.V.Matkovskii <alpha900i@mail.ru>
M.S.Ushakova <ksv@akadem.ru>
Siberian Federal Univerisity,
660041, Russian Federation, Krasnoyarsk, 79 Svobodniy pr.

Abstract. In the article, a novel approach to the development, analysis and transformation of parallel programs is considered. A functional dataflow parallel programming language is used. It supports writing programs independently of any resource limitations. This allows to develop algorithms with maximal level of parallelism. Support tools for translation, execution, debugging, optimization and verification of functional dataflow parallel programs are developed. The translator transforms source code of a program to an intermediate representation, in which each function is defined by its dataflow graph. A dataflow graph describes data dependencies in the function and allows to construct the control graph, which defines the organization of computations by specifying the order of the operators execution. The optimization and verification of the program is carried out on their dataflow and control graphs. In order to execute a program the maximal parallelism is to be «compressed» according to particular computing systems' resource limitations. A computation process is considered as a juxtaposition of the control graph and the dataflow graph. It is possible to employ different control strategies by means of control graphs modification. The developed support tools allow to change computation control strategies adapting them to the peculiarities of a computational environment. The suggested tools provide generation of intermediate representation, which could be used as a basis for the following transformations of a program to the program for existed parallel computing systems architecture.

Keywords: architecture-independent parallel programming; functional-dataflow parallel programming; transformation of programs; software development tools, dataflow graph, control graph

DOI: 10.15514/ISPRAS-2017-29(5)-10

For citation: Legalov A.I., Vasilyev V.S., Matkovskii I.V., Ushakova M.S. Support tools for creation and transformation of functional-dataflow parallel programs. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 165-184 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-10

References

[1]. Matloff Norman. Parallel Computing for Data Science With Examples in R, C++ and CUDA. CRC Press. Taylor & Francis Group, 2016.
[2]. McKenney Paul E. Is Parallel Programming Hard, And, If So, What Can You Do About It? - 2014.
[3]. Legalov A.I. About computation control in parallel system and programming languages. Nauchiy Vestnik NGTU [Scientific Bulletin of NSTU], № 3 (18), 2004, pp. 63-72 (in Russian).
[4]. Karpov Y.G. Model Checking. Parallel and distributed program system verification. - SPb., BHV-Petersburg, 2010. - 560 p. (in Russian).
[5]. Korneev V.D. Parallel Programming in MPI. Novosibirsk, ICMMG SB RAS, 2002, 215 p. (in Russian).
[6]. Shameem Akhter, Jason Roberts. Multi-Core Programming. Intel Press, 2006.
[7]. Cheng John, Grossman Max, McKercher Ty. Professional CUDA ® C Programming. John Wiley & Sons, Inc., Indianapolis, Indiana, 2014
[8]. Tay Raymond. OpenCL Parallel Programming Development Cookbook. Published by Packt Publishing Ltd., 2013
[9]. Lastovetsky Alexey L. Parallel Computing on Heterogeneous Networks. John Willey & Sons, 2003
[10]. Grid Computing – Technology and Applications, Widespread Coverage and New Horizons. Edited by Soha Maad. Published by InTech Janeza Trdine 9, 51000 Rijeka, Croatia, 2012.
[11]. Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa. Heterogeneous Computing with OpenCL. Advanced Micro Devices, Inc. Published by Elsevier Inc., 2013
[12]. Voevodin V.V., Voevodin Vl.V. Parallel computing. SPb., BHV-Petersburg, 2002, 608 p. (in Russian).
[13]. A.I.Legalov Functional language for architecture-independent programming] Vichislitelnye Technologii [Computing technologies], № 1 (10), 2005, pp. 71-89 (in Russian).
[14]. Backus J. Can programming be liberated from von Neuman style? A functional stile and its algebra of programs CACM. 1978, Vol. 21, N 8, pp. 613–641.
[15]. Vasilyev V.S., Ryzhenko I.N., Matkovskii I.V. Parallel list optimization for function dataflow programming language “Pifagor”. Systemy. Methody. Technologii [Systems. Methods. Technologies]. № 3 (23), 2014, pp. 102-107. (in Russian).
[16]. Hoare, C. A. R. An axiomatic basis for computer programming. Communications of the ACM, 1969, Vol. 10, No. 12, pp. 576–585. DOI: 10.1145/363235.363259
[17]. Kropacheva M., Legalov A. Formal Verification of Programs in the Pifagor Language. Parallel Computing Technologies, 12th International Confernce PACT September-October, 2013. St. Petersburg, Russia. Lecture Notes in Computer Science 7979, Springer, 2013, pp. 80-89.
[18]. Ushakova M.S., Legalov A.I. Automation of Formal Verification of Programs in the Pifagor Language. Modeling and Analysis of Information Systems, 2015, 22(4), pp. 578-589. DOI:10.18255/1818-1015-2015-4-578-589

Объектно-ориентированная среда для разработки приложений планирования движения[1]

[1] *К.А. Казаков <kazakov@ispras.ru>*
[1,2] *В.А. Семенов <sem@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*

Аннотация. Обсуждаются принципы организации и функционирования инструментальной среды для программной реализации моделей, методов и приложений теории планирования движения. Среда предоставляет развитый набор готовых к использованию программных компонентов для автоматического построения бесконфликтных траекторий для робота, перемещаемого в статическом и динамическом трехмерном окружении. Организация среды в виде объектно-ориентированного каркаса обеспечивает развитие, адаптацию и гибкое конфигурирование разработанных программных компонентов в составе целевых приложений. Благодаря выделенным интерфейсам разного уровня и предусмотренным точкам расширения среда допускает интеграцию со сторонними прикладными системами.

Ключевые слова: планирование движения; поиск пути; определение столкновений; программная инженерия; объектно-ориентированное программирование.

DOI: 10.15514/ISPRAS-2017-29(5)-11

Для цитирования: Казаков К.А., Семенов В.А. Объектно-ориентированная среда для разработки приложений планирования движения. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 185-238. DOI: 10.15514/ISPRAS-2017-29(5)-11

1. Введение

Под планированием движения обычно понимается поиск бесконфликтного пути для перемещения твердого тела или кинематической конструкции в пространственно-трехмерной сцене. Искомый путь строится в конфигурационном пространстве объекта с учетом его степеней свободы и

[1] Работа поддержана РФФИ (грант 16-07-00606)

представляет собой непрерывную кривую, которая соединяет его начальное и конечное положения, исключает столкновения с препятствиями сцены и удовлетворяет всем установленным кинематическим и динамическим ограничениям [1, 2].

Задачи планирования движения возникают в разнообразных предметных областях, таких как машиностроение, робототехника, геоинформатика, транспорт, строительство, и часто связаны с автоматизацией и интеллектуализацией производственных процессов. Повышенный интерес к данным задачам обусловлен также и развитием современных технологий математического моделирования, компьютерной графики, виртуальной и дополненной реальности, допускающих конструктивное комплексное применение в составе целевых прикладных систем.

К подобным системам следует отнести системы автоматизированного проектирования, производства и инженерии (CAD/CAM/CAE), которые получили широкое распространение благодаря возможностям математического моделирования технологически сложных продуктов и процессов. Функционал данных систем охватывает многие математические и инженерные дисциплины, среди которых важное место занимают геометрическое моделирование и планирование движения.

Например, системы визуального моделирования индустриальных проектов, такие как Synchro Professional, Autodesk Navisworks, Trimble Vico, Rib iTWO, Bentley ConstructSim, реализуют функции навигации в трехмерном пространстве и времени, а также определения коллизий в динамических сценах. Благодаря данным функциям удается промоделировать ход проектных работ, выявить проблемы их координации в условиях ограниченных рабочих пространств и жестких временных сроков и, тем самым, снизить проектные риски и затраты. Не менее важными являются функции моделирования способов доставки материалов и оборудования к месту использования посредством различных грузоподъемных механизмов, а также моделирование процессов монтажа и сборки проектных конструкций. Реализация данных функций связана с решением задач планирования движения в сложном динамическом окружении, состоящем из сотен тысяч объектов с собственными геометрическими моделями и поведением [3, 4].

Другими интересными приложениями теории планирования движения являются задачи автоматической сборки или разборки машиностроительных изделий с целью верификации технологических процессов и предоставления интерактивных инструкций по производству и эксплуатации. Не менее актуальными являются задачи управления манипуляционными роботами, в которых требуется построить согласованные траектории звеньев манипуляторов.

Востребованными в последние годы являются средства навигации мобильных роботов, в частности, колесных транспортных механизмов с небольшим числом степеней свободы и неголономными связями. Колесные механизмы

удовлетворяют специальным геометрическим ограничениям, которые приводят к траекториям движения вдоль кривых Дьюбинса [5], Ридса-Шеппа [6], Балккома-Мейсона [7]. В некоторых постановках требуется прокладывать относительно протяженные траектории в псевдотрехмерном окружении (2.5D), характерном, например, для транспортных и архитектурно-строительных моделей. Информация о моделируемом окружении может быть заранее известна, либо поступать с сенсоров робота в режиме реального времени. В последнем случае возникает необходимость перманентного перепланирования движения объекта в ходе его перемещения [8, 9].

Содержательные примеры планирования движения предоставляют системы реалистичной компьютерной анимации неголономных систем со сложными кинематическими ограничениями. С помощью подобных систем успешно решаются задачи анимации человеческих персонажей, востребованные, например, в компьютерных играх и киноиндустрии, в том числе, с применением технологий виртуальной и дополненной реальности. Воссоздание реалистичной модели движения, учитывающей геометрические, кинематические и дифференциальные ограничения, представляется невозможным без использования соответствующих программных средств.

Перечисленные выше задачи планирования движения являются PSPACE-трудными, и поэтому их решение в индустриально значимых постановках высокой размерности представляет серьезную вычислительную проблему. Разработка необходимого математического и программного обеспечения с самого начала крайне сложна, требует широких компетенций и серьезных ресурсов. Использование же существующих программных средств, как коммерческих, так и с открытым исходным кодом сопряжено с рядом принципиальных ограничений и проблем. К числу наиболее значимых недостатков следует отнести:

— специализацию средств и невозможность их использования для решения задач общего вида, например, задач глобального планирования движения в сложном динамическом окружении;

— ограниченные инструментальные возможности для развития программных средств и реализации новых моделей, методов и приложений теории планирования движения;

— отсутствие виртуализации данных, приводящее к избыточному представлению модели окружения и быстрому исчерпанию оперативной памяти, необходимой для основных вычислений;

— наличие зависимостей от компонентов третьих сторон, часто приводящих к неопределенности и недостоверности результатов, а также порождающих дополнительные трудности при конфигурировании целевых приложений.

Поясним указанные недостатки на примере популярных библиотек планирования движения Motion Planning Kit (MPK) [10], OpenRave [11] и Open Motion Planning Library (OMPL) [12].

Функции данных библиотек, главным образом, предназначены для решения задач локального планирования движения и моделирования неголономных механических систем в режиме реального времени. Их математический арсенал в основном базируется на сэмплинг методах, которые демонстрируют высокую эффективность в приложениях управления промышленными роботами. Однако данные методы несостоятельны в случаях, когда требуется определить протяженные бесконфликтные траектории в трехмерном окружении со сложной топологией и динамическим поведением. Подобные задачи возникают, в частности, при визуальном моделировании архитектурно-строительных проектов и требуют эффективные средства для решения задач планирования движения в глобальных динамических постановках. При этом архитектура библиотек и особенности организации прикладных программных интерфейсов (API) препятствуют реализации в их составе новых методов и алгоритмов планирования движения для решения иных классов задач. Предусмотренные в библиотеках возможности модификации локальных алгоритмов качественно не меняют ситуацию.

Другим недостатком библиотек являются способы организации программных интерфейсов доступа к данным окружения, которые зачастую препятствуют их интеграции в целевые CAD/CAM/CAE системы. Данные системы обычно оперируют масштабными сценами, состоящими из сотен тысяч и миллионов объектов с индивидуальными геометрическими и динамическими характеристиками, и поэтому организация эффективного доступа к ним приобретает ключевое значение. В подобных системах используется свое внутреннее представление трехмерных моделей, которое диктуется их функционалом и прикладной спецификой. Например, трехмерные CAD системы поддерживают работу с аналитическими геометрическими кривыми и поверхностями, граничным и твердотельным представлениями тел. Системы визуального моделирования проектов ориентированы на работу с упрощенными полигональными моделями, предназначенными для быстрой растеризации сцен и локализации пространственных коллизий. CAM и PLM системы в большей степени оперируют кинематическими моделями, предназначенными для моделирования технологических операций.

В силу указанных причин интерфейс доступа к данным окружения должен учитывать альтернативные представления трехмерных моделей, а также допускать возможность непосредственного использования функций целевых систем при определении столкновений и анализе согласованности объектных конфигураций. Данные требования приводят к необходимости иметь точки расширения программной среды (hotspots), которые бы позволили настроить, сконфигурировать или доработать соответствующие компоненты для поддержки альтернативных моделей и реализации операций с ними.

В самом деле, жесткая привязка интерфейса к конкретной модели целевой системы, во-первых, требует написания большого объема адаптирующего кода, а, во-вторых, приводит к нерациональному расходованию оперативной

памяти. С другой стороны, полная независимость от геометрической и кинематической моделей также порождает трудности при интеграции в целевую систему. Например, при использовании библиотеки OMPL, реализующей сэмплинг методы, на разработчика ложится ответственность за предоставление релевантных метрик многомерных конфигурационных пространств, в которых решаются пользовательские задачи. Кроме того, разработчиком должны быть реализованы и предоставлены эффективные средства определения столкновений и анализа согласованности конфигураций, что также является серьезной проблемой с учетом высокой размерности типовых задач.

Наконец, зависимость от компонентов третьих сторон и их сильное зацепление по данным и управлению также является недостатком существующих библиотек планирования движения. С одной стороны, использование детально специфицированных и тщательно отлаженных сторонних компонентов является хорошей практикой при создании сложных программных систем. Однако она оправдывает себя, если только компоненты реализуют независимые функции, оперирующие с единым представлением данных. В противном случае возникает необходимость их перманентной конвертации и согласования. Кроме того, использование компонентов разных версий часто порождает конфликты, которые затрудняют сборку, тестирование и дальнейшее сопровождение целевых систем.

Например, инструментальная среда MoveIt, входящая в состав ROS, построена на основе сразу всех упомянутых библиотек планирования движения. Причем OMPL предлагается в качестве основного средства построения бесконфликтных траекторий, OpenRave используется как средство решения обратной кинематической задачи, а MPK применяется для быстрого разрешения одиночных запросов на основе двухнаправленных маршрутных сетей с отложенным определением столкновений на основе SBL алгоритма. Очевидно, что систематическое построение приложений планирования движения с помощью данной инструментальной среды проблематично, а полученные результаты могут быть недостоверными.

Множественные зависимости среды от ядра ROS, библиотеки определения столкновений FCL, а также FLANN, MongoDB и boost создают дополнительные трудности при ее практическом использовании.

2. Назначение и общая организация среды

Разработанная инструментальная среда предназначена для программной реализации моделей, методов и приложений теории планирования движения. Среда предоставляет развитый набор готовых к использованию программных компонентов для автоматического построения бесконфликтных траекторий в статическом и динамическом трехмерном окружении для робота, имеющего произвольное число степеней свободы и функционирующего как на основе априорных знаний о сцене, так и в неизвестном окружении в режиме

реального времени. Среда реализуется не только как специализированная библиотека для решения типовых задач планирования движения, но и как библиотека программных компонентов для построения приложений в разных предметных областях. Именно данная возможность принципиально отличает ее от упомянутых выше решений. При этом декларируемая универсальность среды не препятствует эффективности разрабатываемых целевых приложений, поскольку предусматривает возможности гибкого конфигурирования программных компонентов, их развития, адаптации и настройки с учетом специфики решаемых прикладных задач.

Среда спроектирована и реализована в виде каркаса (архитектурного шаблона) на основе технологий объектно-ориентированного и компонентно-ориентированного программирования. Данные технологии широко применяются при создании сложных программных систем с развиваемым функционалом и при разработке серий программных приложений. Ожидается, что благодаря каркасной организации среда позволит существенно сократить сроки и затраты на создание приложений и при этом обеспечит их надежность и эффективность, необходимые для решения практических вычислительно сложных задач. Среда программно реализована на языке Си++, поэтому в дальнейшем ее компоненты описываются в терминах абстрактных и конкретных классов.

Проектированию каркаса предшествовали исследовательские работы, связанные с систематизацией и концептуализацией задач и методов теории планирования движения. В ранее опубликованных авторами работах [2, 13] был проведен анализ современных методов планирования движения, который позволил выделить основные постановки задач, ключевые подходы к их решению, перспективные семейства методов и эффективные алгоритмы. В дальнейшем мы выделяем две основные постановки задач и связанные с ними вычислительные стратегии, а именно: локальное и глобальное планирование движения.

Глобальное планирование предполагает априорное знание о моделируемом окружении и подразумевает предварительный анализ трехмерной сцены с последующим разрешением множественных запросов поиска пути. Данная постановка реализуется посредством предварительного построения маршрутного графа, отражающего топологию моделируемой сцены. Маршрутный топологический граф агрегирует множество локальных маршрутов в рабочем пространстве сцены, оценки их стоимости, а также дополнительную информацию, которая может быть полезна для разрешения последующих запросов на основе эвристических правил. Глобальное планирование на основе маршрутных сетей предполагает быстрый поиск перспективных маршрутов движения, которые затем могут быть проверифицированы и при необходимости скорректированы с помощью алгоритмов локального планирования.

Локальное планирование подразумевает разрешение одиночных запросов поиска бесконфликтных траекторий для конкретного объекта (твердого тела или кинематической системы), учитывая его конкретное геометрическое представление, а также наложенные кинематические и динамические ограничения. Для программной реализации средств локального планирования было выбрано семейство сэмплинг методов, зарекомендовавших себя при решении задач в сложных многомерных конфигурационных пространствах.

Для решения широких классов задач в локальной и глобальной постановке с использованием альтернативных методов и алгоритмов был проведен объектный анализ теории планирования движения. Он позволил выделить ее ключевые абстракции, их взаимосвязи, и тем самым, определить необходимый состав, структуру и возможные точки расширения среды. В ее общей структуре удобно выделить группы классов для представления моделируемого трехмерного окружения (пакет WorkSpace), для решения вспомогательных задач в конфигурационных пространствах объектов (пакет ConfigurationSpace) и для работы с графами и маршрутными сетями (пакет DiscreteSpace).

Классы пакета WorkSpace определяют интерфейсы доступа к объектам моделируемого окружения, которое включает в себя и перемещаемый объект и наложенные на него кинетические ограничения. В этом же пакете реализуется подсистема классов для определения столкновений с использованием структур пространственно-временной индексации. Последние применяются для быстрой локализации столкновений и ускорения работы основных алгоритмов. Поскольку некоторые CAD/CAM/CAE системы имеют собственные средства определения столкновений, оперирующие непосредственно с внутренним представлением данных окружения, классы пакета определяют единые интерфейсы для формирования и исполнения соответствующих запросов. Благодаря интерфейсам разработчики целевых приложений могут предоставить альтернативные реализации средств определения столкновений.

Пакет ConfigurationSpace составляют классы, предназначенные для решения задач в конфигурационных пространствах объектов. Данные классы позволяют задать множество допустимых конфигураций для объекта планирования, сформировать и исполнить запросы поиска бесконфликтных траекторий движения. Специальные классы решателей реализуют различные семейства сэмплинг методов для планирования движения в локальной постановке.

Пакет классов DiscreteSpace предназначен для представления графов и решения задач поиска путей в них. Обобщенные реализации классов позволяют использовать их в качестве базовых при специализации графов в виде быстрорастущих деревьев и маршрутных сетей. Заметим, что последние могут строиться как в трехмерном пространстве окружения, так и в конфигурационном пространстве объекта планирования. Тем самым,

обеспечивается возможность многоцелевого использования графовых структур данных и алгоритмов поиска без необходимости дублирования кода.

Пакеты используют единый набор вспомогательных типов данных, предназначенных в основном для работы с примитивами компьютерной графики и реализации векторно-матричных операций.

Наконец, среда включает в себя подсистему классов для глобального планирования движения. Подсистема реализует общую вычислительную стратегию, заключающуюся в последовательной редукции исходной вычислительно сложной задачи планирования движения к типовым, относительно простым задачам поиска в графах. Наши предыдущие исследования показали эффективность и перспективность подобной стратегии [3, 13, 14]. Примечательно, что упомянутые выше сэмплинг методы, маршрутные сети и алгоритмы теории графов являются элементами данной общей стратегии, а ее программная реализация базируется на согласованном использовании рабочего, конфигурационного и дискретного представлений, поддерживаемых соответствующими классами среды. При изменениях в моделируемом окружении, данные представления инкрементально обновляются с возможностью оперативного разрешения последующих запросов маршрутизации. Подсистема глобального планирования встраивается в цикл работы целевой системы и, работая в фоновом режиме, формирует и обрабатывает очередь запросов, связанных с изменениями в моделируемом окружении.

Классы среды тесно связаны друг с другом и составляют единый инструментарий для программной реализации приложений теории планирования движения. Предусмотренные точки расширения в виде абстрактных классов позволяют адаптировать существующие компоненты и реализовать новые с учетом специфики прикладных задач и возможностей применения эффективных методов и алгоритмов.

3. Пакет классов WorkSpace

Рассмотрим более подробно организацию пакета классов WorkSpace. Моделируемое трехмерное окружение представляет собой множество разнородных геометрических объектов. С каждым объектом связан уникальный идентификатор, геометрическая модель и локальная система координат, определяющая его положение в окружении. Предполагается, что с каждым объектом может быть ассоциирована альтернативная геометрическая модель, применяемая, например, для быстрой локализации столкновений или растеризации сцены. В дальнейшем рассматривается два вида объектов: простые твердые тела и кинематические системы взаимосвязанных твердотельных звеньев.

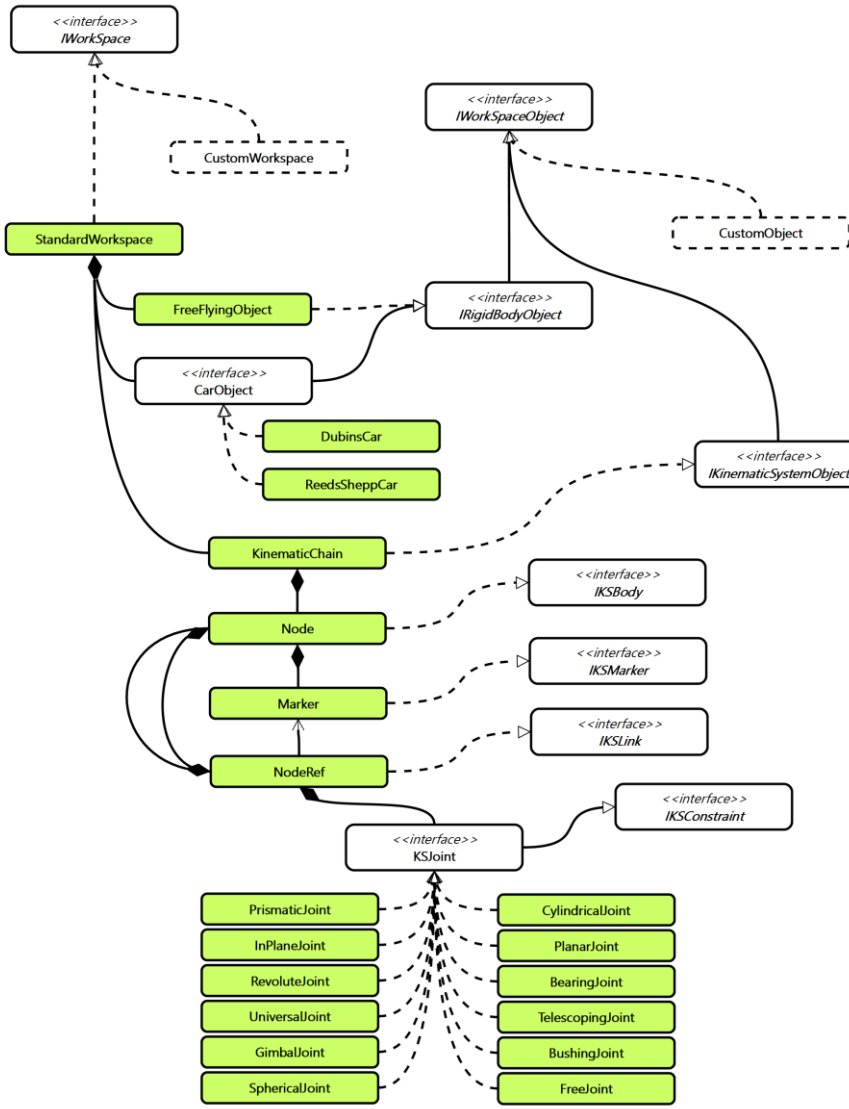


Рис. 1. Диаграмма классов трехмерного окружения.
Fig. 1. 3D WorkSpace class diagram.

Заметим, что в CAD/CAM/CAE приложениях часто используются, так называемые, сложные или составные объекты, которые представляются композицией дочерних объектов и организованы в самостоятельную иерархию. Сложные объекты могут, например, определять слои, соответствующие поэтажному плану здания или составу его конструктивных элементов. В машиностроительных отраслях сложные объекты часто используются для определения сборок деталей. Для обсуждаемых задач планирования движения композиционная структура объектов не важна и далее во внимание не принимается. Последнее не исключает, что объекты окружения могут иметь сложное геометрическое представление.

Поскольку среда предназначена для разработки приложений планирования движения, в которых может использоваться свое собственное представление данных окружения, классы пакета WorkSpace определяют лишь общий интерфейс доступа к ним. Ответственность за реализацию интерфейса целиком ложится на разработчика приложения. Интерфейс позволяет избежать привязки к прикладным типам данных, при этом предоставляя необходимые общие методы обхода объектов, получения их индивидуальных геометрических и поведенческих моделей, а также текущего состояния. Обсудим вопросы организации интерфейса более подробно.

3.1 Доступ к объектам окружения

Абстрактные классы ***IWorkSpace*** и ***IWorkSpaceObject*** определяют интерфейс доступа к моделируемому окружению.

Интерфейс ***IWorkSpace*** включает в себя следующие виртуальные методы:

- ***getModellingBounds()-> aabb_t***
  возвращает границы моделируемого окружения в виде AABB (Axis Aligned Bounding Box) параллелепипеда. Возвращаемый параллелепипед может охватывать как всю сцену, так и любую ее подобласть, внутри которой необходимо построить маршрутную сеть.
- ***createObjectIterator()-> abstract_iterator<IWorkSpaceObject>***
  создает итератор для однонаправленного обхода разнородных геометрических объектов с общим интерфейсом ***IWorkSpaceObject***.
- ***getObject(uid_t*** *objectID* ***)-> IWorkSpaceObject***
  предоставляет доступ к объекту окружения по заданному идентификатору.
- ***getCurrentState(uid_t*** *objectID* ***)-> IState***
  предоставляет доступ к текущему состоянию объекта по заданному идентификатору.
- ***resolveState(uid_t*** *objectID*, ***IState*** *objectState*, ***resolve_state_callback_t*** *resolveStateCallback* ***)-> bool***

объект с заданным идентификатором устанавливает в положение, соответствующее приписанному состоянию в конфигурационном пространстве. В качестве последнего параметра метода выступает заданная функция применения рассчитанных трансформаций к индивидуальным объектам ***resolve_state_callback_t(uid_t*** *bodyID****, mat4_t*** *bodyTransform* ***)***. В случае задания в качестве объекта кинематической системы метод реализует решение прямой кинематической задачи.

Абстрактный класс ***IWorkSpaceObject*** определяет следующий интерфейс доступа к индивидуальному объекту окружения:

- ***getID()->uid_t***
  получить уникальный идентификатор объекта
- ***createCollisionShape()->ICollisionShape***
  создать альтернативную геометрическую модель объекта для определения столкновений
- ***createStateSpace(aabb_t*** *modellingBounds* ***)->IStateSpace***
  сформировать множество допустимых конфигураций объекта

Интерфейс предусматривает две ключевые функции, а именно: конструирование альтернативной геометрической модели объекта, предназначенной, прежде всего, для эффективного определения столкновений, а также формирование множества допустимых конфигураций объекта для анализа его согласованных состояний и бесконфликтных переходов между ними.

Специальные классы ***IRigidBodyObject*** и ***IKinematicSystemObject*** расширяют базовый интерфейс ***IWorkSpaceObject***, определяя операции для твердотельных объектов и кинематических систем. В ряде случаев необходимо различать подтипы сконструируемых объектов на базовом уровне ***IWorkSpaceObject***, поэтому в интерфейсе предусмотрены соответствующие методы для получения подтипов объектов и приведения к ним объектных ссылок.

Абстрактный класс ***IKinematicSystemObject*** служит для доступа к внутренней структуре кинематической системы, предоставляя методы получения отдельных твердотельных звеньев и кинематических сочленений:

- ***getNumberOfBodies()-> int***
  получить число звеньев
- ***getBody(kuid_t*** *bodyID* ***)-> IKSBody***
  найти звено по идентификатору
- ***createBodyIterator()-> abstract_iterator<IKSBody>***
  создать итератор для обхода звеньев
- ***getNumberOfLinks()-> int***
  получить число сочленений
- ***getLink(kuid_t*** *linkID* ***)-> IKSLink***

найти сочленение по идентификатору

- ***createLinkIterator()-> abstract_iterator<IKSLink>***
  создать итератор для обхода сочленений

Приведенные методы позволяют выполнить обход звеньев и сочленений кинематической системы и получить необходимый доступ к их параметрам. Такая организация класса преследует сразу несколько целей. Во-первых, могу быть заданы целевые положения звеньев системы, необходимые для формирования запросов планирования движения. Во-вторых, упрощается и унифицируется процедура формирования множеств допустимых конфигураций для сложных кинематических систем. В-третьих, становится возможным идентифицировать конфигурации, приводящие к самопересечениям кинематических систем. В-четвертых, благодаря выделенным абстракциям звена и сочленения обеспечивается возможность развития среды в направлении поддержки средств физического моделирования.

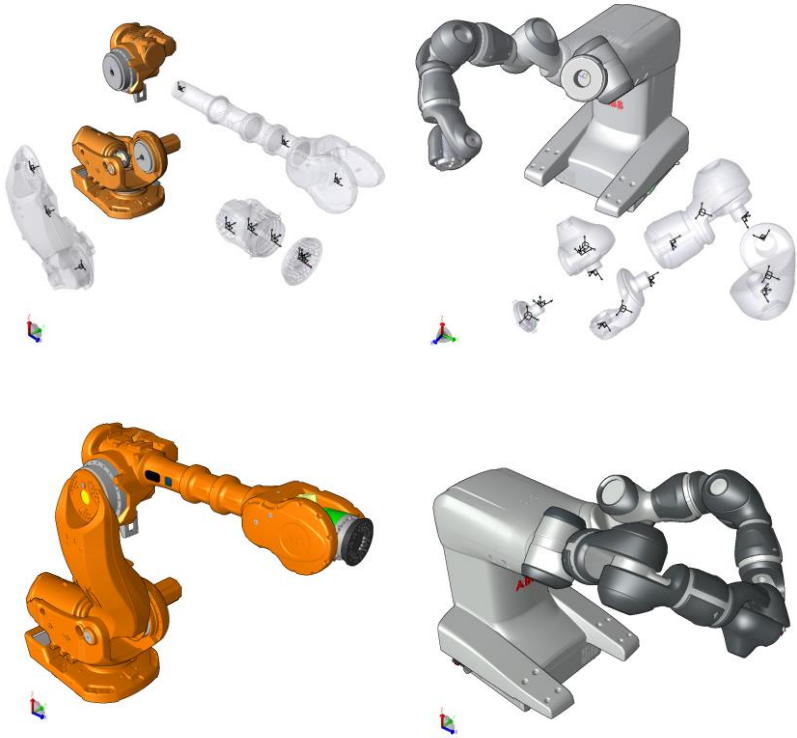


Рис. 2. Примеры моделей манипуляционных роботов: IRB7600 и IRB14000
Fig. 2. Example manipulation robot models: IRB7600 и IRB14000.

Звено кинематической системы предоставлено в среде абстрактным классом ***IKSBody***. Он позволяет получить основные параметры звена, такие как локальная система координат, масса, центр масс и матрица тензора инерции. Кроме того, он предоставляет возможность обхода точек сочленения звена, представленных абстрактным классом ***IKSMarker***. Точка сочленения (или маркер) является уникально идентифицируемым объектом, атрибутом которой является локальная система координат, заданная в базисе звена.

Интерфейс класса ***IKSBody*** определяет следующие виртуальные методы:

- ***getID()-> uid_t***
  получить уникальный идентификатор
- ***getFrame()-> frame_t***
  получить локальную систему координат
- ***getMass()-> float***
  получить значение массы тела
- ***getCenterOfMass()-> vec3_t***
  получить координаты центра масс
- ***getInertiaTensor()-> mat3_t***
  получить матрицу тензора инерции
- ***createCollisionShape()-> ICollisionShape***
  построить альтернативную геометрическую модель
- ***getMarkerCount()-> int***
  получить число точек сочленения
- ***getMarker(uid_t*** *markerID* ***)-> IKSMarker***
  найти точку сочленения по идентификатору
- ***createMarkerIterator()-> abstract_iterator<IKSMarker>***
  создать итератор для обхода точек сочленения

Для представления кинематических связей используется абстрактный класс ***IKSLink***, через интерфейс которого можно получить доступ к звеньям и точкам сочленения кинематической пары. Одна из точек сочленения рассматривается в качестве ведущей (*Master*), а другая — ведомой (*Slave*).

Интерфейс ***IKSLink*** включает в себя следующий набор методов:

- ***getID()-> uid_t***
  получить уникальный идентификатор
- ***getMasterBodyID()-> uid_t***
  получить идентификатор ведущего звена
- ***getMasterBodyMarkerID()-> uid_t***
  получить идентификатор точки сочленения с ведущим звеном
- ***getSlaveBodyID()-> uid_t***
  получить идентификатор ведомого звена

- ***getSlaveBodyMarkerID()-> uid_t***
  получить идентификатор точки сочленения с ведомым звеном
- ***getConstraint()-> IKSConstraint***
  получить ограничения кинематической связи

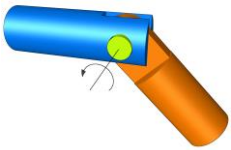
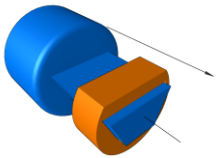
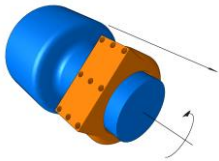
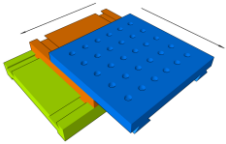
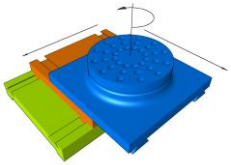
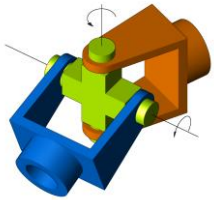
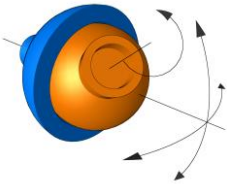
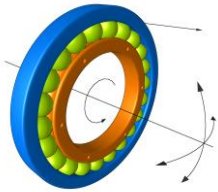
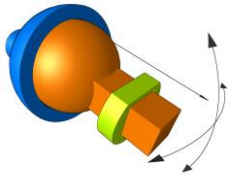
Название класса / Число степеней свободы		Название класса / Число степеней свободы		Название класса / Число степеней свободы	
RevoluteJoint	1	***PrismaticJoint***	1	***CylindricalJoint***	2
					
InPlaneJoint	2	***PlanarJoint***	3	***UniversalJoint***	3
					
SpherialJoint	3	***BearingJoint***	4	***TelescopingJoint***	4
					

Рис. 3. Некоторые классы кинематических ограничений
Fig. 3. Some classes of kinematic constraints.

Согласованное относительное положение звеньев кинематической пары определяется наложенными алгебраическими ограничениями. Для их получения можно воспользоваться виртуальным методом ***getConstraint()***, возвращающим ссылку на объект типа ***IKSConstraint***. Данный абстрактный класс предусматривает методы, необходимые для определения множества

допустимых конфигураций кинематической пары, а также для проверки конфигурации на согласованность с наложенными ограничениями. На основе ограничений, полученных для отдельных кинематических пар, можно сформировать множество допустимых конфигураций для всей кинематической системы. Данная функция естественным образом реализуется на уровне базового класса ***IKinematicSystemObject***.

В состав среды включены конкретные классы ограничений, служащие для задания подвижных соединений с различным количеством степеней свободы и различными комбинациями поступательного и вращательного движения. Данные классы реализуются как наследники базового класса ***IKSConstraint*** (рис. 3).

3.2 Подсистема для определения столкновений

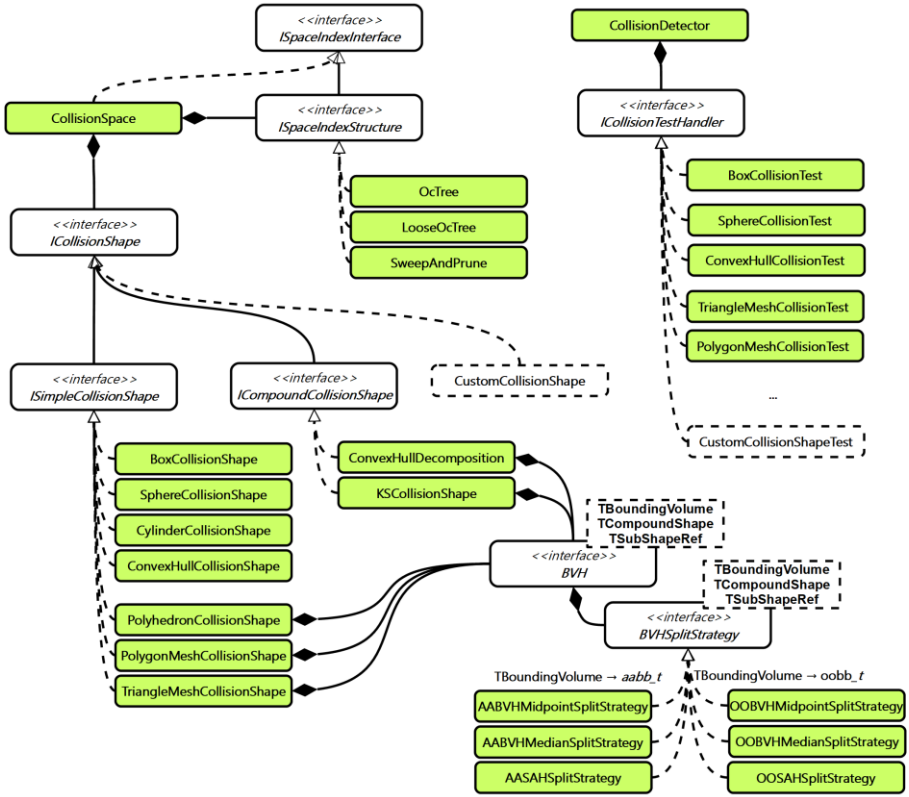


Рис. 4. Диаграмма классов подсистемы определения столкновений
Fig. 4. Collision detection subsystem class diagram.

Подсистема для определения столкновений реализуется как часть среды и представлена классами для задания геометрических моделей типа ***ICollisionShape***, классами представления окружения как композиций объектов и их геометрических моделей — ***StandardWorkSpace*** и ***CollisionSpace*** соответственно, а также классом реализации методов определения столкновения ***CollisionDetector***.

К числу первых относится уже упоминаемый абстрактный класс геометрических моделей ***ICollisionShape***, а также наследуемые от него конкретные классы геометрических примитивов ***BoxCollisionShape***, ***SphereCollisionShape***, ***CylinderCollisionShape***, класс многогранников ***PolyhedronCollisionShape***, классы представления полигональных и триангулированных сеток ***PolygonMeshCollisionShape*** и ***TriangleMeshCollisionShape***, класс выпуклых полигональных оболочек ***ConvexHullCollisionShape***. Обсудим вопросы организации и функционирования подсистемы определения столкновений более подробно.

3.2.1 Геометрические модели

Абстрактный класс ***ICollisionShape*** предназначен для определения интерфейса доступа к альтернативному геометрическому представлению объекта, которое следует использовать для быстрого определения столкновений в ходе исполнения запросов планирования движения. Заметим, что данное представление не обязано совпадать с оригинальной геометрией объекта. Поскольку идентификация столкновений является вычислительно затратной операцией, а методы планирования движения используют ее в качестве базовой, в ряде случаев целесообразно упростить геометрическую модель объекта. Например, она может быть заменена полигональным граничным представлением с меньшим числом граней или примитивными ограничивающими объемами, для которых известны эффективные алгоритмы пересечения.

Интерфейс класса ***ICollisionShape*** представлен следующими виртуальными методами:

- ***getObjectID()->uid_t***
  получить идентификатор исходного объекта (***IWorkSpaceObject***)
- ***getBoundingBox()->aabb_t***
  получить ограничивающий AABB параллелепипед
- ***setTransform(uid_t bodyID, mat4_t bodyTransform)***
  применить трансформацию к объекту с заданным идентификатором
- ***getMargin()->float***
  получить оценку точности геометрического представления (минимальная глубина проникновения, при которой следует идентифицировать пересечение объектов)

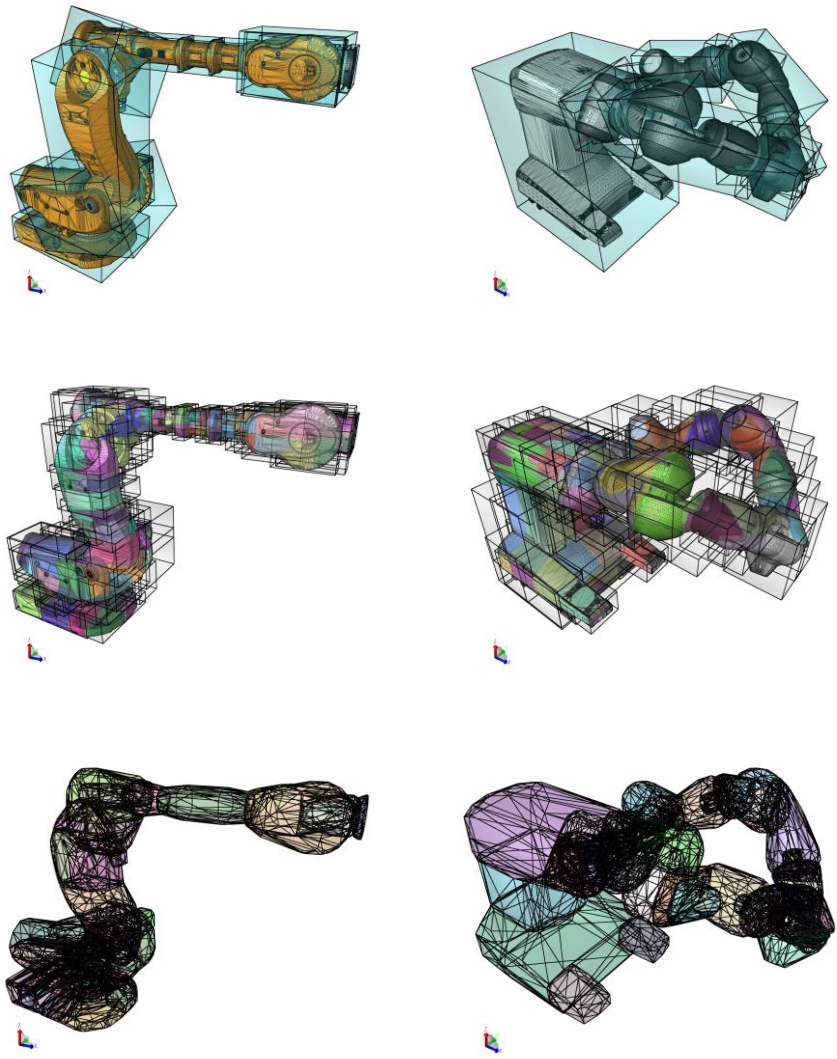


Рис. 5. Альтернативные геометрические модели составных трехмерных объектов: OOBB-дерево, AABB-дерево, декомпозиционное представление выпуклыми оболочками
Fig. 5. Alternative geometry models of compound 3D object: OOBB-tree, AABB-tree, convex hull decomposition.

Поскольку объект окружения и его геометрические представления взаимосвязаны, при обнаружении пересечений могут быть указаны конфликтующие объекты, а также уточнен характер их пересечений.

Среда предоставляет набор готовых к использованию классов геометрических моделей, которые могут быть выбраны разработчиком в качестве альтернативных представлений объектов окружения. Во внимание могут приниматься особенности целевого приложения, способы представления трехмерных данных, требуемая точность локализации столкновений, а также имеющиеся вычислительные ресурсы.

Классы геометрических моделей организованы в виде единой иерархии, наследуемой от базового класса ***ICollisionShape***. Абстрактные классы простых и составных моделей ***ISimpleCollisionShape*** и ***ICompoundCollisionShape*** являются специализациями базового и уточняют его методы.

Простые модели представлены конкретными классами геометрических примитивов ***BoxCollisionShape***, ***SphereCollisionShape***, ***CylinderCollisionShape***, классами многогранников ***PolyhedronCollisionShape***, ***ConvexHullCollisionShape*** и классами сеток ***TriangleMeshCollisionShape***, ***PolygonMeshCollisionShape***. Как правило, в качестве альтернативных геометрических представлений используются полигональные сетки. Гранями таких сеток обычно являются треугольники, четырехугольники или другие простые многоугольники, для которых операции взаимного пересечения в пространстве реализуются относительно просто. Реализации упомянутых классов многогранников и полигональных сеток рассчитаны на более общий случай и допускают задание граней в виде невыпуклых многоугольников и многоугольников с дырками.

Составные геометрические модели представлены классами ***KSCollisionShape*** и ***ConvexHullDecomposition***. Первый реализует составную геометрическую модель кинематической конструкции, второй — декомпозиционное представление произвольного многогранника на основе выпуклых оболочек [15].

Геометрическая модель всего окружения реализуется конкретным классом ***CollisionSpace***. Данный класс позволяет выполнить обход моделей всех объектов, а также синхронизовать их с текущим представлением окружения как результат реакции на происходящие в нем события.

Обновления геометрических моделей в классе ***CollisionSpace*** реализуются с помощью следующих методов:

- ***rebuild()***
  выполняет полное обновление моделей для всех объектов окружения
- ***rebuildObject(uid_t*** *objectID* ***)***
  обновляет модель заданного объекта
- ***updateObjectState(uid_t*** *objectID* ***)***
  устанавливает модель объекта в заданное положение

- ***removeObject(uid_t** objectID **)***
  удаляет заданный объект

3.2.2 Определение столкновений

Определение столкновений в сложном масштабном окружении представляет собой серьезную проблему. Вычислительная сложность определения столкновений может быть существенно уменьшена при использовании пространственных индексов. Основное назначение индексов — локализация потенциальных столкновений за относительно небольшое время на, так называемой, широкой фазе. Выявленные возможные столкновения затем анализируются с использованием точных и вычислительно сложных алгоритмов на узкой фазе. Тем самым, минимизируются затраты на определение столкновений за счет дешевых негативных тестов на пересечения, а точные алгоритмы применяются избирательно только для выявленных пар объектов, допускающих пересечения.

Для реализации подобной стратегии подсистема поддерживает два вида пространственных индексов. Первый использует пространственную декомпозицию всего моделируемого окружения на основе регулярных сеток и позволяет выделить потенциально пересекающиеся группы объектов, которые принадлежат одним пространственным ячейкам [16]. Другой вид индексов — иерархии ограничивающих объемов (BVH) (рис. 5), которые строятся индивидуально для каждой геометрической модели объекта и позволяют выделить пары объектов, элементы которых допускают пересечения. В качестве ограничивающих объемов обычно применяют AABB и OOBB параллелепипеды [17,18]. Рассмотрим вопросы реализации и применения индексов более подробно.

Индексы пространственной декомпозиции строятся для модели всего окружения ***CollisionSpace*** и реализуются классами с общим интерфейсом ***ISpaceIndexStructure***. Данный интерфейс определяет методы построения, инкрементального обновления и применения индекса при поиске ближайших соседей и локализации столкновений независимо от алгоритмических и программных особенностей его реализации. Поведение индекса делегируется соответствующему объекту ***CollisionSpace***, который поддерживает его в состоянии согласованном с геометрической моделью окружения. При изменениях окружения индекс автоматически перестраивается.

Запросы поиска столкновений вынесены в отдельный абстрактный интерфейс ***ISpaceIndexSearchInterface***, который наследуют оба класса ***ISpaceIndexStructure*** и ***CollisionSpace***. Запросы представлены следующими виртуальными методами:

- ***intersectionTest(ICollisionShape** object**, function<bool(ICollisionShape** neighbour**)>** callback **)-> bool***
  устанавливает факт пересечения заданного объекта с окружением

- ***clearanceTest(ICollisionShape*** *object****, function<float(ICollisionShape*** *neighbour****)>*** *callback* ***) -> float***

  осуществляет поиск кратчайшего расстояния между заданным объектом и окружением

Метод ***intersectionTest*** осуществляет поиск объектов, которые потенциально пересекаются с заданным. Результаты поиска возвращаются через функцию обратного вызова, в которой выполняется точное пересечение геометрических моделей. Это позволяет вызывающему коду прервать операцию при обнаружении первого пересечения и повысить эффективность исполнения запросов. Аналогичным образом метод ***clearanceTest*** принимает в качестве входного параметра функцию точного определения расстояния между объектами. Промежуточные результаты используются для динамического уменьшения радиуса поиска и исключения вызовов вычислительно сложной операции для объектов, расположенных на значительном удалении от заданного и не влияющих на конечный результат.

Разработчику предоставляется возможность использовать альтернативные реализации пространственных индексов, основанных на октодеревьях (***OcTree***), октодеревьях с релаксацией границ (***LooseOcTree***) [19] и сортированных списках ограничивающих объемов (***SeepAndPrune***) [20]. При необходимости разработчик может реализовать собственные методы пространственной локализации объектов и определения стокновений с учетом особенностей решаемых прикладных задач.

За реализацию узкой фазы определения столкновений отвечает класс ***CollisionDetector***. Он обеспечивает регистрацию обработчиков столкновений для каждой пары геометрических моделей, используемых в представлении объектов окружения и имеющих тип ***ICollisionShape***. Использование для этого хэш-таблицы с ключом в виде пары идентификаторов геометрических моделей позволяет ускорить поиск и применение обработчиков при анализе окружения, состоящего из разнотипных объектов. Сами обработчики наследуют общий интерфейс ***ICollisionTestHandler***:

- ***getKey()-> unique_pair<type_index,type_index>***

  получить уникальный ключ обработчика
- ***intersect(ICollisionShape*** *firstObject****, ICollisionShape*** *secondObject* ***) -> bool***

  выполнить проверку пары объектов на пересечение
- ***distance(ICollisionShape*** *firstObject****, ICollisionShape*** *secondObject* ***) -> float***

  найти расстояние между объектами

Обобщенная реализация ***CollisionDetector*** позволяет разработчику поддерживать в подсистеме определения столкновений собственные геометрические модели и регистрировать для них соответствующие функции пересечения и определения расстояния. Реализация данных функций для

определенных типов геометрических объектов имеет свои особенности. Например, классы многогранников, полигональных и треугольных сеток помимо внутреннего представления агрегируют вспомогательный пространственный индекс в виде иерархии ограничивающих объектов, реализуемой шаблонным классом ***TBoundingVolumeHierarchy***. Иерархии ограничивающих объемов строятся единожды при конструировании объектов. При изменении положения объектов нет необходимости перестраивать иерархии заново, поскольку при локализации столкновений соответствующие трансформации могут применяться непосредственно к ограничивающим объемам. Для пересечения выпуклых многогранников, а также оценки возможной глубины проникновения применяется алгоритм расширенных политопов [21], являющийся развитием известного алгоритма Гилберта-Джонсона-Керти [22].

3.3 Моделируемое окружение

Конкретный класс ***StandardWorkSpace*** предоставляет типовую реализацию моделируемого окружения с использованием таких объектов как твердое тело, свободно движущееся в пространстве (***FreeFlyingObject***), машина Дьюбинса [5] (***DubinsCar***) и Ридса-Шеппа [6] (***ReedsSheppCar***), кинематическая цепь (***KinematicChain***). Используемая в классе ***StandardWorkSpace*** фабрика объектов позволяет разработчику добавлять реализации новых типов объектов и, тем самым, расширять возможные постановки задач планирования движения. Поскольку класс представления моделируемого окружения наследуется от ***IWorkSpace***, все реализуемые средой функции, включая методы планирования движения, распространяются и на новые типы объектов.

4. Пакет классов DiscreteSpace

В основе большинства методов планирования движения лежит идея редукции исходной вычислительно сложной задачи к задаче поиска маршрута в графе, разрешимой известными алгоритмами Дейкстры или A* за приемлемое время [2,23]. Вершинам графа ставятся в соответствие точки в рабочем пространстве окружения или в конфигурационном пространстве объекта, а ребрам — бесконфликтные переходы между точками. Элементам графа могут быть приписаны дополнительные данные о стоимости переходов, расстояниях до препятствий окружения, об успешных или неуспешных прецедентах перемещения некоторых объектов и т.п.

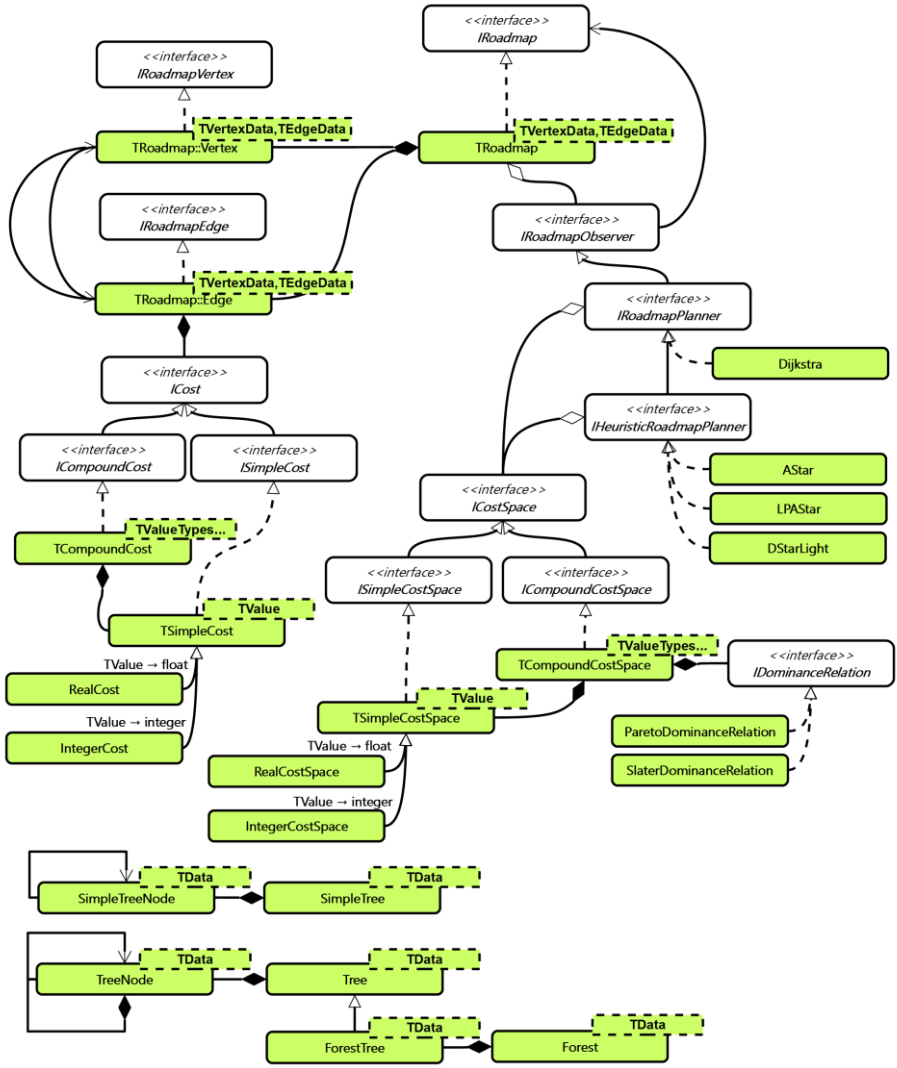


Рис. 6. Диаграмма классов пакета DiscreteSpace

Fig. 6. DiscreteSpace class diagram.

Основным назначением пакета классов DiscreteSpace является задание условий и решение задач теории графии. Обобщенные реализации классов позволяют использовать их в качестве базовых при специализации графов в виде быстрорастущих деревьев и маршрутных сетей. Тем самым обеспечивается возможность многоцелевого использования пакета для представления графов и программной реализации алгоритмов поиска путей в них. В частности, среда предоставляет готовые к использованию классы для представления таких математических объектов как дерево, лес и маршрутная сеть. Обсудим их реализации в виде шаблонных классов более подробно.

4.1 Деревья

Деревья представлены шаблонным классом ***Tree<TData>*** с параметризуемым типом узлов. Класс реализует базовые операции, необходимые для построения и модификации деревьев:

- ***createNode(Node*** *parent* ***)->Node***
  создать новый узел дерева
- ***deleteNode(Node*** *node* ***)***
  удалить узел дерева
- ***moveNode(Node*** *node****, Node*** *newParent* ***)***
  перенести ветвь дерева
- ***rotateTo(Node*** *node* ***)***
  развернуть дерево относительно узла node
- ***randomNode()->Node***
  выбрать случайный узел дерева

Приведенные операции удаления узлов и переноса ветвей необходимы, в частности, для построения оптимальных путей RRT* алгоритмом [24].

В других случаях, например, при реализации RRT и RRT-connect алгоритмов [25,26], операции трансформации не требуются и структура представления дерева может быть существенно упрощена за счет хранения однонаправленных ассоциаций узлов на родителей и исключения обратных ассоциаций. Подобный компактный способ представления дерева с необходимым набором базовых операций реализуется в классе ***SimpleTree<TData>***.

4.2 Лес деревьев

Для реализации некоторых алгоритмов сэмплирования требуются операции над множеством деревьев. Для этих целей в среде предусмотрен шаблонный класс ***Forest<TData>***, агрегирующий коллекцию уникально идентифицируемых деревьев класса ***ForestTree<TData>***. Последний является наследником рассмотренного выше класса ***Tree<TData>***. Класс представления

леса реализует специфические операции разбиения и слияния деревьев, необходимые, в частности для реализации алгоритмов с отложенной верификацией ребер [27]. Интерфейс класса включает следующие методы:

- ***findTree(uid_t** treeID **)-> ForestTree***
  найти дерево по уникальному идентификатору
- ***createTree()-> ForestTree***
  создать дерево
- ***deleteTree(ForestTree** tree **)***
  удалить дерево
- ***insertEdge(ForestTree** sourceTree**, Node** sourceNode**, ForestTree** targetTree**, Node** targetNode **)***
  выполнить слияние деревьев *sourceTree* и *targetTree* путем создания ребра, ведущего из узла *sourceNode* в узел *targetNode*
- ***deleteEdge(ForestTree** sourceTree**, Node** node **)***
  выполнить разбиение дерева *sourceTree* путем удаления ребра, ведущее в узел *node*

4.3 Маршрутные сети

Для построения маршрутных сетей предназначен шаблонный класс ***TRoadmap<TVertexData,TEdgeData>***, агрегирующий коллекции вершин и ребер соответствующих классов ***TRoadmap::Vertex*** и ***TRoadmap::Edge***. В качестве параметров шаблона выступают пользовательские типы данных, которые используются для представления атрибутов, приписанных вершинам и ребрам. Данный класс ***TRoadmap*** реализует основные операции, необходимые для построения и модификации маршрутных сетей:

- ***createVertexIterator()-> iterator<IRoadmapVertex>***
  создать итератор для обхода вершин
- ***createEdgeIterator()-> iterator<IRoadmapEdge>***
  создать итератор для обхода ребер
- ***createVertex()-> Vertex***
  создать вершину
- ***createEdge(Vertex firstVertex,Vertex secondVertex)-> Edge***
  создать ребро
- ***deleteVertex(Vertex vertex)***
  удалить вершину
- ***deleteEdge(Edge edge)***
  удалить ребро
- ***updateEdgeCost(Edge edge, ICost cost)***

установить вес ребра

- ***createPath()->TRoadmapPath<Vertex>***
  создать путь
- ***clear()***
  очистить структуру

Шаблонные классы маршрутных сетей и элементов сети наследуются от соответствующихся абстрактных классов ***IRoadmap***, ***IRoadmapVertex***, ***IRoadmapEdge***. В конечном счете, это обеспечивает возможность обобщенной реализации алгоритмов планирования движения на уровне абстрактных классов независимо от способов представления сетей и особенностей доступа к их атрибутам.

4.4 Исчисление стоимости

В задачах поиска оптимальных путей обычно ребрам маршрутной сети приписывают веса или стоимости переходов между смежными вершинами. В зависимости от прикладной постановки стоимость переходов может определять разные критерии поиска. В большинстве случаев решается задача поиска наикратчайшего маршрута в рабочем пространстве, а стоимость переходов — длина маршрута между точками рабочего пространства, соответствующими вершинам сети.

В других случаях решаются задачи маршрутизации с учетом иных, в том числе множественных критериев. Например, в работе [28] ставится задача маршрутизации объекта, в которой предпочтение отдается поступательному движению и минимизируется вращательная составляющая. В работе [29] рассматривается задача поиска безопасных маршрутов, наиболее удаленных от препятствий окружения.

Для унификации способов задания критериев поиска в среде предусмотрены соответствующие классы для представления и исчисления стоимостей. Предполагается, что стоимости представимы абстрактным классом ***ICost***, а абстрактный класс ***ICostSpace*** определяет сигнатуры операций над ними:

- ***null()->ICost***
  получить нулевое значение стоимости
- ***equal(ICost a, ICost b)->bool***
  оператор равенства
- ***less(ICost a, ICost b)->bool***
  оператор сравнения
- ***add(ICost a, ICost b)->ICost***
  оператор сложения

Средой допускается задание условий многокритериального поиска, поэтому абстрактные классы ***ICost*** и ***ICostSpace*** уточняются соответствующими

классами для операций над скалярными величинами (***ISimpleCost*** и ***ISimpleCostSpace***) и векторными величинами (***ICompoundCost*** и ***ICompoundCostSpace***). Первая пара классов реализуется в виде шаблонов ***TSimpleCost<TValue>*** и ***TSimpleCostSpace<TValue>***, параметризуемых конкретным типом скалярной переменной. Вторая пара реализуется в виде шаблонных кортежей ***TCompoundCost<TValues…>*** и ***TCompoundCostSpace<TValues…>***. При этом в классе ***TCompoundCostSpace*** предусмотрена возможность задания бинарного отношения доминирования по Парето или Слейтеру. Данные отношения реализуются соответствующими классами ***ParetoDominanceRelation*** и ***SlaterDominanceRelation*** с общим интерфейсом ***IDominanceRelation***.

4.5 Маршрутизаторы

Алгоритмы поиска путей реализуются в конкретных классах, наследуемых от абстрактного класса ***IRoadmapPlanner*** и получающих доступ к заданной маршрутной сети через его ассоциацию типа ***IRoadmap***. Поиск осуществляется с помощью метода ***findPath(IRoadmapVertex*** *initialVertex*, ***IRoadmapVertex*** *goalVertex*, ***IRoadmapPath*** *path* ***)-> bool***.

Для реализации эвристических алгоритмов предназначен абстрактный класс ***IHeuristicRoadmapPlanner***, который является специализацией базового класса ***IRoadmapPlanner*** и дополнительно агрегирует объект типа ***ICostSpace*** для задания эвристической функции приоритизации вершин.

Среда предоставляет реализации нескольких популярных алгоритмов поиска путей, которые представлены конкретными классами ***Dijkstra*** (алгоритм Дейкстры [23]), ***AStar*** (алгоритм A* [23]), ***LPAStar*** (алгоритм LPA* [30]) и ***DStarLight*** (алгоритм D*-light [31]). Тем самым разработчику предоставляется возможность выбора алгоритма маршрутизации, наиболее подходящего для решаемых прикладных задач и применяемых методов планирования движения.

Для планирования в динамических маршрутных сетях класс ***TRoadmap*** реализует механизм оповещения наблюдателей типа ***IRoadmapObserver***. Сам механизм оповещения реализуется в конкретных классах маршрутизаторов, наследуемых от базового ***IRoadmapPlanner***, который в свою очередь наследуется от интерфейса наблюдателя ***IRoadmapObserver***. Данный интерфейс определяет следующие методы реакции на изменения в маршрутной сети:

- ***onAfterVertexAdded(Vertex*** *vertex* ***)***
  добавлена новая вершина
- ***onBeforeVertexRemoved(Vertex*** *vertex* ***)***
  вершина будет удалена
- ***onAfterEdgeAdded(Edge*** *edge* ***)***

добавлено новое ребро

- ***onBeforeEdgeRemoved(Edge** edge **)***
  ребро будет удалено
- ***onAfterEdgeCostChanged(Edge** edge **)***
  изменился вес ребра
- ***onBeforeClear()***
  структура будет очищена

Данные методы реализуются в конкретных классах маршрутизаторов с учетом характера изменений. Например, в алгоритмах LPA* и D*-light в качестве реакции на изменение веса ребра обновляется приоритетная очередь вершин.

5. Пакет классов StateSpace

Под состоянием или конфигурацией объекта понимается набор значений параметров, однозначно определяющих положение всех точек его геометрической модели в трехмерном пространстве окружения. Обычно используется минимальный набор параметров, соответствующий количеству степеней свободы объекта и определяющий конфигурационное пространство объекта. Задача планирования пути формулируется как задача построения бесконфликтной непрерывной траектории в конфигурационном пространстве объекта, которая соединяет заданную пару точек, соответствующих его начальному и конечному состоянию.

Конфигурационное пространство, допустимое состояние, траектория, генератор конфигураций, планировщик и верификатор траекторий — ключевые математические абстракции, которые положены в основу средств планирования движения в составе объектно-ориентированной среды. Подобные абстракции представлены соответствующими классами и интерфейсами ***IStateSpace***, ***IState***, ***ISampler***, ***IProbabilisticPlanner***, ***TSteeringMethod<TState>***. Обсудим особенности их реализации более подробно.

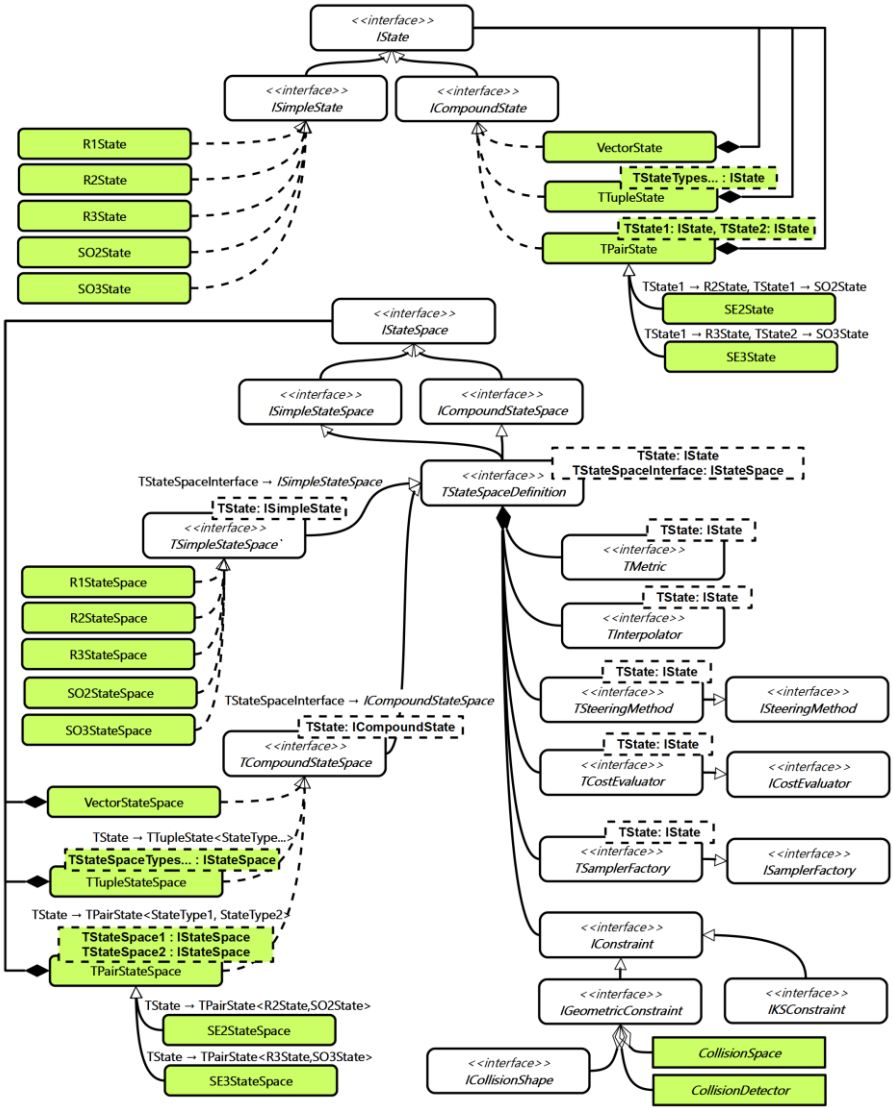


Рис. 7. Диаграмма классов конфигурационного пространства

Fig. 7. Configuration space class diagram.

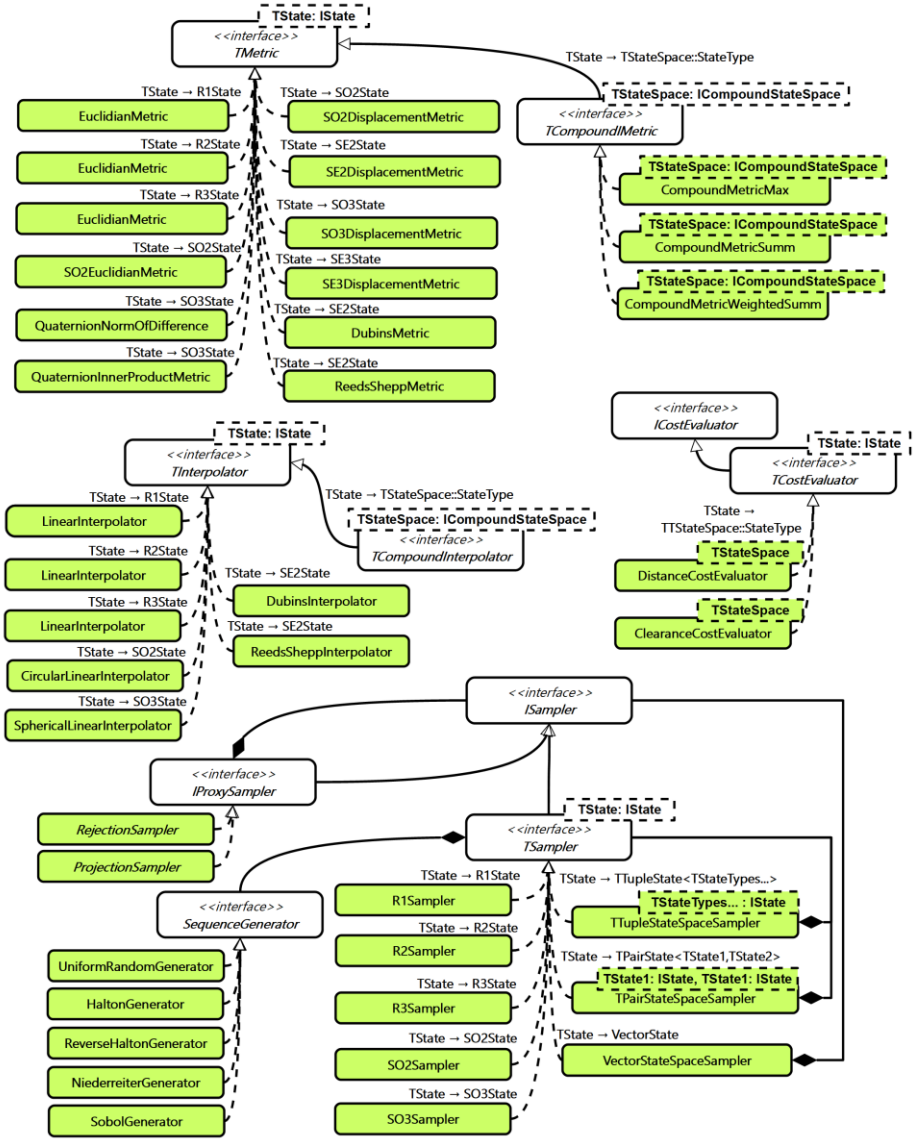


Рис. 8. Диаграмма классов конфигурационного пространства

Рис. 8. Configuration space class diagram.

5.1 Конфигурационные пространства

Выделение абстракций конфигурационного пространства и точки пространства и их представление в среде соответствующими интерфейсами ***IStateSpace*** и ***IState*** обусловлены необходимостью поддержать альтернативные способы задания обобщенных координат объекта. Например, твердое тело, свободно движущееся в трехмерном пространстве, имеет шесть степеней свободы, соответствующих поступательному и вращательному движению. Более сложные объекты могут иметь большее количество степеней свободы. Например, состояние манипуляционного робота может задаваться многомерным вектором, элементы которого определяют углы поворота всех его подвижных шарниров. При этом важно обеспечить возможность обобщенной реализации алгоритмов планирования движения и, в частности популярных сэмплинг алгоритмов, без какой-либо конкретизации явных или неявных способов задания обобщенных координат объекта. Интерфейс ***IStateSpace*** определяет необходимый для этого набор методов:

- ***getSpaceDimension()->integer***
  возвращает размерность конфигурационного пространства
- ***distance(IState first, IState second)->real***
  возвращает значение расстояния между конфигурациями
- ***interpolate(IState start, IState end, real time)->IState***
  возвращает интерполированную точку по заданной начальной и целевой конфигурации и параметру $time \in [0; 1]$
- ***isFree(IState state)->bool***
  верифицирует заданную конфигурацию
- ***isFree(IState state)->(bool,float)***
  верифицирует заданную конфигурацию и возвращает значение расстояния до ближайшего препятствия
- ***getSteeringMethod()->ISteeringMethod***
  предоставляет доступ к верификатору путей
- ***getCostEvaluator()->ICostEvaluator***
  предоставляет доступ к вычислителю стоимости движения
- ***createUniformSampler()->ISampler***
  создает генератор конфигураций, равномерно распределенных в заданной области

Вопросы применения данных методов при реализации сэмплинг алгоритмов подробно обсуждаются в следующих разделах. В данном разделе остановимся на специализациях базового класса ***IStateSpace***, в частности, на абстрактных классах ***ISimpleStateSpace*** и ***ICompoundStateSpace***, служащих для определения простых и составных конфигураций. В отличие от ***ISimpleStateSpace***, класс ***ICompoundStateSpace*** предполагает задание

конфигурационного пространства составного объекта как прямого декартова произведения пространств соответствующих частей. Выделение перечисленных абстракций позволяет унифицировать основные операции анализа конфигураций, а также упростить процедуру их обратного преобразования в рабочее трехмерное пространство.

ICompoundStateSpace конкретизируется классами, определяющими конфигурационное пространство как произведение фиксированного и переменного числа операндов: ***TPairStateSpace<TSpace1,TSpace2>***, ***TTupleStateSpace<TStateSpace...>***, ***VectorStateSpace***.

Например, конфигурационное пространство твердого тела, свободное движение которого в трехмерном пространстве определяется группой преобразований $SE(3) = R^3 \times SO(3)$, может быть задано следующей специализацией шаблона ***SE3Space=TPairStateSpace<R3Space,SO3Space>***. Конфигурационные пространства колесных механизмов ***DubinsSpace*** и ***ReedsSheppSpace*** могут быть определены путем наследования от класса ***SE2Space=TPairStateSpace<R2Space,SO2Space>***, соответствующего группе преобразований $\mathrm{SE}(2) = \mathrm{R}^2 \times \mathrm{SO}(2)$.

Наконец, класс ***VectorStateSpace*** позволяет задавать конфигурационные пространства динамически в ходе выполнения программы, поскольку поддерживает неоднородную коллекцию объектов типа ***IStateSpace*** переменного размера. В частности, данная возможность полезна при работе с кинематическими системами, описание которых хранится в файлах или формируется непосредственно в ходе пользовательской сессии.

Для определения конкретных классов конфигурационных пространств со строгим контролем соответствия типов и гибкой настройкой поведения предназначен шаблонный класс ***TStateSpaceDefinition<TState,TStateSpaceInterface>***. Параметр шаблона ***TState*** соответствует конкретному классу конфигурации типа ***IState***, а ***TStateSpaceInterface*** — интерфейсу простого или составного конфигурационного пространства ***ISimpleStateSpace*** или ***ICompoundStateSpace***.

Данный класс предусматривает агрегацию ключевых алгоритмических компонентов, необходимых для реализации методов интерфейса ***IStateSpace***. Поскольку алгоритмические компоненты одного назначения представлены единой иерархией классов, в классе возможна настройка альтернативных алгоритмов соответствующих типов. В данной проектной схеме делегирования операции определения расстояния между конфигурациями и интерполяции точек выполняются установленными объектами классов ***TMetric<TState>*** и ***TInterpolator<TState>***. Верификация линейных сегментов пути реализуется назначенным объектом класса ***TSteeringMethod<TState>***. Оценка стоимости перехода между конфигурациями делегируется соответствующему объекту класса ***TCostEvaluator<TState>***, а генерация

конфигураций с равномерным рапределением — объекту класса ***TSamplerFactory<TState>***.

5.2 Генераторы конфигураций

Генерация конфигураций является ключевым элементом всех сэмплинг методов планирования движения и имеет свои особенности. Например, в работе [32] показано, что наивный метод, основанный на случайном выборе координат, соответствующих эйлеровым углам, не обеспечивает равномерное распределение конфигураций в пространстве SO(3) и предложен метод на основе единичных кватернионов.

Генераторы в среде представлены абстрактным классом ***ISampler***, в котором определен виртуальный метод получения очередной конфигурации ***generateState()->IState***. Метод допускает альтернативные реализации генераторов псевдослучайных чисел, квазислучайных последовательностей и регулярных сеток [1] в наследуемых конкретных классах.

Среда предоставляет набор классов для построения выборок с равномерным распределением для каждого конкретного класса конфигурационного пространства. В частности, доступны генератор псевдослучайных чисел ***UniformRandomGenerator***, генераторы последовательностей Холтона ***HaltonGenerator*** и ***ReverseHaltonGenerator***, генераторы Соболя ***SobolGenerator*** и Нидеррайтера ***NiederretierGenerator***. На этапе конструирования генераторов устанавливаются границы области сэмплирования.

В реализациях генераторов предусмотрены возможности задания эвристик сэмплирования, позволяющих формировать выборки преимущественно из допустимых конфигураций, имеющих перспективу стать точками конструируемых путей и снижающих вычислительные расходы на поиск подобных точек в равномерно распределенных выборках. Для этих целей используется абстрактный класс ***IProxySampler***, который являясь наследником ***ISampler***, предоставляет метод получения генерируемых конфигураций. В процессе отбора перспективных конфигураций используется вспомогательный генератор исходных конфигураций, например, с равномерным распределением, поэтому класс ***IProxySampler*** агрегирует требуемый для этого объект типа ***ISampler***.

Абстрактный класс ***IProxySampler*** специализируется конкретными классами ***RejectionSampler*** и ***ProjectionSampler***, реализующими две основные стратегии эвристического сэмплирования конфигураций: путем отклонения неперспективных точек на основе заданного условия или путем их последующей трансформации.

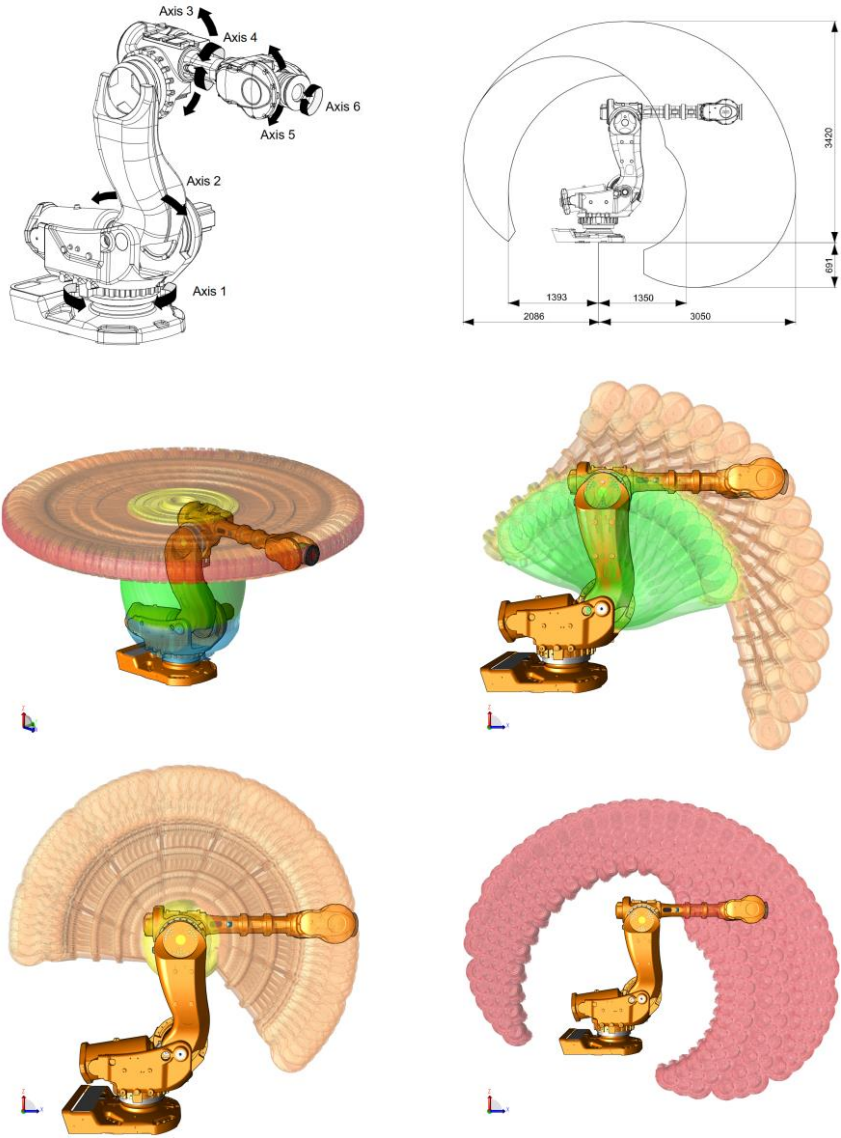


Рис. 9. Результат работы генератора точек в конфигурационном пространстве семизвенного робота

Fig. 9. Example of 6DOF robot configuration space sampling.

Первый подход используется для генерации точек вдоль границ препятствий [33–35] и для отсечения точек по области видимости [36]. Класс ***RejectionSampler*** использует указатель на функцию с сигнатурой ***acceptStateRule(IState*** *state* ***)-> bool*** для проверки соответствия конфигурации заданному условию и пороговое значение, определяющее максимальное количество предпринимаемых попыток генерации надлежащей конфигурации.

Аналогичным образом, класс ***ProjectionSampler*** использует указатель на функцию с сигнатурой ***transformStateRule(IState state)->State*** для трансформации конфигурации в соответствии с заданным правилом. Данный класс может использоваться для построения точек, наиболее удаленных от препятствий [37], или для построения проекций на подпространство конфигураций, удовлетворяющих кинематическим ограничениям [38,39].

5.3 Верификаторы путей

Абстрактный класс ***TSteeringMethod<TState>*** определяет единый интерфейс верификаторов путей, предназначенных для проверки возможности бесконфликтного перехода между парой точек конфигурационного пространства и верификации ребер маршрутной сети. Интерфейс содержит следующий набор виртуальных методов:

- ***verifyMotion(IStateSpace*** *cspace****, IState*** *start****, IState*** *end* ***)-> bool***
  устанавливает факт возможности бесконфликтного перехода из конфигурации *start* в конфигурацию *end*
- ***verifyMotion(IStateSpace*** *cspace****, IState*** *start****, IState*** *end* ***)–> (bool,IState)***
  возвращает последнюю бесконфликтную точку при переходе из конфигурации *start* в конфигурацию *end*
- ***verifyMotion (IStateSpace*** *cspace****, IState*** *start****, IState*** *end* ***)-> (bool,IState[])***
  возвращает массив бесконфликтных точек при переходе из конфигурации *start* в конфигурацию *end*

В состав среды включен конкретный класс ***DiscreteSteeringMethod***, который наследует интерфейс ***TSteeringMethod*** и реализует процедуру верификации путем дискретизации отрезка с сопутствующей проверкой промежуточных точек на столкновения (рис. 10). Шаг дискретизации определяется значением погрешности, которое устанавливается в качестве параметра при конструировании экземпляров конфигурационного пространства типа ***IStateSpace***. Значение погрешности соответствует расстоянию между точками, определяемому заданной метрикой конфигурационного пространства с учетом габаритов трехмерного объекта.

Выделение абстрактного класса ***TSteeringMethod*** обеспечивает возможность реализации более эффективных способов верификации путей с учетом особенностей прикладных задач. Например, в случае простых твердотельных

объектов повысить эффективность верификации путей можно с помощью техники протяжек ограничивающих выпуклых оболочек или алгоритмов непрерывного определения столкновений (Continuous Collision Detection) [40,41].

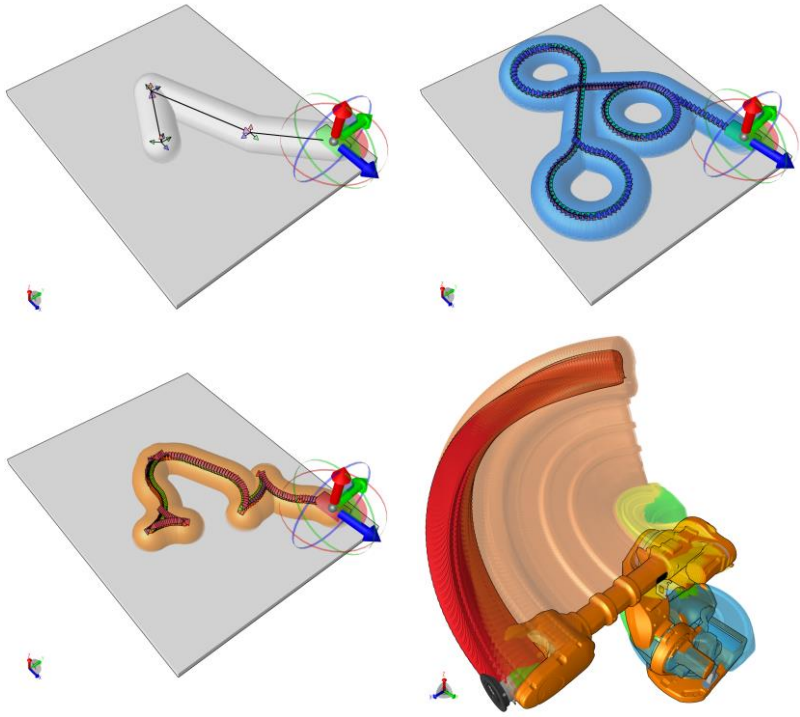


Рис. 10. Верификация пути для различных моделей движения
Fig. 10. Path verification for different types of motion.

5.4 Вычислитель стоимости переходов

Алгоритмы планирования движения, нацеленные на поиск оптимальных путей в процессе их построения, в частности, TRRT [29], RRT* [24] и TRRT* [42], требуют задания функции для оценки качества решений. Для этих целей служит абстрактный класс ***TStateCostEvaluator<TState>***, наследующий интерфейс стоимостного пространства ***ICostSpace*** и дополнительно определяющий виртуальный метод ***evaluate(IState*** *from****, IState*** *to* ***)-> ICost***. Данный метод выступает в роли функции стоимости перехода между

заданными конфигурациями и реализуется в конкретных наследуемых классах вычислителей. Тем самым, средой допускается поддержка альтернативных критериев оптимальности.

Для поиска оптимальных путей по критериям длины и удаленности от препятствий реализованы и включены в состав среды классы ***DistanceStateCostEvaluator*** и ***ClearanceStateCostEvaluator***. Для задания критериев, определяемых пользователем, может быть использован вспомогательный абстрактный шаблонный класс ***TStateCostEvaluator<TStateSpace,TCostSpace>***, специализация которого выполняется в результате задания конкретных классов конфигурационного и стоимостного пространств.

6. Подсистема локального планирования движения

6.1 Запросы планирования движения

Алгоритмы локального планирования движения в конфигурационном пространстве объекта реализуются на основе абстрактного класса ***IProbabilisticPlanner***, который предоставляет внешний интерфейс запросов планирования в виде ***findPath(IState** initialState**, IState** goalState**, IPath** path **)-> PlanningResult***. Входными параметрами метода являются начальная и целевая конфигурации объекта, а возвращаемые результаты — ссылка на построенный путь типа ***IPath*** и значение перечислимого типа ***PlanningResult***, отражающее статус выполнения запроса планирования: *SUCCESS* – путь успешно найден, *FAILURE* – бесконфликтный путь не существует, *PROBABLY_FAILURE* – путь не найден, *INVALID_INPUT* – неверно заданы входные данные, *INTERNAL_ERROR* – внутренняя ошибка программы.

Найденные пути представляются шаблонным классом ***TPath<TStateSpace>***, наследующим интерфейс ***IPath*** и параметризуемым конкретным типом конфигурационного пространства, в котором они строятся. Каждый путь представляет собой упорядоченную коллекцию бесконфликтных конфигураций. При этом подразумевается, что сегменты путей между соседними конфигурациями, построенные в соответствии с предопределенной интерполяционной функцией, также неконфликтны.

Поскольку запросы планирования движения объекта могут быть одиночными и множественными, каждый планировщик типа ***IProbabilisticPlanner*** хранит ссылку на конфигурационное пространство объекта, в котором разрешаются подобные запросы и для которого может быть уже развернуты деревья поиска или маршрутная сеть. Данная ссылка реализуется соответствующей ассоциацией класса и устанавливается при конструировании планировщиков.

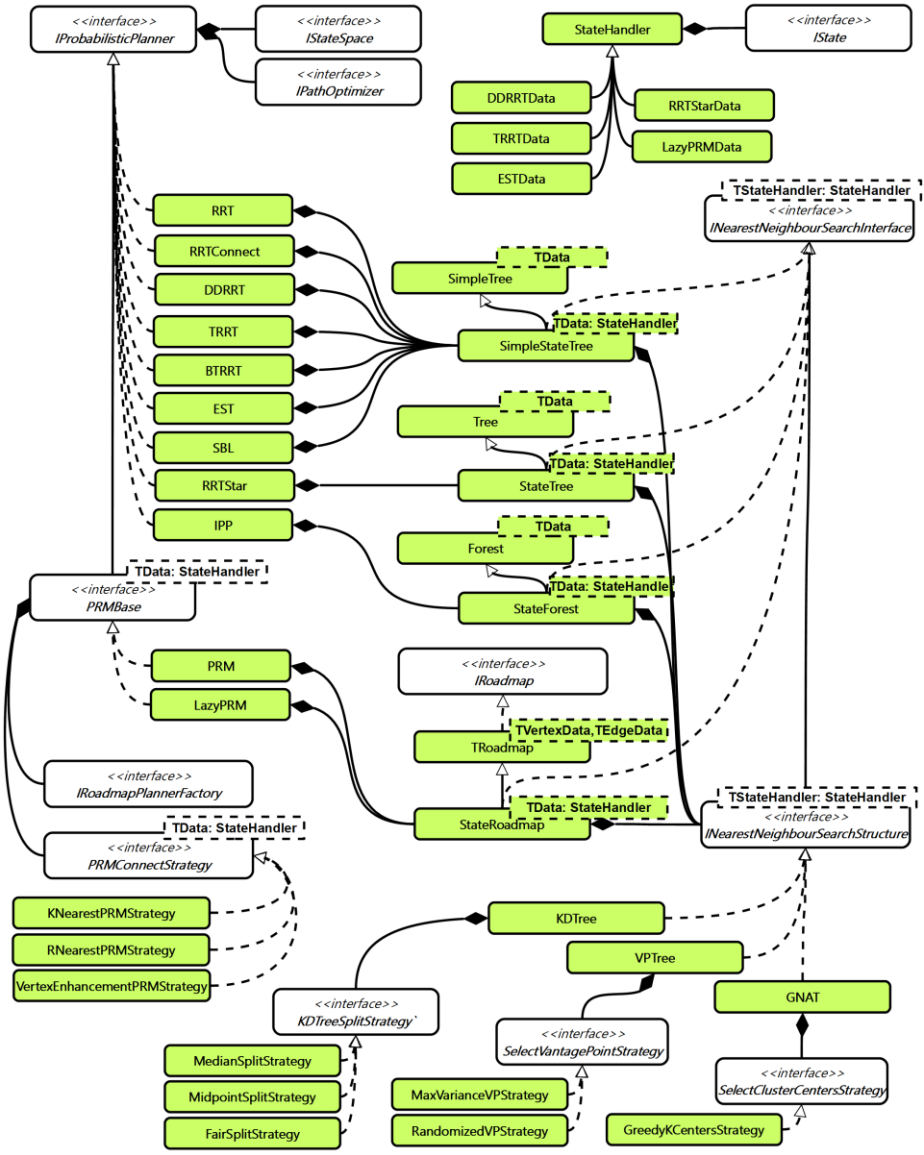


Рис. 11. Диаграмма классов локального планирования движения
Fig. 11. Local planning class diagram.

Общая алгоритмическая схема разрешения запроса планирования реализуется непосредственно в классе ***IProbabilisticPlanner***, в котором проводится контроль входных данных, предобработка, необходимая, например, для реализации алгоритмов на основе вероятностных маршрутных сетей, сам поиск и постобработка найденных путей для их оптимизации. Сами алгоритмы реализуются в наследуемых или ассоциируемых классах. В частности, на основе класса ***IProbabilisticPlanner*** реализуется обсуждаемое ниже семейство сэмплинг алгоритмов с различными эвристическими стратегиями поиска.

6.2 Пространственные индексы

Одной из базовых операций, используемых при реализации сэмплинг алгоритмов, является поиск ближайших соседей в представлении поискового дерева. Для повышения производительности обычно используют пространственные индексы, которые строятся на множестве точек, полученных в результате сэмлирования конфигурационного пространства.

Среда предоставляет несколько готовых к использованию классов, реализующих необходимые индексные структуры, а именно: ***KDTree*** (kD-дерево [43]), ***GNAT*** (Geometric Near-neighbor Access Tree [44,45]) и ***VPTree*** (Vantage Point Tree [46]). Поскольку затраты на построение индексов и исполнение запросов могут существенно варьироваться в зависимости от размерности пространства, характера распределения допустимых конфигураций и алгоритма сэмплирования, организация классов индексов предусматривает возможность подмены альтернативных реализаций. С этой целью конкретные реализации классов индексов унаследованы от абстрактного класса ***INearestNeighbourSearchStructure*** и его базового интерфейса ***INearestNeighbourSearchInterface<TStateHandler>***, определяющего запросы поиска ближайших соседей в известных постановках:

- ***nnSearch(IState state) -> TStateHandler***
  поиск ближайшей точки
- ***kNNSearch(IState state, integer K) -> TStateHandler[]***
  поиск K ближайших точек
- ***rNNSearch(IState state, float R) -> TStateHandler[]***
  поиск точек, находящихся в радиусе R от заданной точки

Поскольку реализация пространственных индексов предполагает задание метрики, базовый класс индексов определяет соответствующую ассоциацию на конфигурационное пространство типа ***IStateSpace*** с необходимой функцией определения расстояния между заданными точками. Параметр шаблона ***StateHandler*** имеет тип вершины поискового дерева или маршрутной сети, связанной с соответствующей точкой конфигурационного пространства.

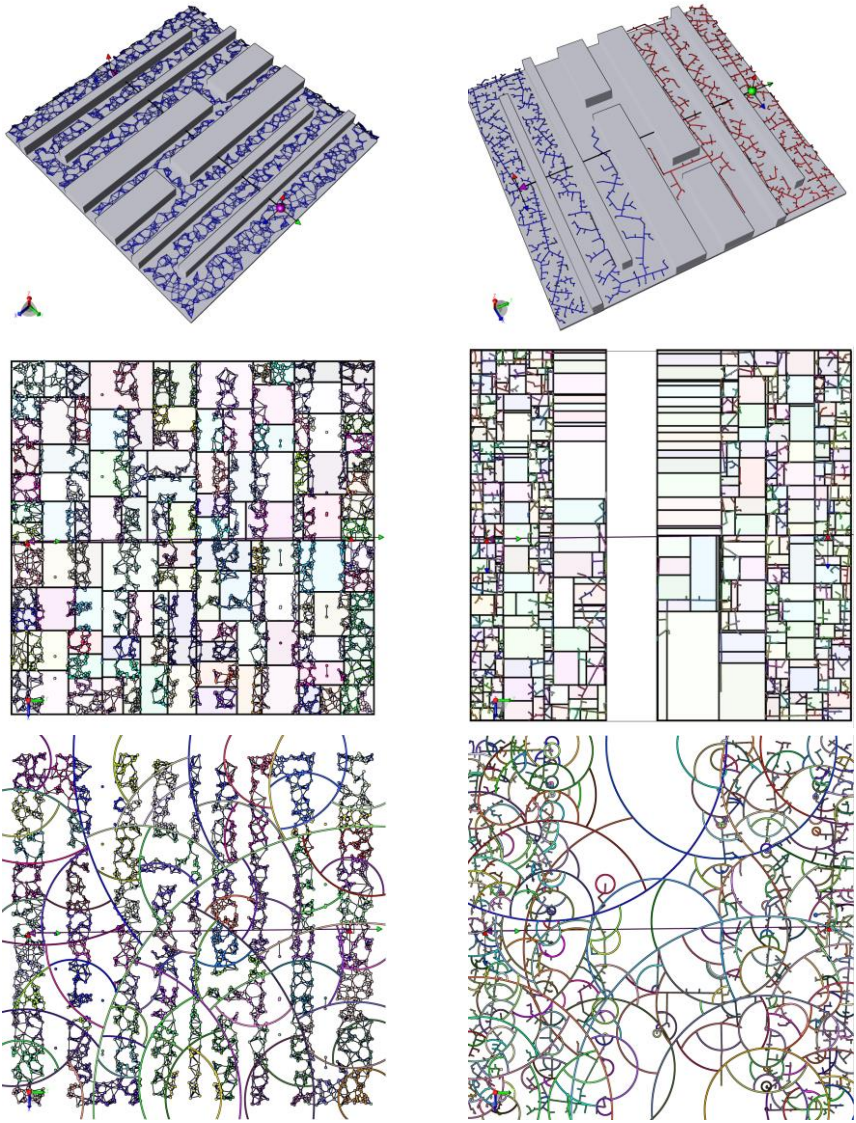


Рис. 12. kD-деревья и vp-деревья,построенные на множестве вершин вероятностной маршрутной сети (PRM) и быстро растущих случайных деревьев (RRT-connect)
Fig. 12. kD-trees and и vp-trees built on PRM and RRT-connect vertex set.

Для удобства реализации различных алгоритмов планирования движения в состав среды включены шаблонные классы ***StateTree<TNodeData>***, ***StateForest<TNodeData>*** и ***StateRoadmap<TVertexData,TEdgeData>***, которые, являясь специализациями соответствующих базовых классов ***Tree***, ***Forest*** и ***Roadmap***, наследуют необходимые операции работы с графами. Помимо этого, данные классы реализуют наследуемый интерфейс ***INearestNeighbourSearchInterface***, делегируя выполнение операций поиска ближайших соседей агрегируемому пространственному индексу типа ***INearestNeighbourSearchStructure***. Индекс обновляется автоматически при добавлении, удалении, модификации элементов поисковых деревьев и маршрутных сетей.

6.3 Семейство локальных планировщиков путей

Рассмотренные выше классы среды служат удобным инструментарием для реализации популярных алгоритмов планирования движения, основанных на поисковых деревьях и вероятностных маршрутных сетях. Обобщенные шаблонные реализации позволяют относительно просто конфигурировать интерфейсы и классы среды для разработки приложений и настройки алгоритмов с учетом плотности покрытия в окрестности вершин [47], количества успешных и неуспешных попыток распространения [48], динамической области сэмлирования [26], суммарной стоимости пути из корня дерева [24] и т.п.

Среда предоставляет развитое семейство локальных планировщиков путей, реализованных в виде соответствующих классов-наследников ***IProbabilisticPlanner***. Классы ***RRT*** и ***EST*** реализуют алгоритмы на основе поисковых деревьев Rapidly Exploring Random Trees и Expansive-Spaces Trees соответственно. Алгоритмы с онлайн-оптимизацией деревьев Transition-based RRT и RRT* реализованы в классах ***TRRT*** и ***RRTStar***. Алгоритмические версии, адаптированные для двунаправленного поиска, представлены классами ***RRTConnect***, ***SBL*** и ***BTRRT***. Класс ***IPP*** реализует диффузионный алгоритм с отложенной проверкой на столкновения (Iterative Diffuse Path Planner). Наконец, алгоритмы на основе вероятностных маршрутных сетей, ориентированные на обработку множественных запросов поиска, представлены классами ***PRM*** и ***LazyPRM***.

6.4 Оптимизация путей

Случайный характер поиска допустимых конфигураций и дискретный способ построения путей сэмплинг алгоритмами крайне негативно влияют на качество получаемых решений. Естественными требованиями, предъявляемыми к найденным путям, являются их минимальная длина, гладкость, наибольшее удаление от препятствий окружения и т.п. В связи с этим постобработка найденных путей является важным этапом улучшения их

качества, который предусматривается классом локальных планировщиков ***IProbabilisticPlanner***.

С этой целью на заключительном этапе выполнения запроса планирования вызывается метод ***optimize(IStateSpace*** *cspace****, IPath*** *path* ***)-> IPath*** в назначенном оптимизаторе путей типа ***IPathOptimizer***. Данный метод вызывается автоматически при успешном выполнении предыдущих этапов и, в частности, при наличии хотя бы одного найденного бесконфликтного пути. Оптимизация пути выполняется с помощью алгоритмов сглаживания, укорачивания и построения ретракта [28], реализуемых соответствующими классами ***PathSmoothing***, ***PathShortening*** и ***PathRetractor*** — наследниками ***IPathOptimizer***.

7. Подсистема глобального планирования движения

Рассмотренные выше программные средства среды обеспечивают задание условий и решения задач планирования движения в локальной статической постановке. Вместе с тем, на практике возникает необходимость решения более сложных задач, связанных с построением путей в сложном динамическом окружении. С этой целью в состав среды включены соответствующие средства, составляющие подсистему глобального планирования и реализующие общую вычислительную стратегию, предложенную и апробированную авторами ранее [13,49].

Стратегия подразумевает построение единой маршрутной сети в рабочем трехмерном пространстве окружения, которая затем используется для принятия решений о наиболее перспективных маршрутах. Отобранные маршруты верифицируются и при необходимости корректируются локальным планировщиком с учетом геометрии конкретного объекта и наложенных на него кинематических ограничений. Представление сети обновляется синхронно с событиями, происходящими в динамическом окружении. Для построения сети могут применяться методы пространственной декомпозиции, диаграммы Вороного или планы окружения, построенные вручную.

Принципы организации и функционирования подсистемы обеспечивают возможность гибкого конфигурирования глобального планировщика из компонентов, реализующих альтеративные способы построения случайных деревьев и маршрутных сетей, а также осуществляющих их верификацию и коррекцию рассмотренными выше алгоритмами локального планирования.

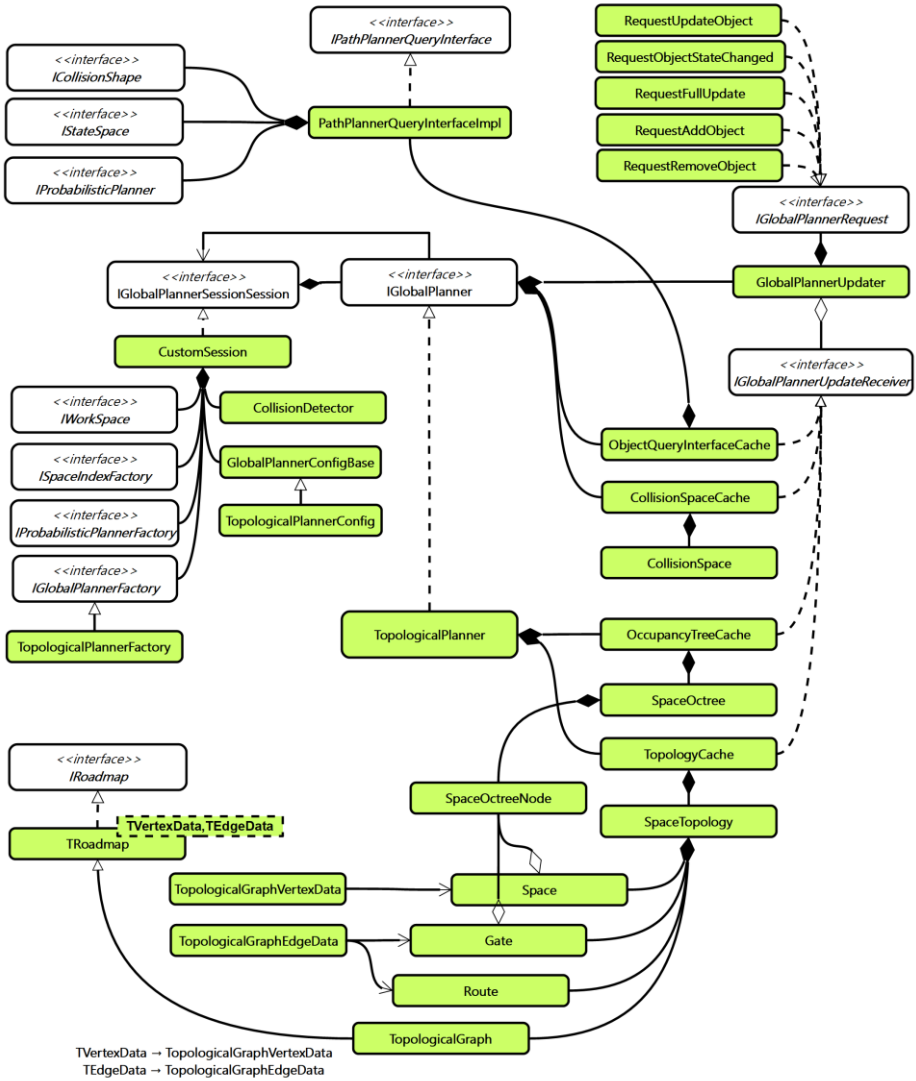


Рис. 13. Диаграмма классов подсистемы глобального планирования

Fig. 13. Global planning subsystem class diagram

7.1 Глобальный планировщик

Глобальный планировщик выполняет две основные функции: формирование единой маршрутной сети на основе пространственного анализа трехмерного окружения и разрешение запросов планирования движения с помощью развернутой сети. Эффективное исполнение множественных запросов предполагает поддержку согласованного представления сети на протяжении всей пользовательской сессии. Это же относится и к связанным с ней вспомогательным данным, в частности, альтернативному геометрическому представлению окружения и пространственным индексам. В случае динамического окружения сеть и вспомогательные данные должны обновляться синхронно с происходящими событиями, причем инкрементальным образом. Поскольку подобные обновления являются вычислительно затратными операциями, они выполняются в фоновом режиме.

За реализацию перечисленных функций в среде отвечает класс планировщиков ***IGlobalPlanner***. Остановимся более подробно на механизме сообщений, с помощью которого происходят обновления в подсистеме глобального планирования.

Сообщения типизируются и представляются следующим набором классов, наследуемых от абстрактного ***IGlobalPlannerRequest***:

- ***RequestAddObject***
  создан новый объект
- ***RequestRemoveObject***
  объект удален
- ***RequestUpdateObject***
  изменилось представление объекта
- ***RequestObjectStateChanged***
  изменилось состояние объекта
- ***RequestFullUpdate***
  требуется полное обновление всех насчитываемых данных

Обновление вспомогательных данных реализуется классами обработчиков как реакция на полученные сообщения. Данные классы наследуют от абстрактного ***IPathPlannerUpdateReceiver*** следующие методы:

- ***getUniqueName()->string***
  возвращает уникальное имя обработчика
- ***dependsOn(string*** *receiverName* ***)-> bool***
  установливает факт зависимости от другого обработчика с заданным именем
- ***fullUpdate()***
  выполняет полное обновление
- ***smartUpdate(IPathPlannerRequest*** *request* ***)***

выполняет инкрементальное обновление в зависимости от типа сообщения

Обработчик сообщений автоматически регистрируется в диспетчере сообщений при конструировании. Сообщения попадают в очередь диспетчера и затем рассылаются всем зарегистрированным обработчикам.

Для поддержки функций глобального планирования в целевом приложении необходимо разработать класс пользовательской сессии с предопределенным интерфейсом ***IGlobalPlannerSession*** и следующими виртуальными методами конфигурирования подсистемы планирования:

- ***getConfig()->GlobalPlannerConfig***
  возвращает настройки подсистемы планирования
- ***getWorkSpace()->IWorkSpace***
  предоставляет доступ к окружению
- ***getCollisionDetector()->CollisionDetector***
  возвращает анализатор столкновений
- ***getSpaceIndexFactory()->ISpaceIndexStructureFactory***
  предоставляет доступ к фабрике пространственных индексов
- ***getLocalPlannerFactory()->IProbabilisticPlannerFactory***
  предоставляет доступ к фабрике локальных планировщиков
- ***getGlobalPlannerFactory()->IGlobalPlannerFactory***
  предоставляет доступ к фабрике глобальных планировщиков
- ***OnSessionStart()->bool***
  выполняет необходимые действия при старте сессии
- ***OnSessionStop()***
  выполняет необходимые действия при завершении сессии

Целевому приложению при этом доступны следующие методы, определяемые **IGlobalPlannerSession**:

- ***Start()->bool***
  начинает сессию
- ***Stop()***
  завершает сессию
- ***getSessionID()->uid_t***
  возвращает уникальный идентификатор сессии
- ***getPathPlannerQueryInterface(uid_t*** *objectID* ***)-> PathPlannerQueryInterfaceHandler***
  предоставляет интерфейс поиска пути для заданного объекта
- ***processRequest(IGlobalPlannerRequest*** *request* ***)***
  уведомляет о событиях в динамическом окружении

Проследим последовательность вызовов методов в ходе работы подсистемы глобального планирования. Запуск подсистемы инициируется методом ***Start()***, который выполняет верификацию настроек с последующим созданием планировщика типа ***IGlobalPlanner***. Данная операция выполняется путем обращения к соответствующей фабрике, доступной в результате вызова метода сессии ***getGlobalPlannerFactory()***.

При конструировании глобальный планировщик создает диспетчер сообщений, инициирует создание и регистрацию обрабочиков сообщений. В самом планировщике разворачивается кэш геометрических моделей объектов окружения ***CollisionSpaceCache*** и кэш обработчиков запросов поиска пути ***ObjectQueryInterfaceCache***.

На следующем шаге вызывается метод ***fullUpdate()*** диспетчера сообщений, который выполняет принудительное обновление всех насчитываемых вспомогательных данных, после чего инициируется событие ***OnSessionStart()***, уведомляющее приложение об успешном запуске сессии.

Уведомление подсистемы об изменениях в окружении осуществляется путем вызова метода ***processRequest(IGlobalPlannerRequest*** *request* ***)***, принимающее в качестве параметра соответствующее типизированное сообщение. Сообщения помещаются в очередь диспетчера и управление возвращается основному потоку программного приложения. Обработка сообщений происходит в фоновом режиме в соответствии с порядком их поступления. Диспетчер берет из очереди первое поступившее сообщение и уведомляет о нем всех зарегистрированных обработчиков с учетом зависимостей. Для каждого обработчика вызывается метод ***smartUpdate(IPathPlannerRequest*** *request* ***)***, после чего сообщение уничтожается.

Для исполнения запросов планирования движения предназначен класс ***GlobalPlannerObjectQueryInterface***, внешний интерфейс которого представлен единственным методом ***findPath(IState*** *initialState*, ***IState*** *goalState*, ***IPath*** *path* ***)-> PlanningResult***. Данный класс агрегирует конфигурационное пространство типа ***IStateSpace*** и локальный планировщик типа ***IProbabilisticPlanner*** для решения соответствующих задач планирования с приписанным объектом окружения. Инстанциирование класса осуществляется в результате вызова метода сессии ***getPathPlannerQueryInterface(uid_t*** *objectID* ***)*** с указанным идентификатором объекта. Инстанциирование влечет за собой выполнение следующих действий. Во-первых, осуществляется поиск объекта по уникальному идентификатору посредством метода ***IWorkSpace::findObject()***. Для найденного объекта вызывается метод ***IWorkSpaceObject::createStateSpace()*** и конструируется экземпляр конфигурационного пространства. Далее по заданной ассоциации на геометрическую модель окружения, находящуюся в кэше глобального планировщика, формируются геометрические ограничения класса ***GeometricConstraint***. Далее, путем вызова метода сессии

getLocalPlannerFactory() создается локальный планировщик, ассоциированный с конфигурационным пространством объекта.

Построение пути осуществляется следующим образом. По окончании обработки всех сообщений очередь диспетчера блокируется. Далее строятся ограничивающие параллелепипеды геометрической модели объекта в начальном и конечном положениях, которые передаются глобальному планировщику для поиска предварительного маршрута.

Верификация и коррекция маршрута осуществляется локальным планировщиком, который выполняет построение результирующего бесконфликтного пути в конфигурационном пространстве объекта. Предварительно отобранный маршрут позволяет ограничить область сэмплирования и генерировать конфигурации с распределением, обеспечивающим более высокую плотность в труднопреодолимых областях окружения. Для этих целей генератор гауссова распределения применяется в окрестностях ключевых точек маршрута. Радиус окрестности и количество испытаний определяются, исходя из оценок расстояния до препятствий.

7.2 Построение маршрутной сети

Упомянутую выше вычислительную стратегию глобального планирования реализует класс ***TopologicalPlanner***, являющийся наследником класса ***IGlobalPlanner*** и использующийся по-умолчанию при отсутствии других альтернативных реализаций. Стратегия основана на извлечении пространственной, метрической и топологической информации из геометрического представления окружения, построении маршрутной сети в нем и согласованном использовании сети при поиске путей [3,13,14].

Реализация стратегии подразумевает формирование структуры пространственной заполненности всего объема окружения в виде октодерева с приписанными октантам статусами наполненности. Данная структура может рассматриваться в качестве упрощенного геометрического представления окружения. Октодерево дополняется метрической информацией путем приписывания свободным октантам значений расстояний от их центров до препятствий. Метрическая информация используется для кластеризации свободных октантов в виде связных областей пространства. Последние идентифицируются как "зоны" и "переходы" таким образом, что зоны занимают основные свободные области и соединяются друг с другом переходами. В результате восстанавливается топология свободных областей и формируется единая маршрутная сеть окружения в виде графа, вершины которого соответствуют зонам, а ребра — переходам между ними. Для принятия решений о наиболее перспективных маршрутах элементы сети снабжаются информацией о координатах центров выделенных областей, расстоянии до ближайшего препятствия и длинах переходов между смежными зонами.

Для программной реализации вычислительной стратегии используется два вспомогательных класса: ***SpaceOcTree***, предназначенный для формирования структуры пространственной заполненности, и ***SpaceTopology***, используемый для представления описанной специализированной сети.

Класс ***SpaceOcTree*** поддерживает представление октодерева заполненности и инкрементально обновляет его при изменениях, связанных с добавлением, удалением и модификацией объектов окружения. Одновременно пересчитываемые значения расстояний до препятствий позволяют выделить свободные области путем объединения смежных свободных октантов и идентификации их в виде зон и переходов, представимых классами ***Space*** и ***Gate*** соответственно. Маршруты, соединяющие центры инцидентных зон и переходов, выделены в отдельный класс ***Route***.

Зоны, переходы и маршруты агрегируются классом ***SpaceTopology***, который обеспечивает их поддержку в дополнение к графовому представлению маршрутной сети класса ***TopologicalGraph***. Последний наследует рассмотренный выше шаблон ***TRoadmap*** с надлежащей специализацией типов вершин и ребер как классов зон, переходов и маршрутов.

Для согласованного обновления элементов маршрутной сети при изменениях в окружении используются классы обработчиков сообщений ***SpaceOcTreeCache*** и ***SpaceTopologyCache***. Данные классы обеспечивают эффективное инкрементальное обновление соответствующих производных данных в результате обработки типизированных сообщений.

Заключение

Таким образом, рассмотрены принципы организации и функционирования разработанной инструментальной среды для программной реализации моделей, методов и приложений теории планирования движения. Среда предоставляет развитый набор готовых к использованию программных компонентов для автоматического построения бесконфликтных траекторий для робота, перемещаемого в статическом и динамическом трехмерном окружении.

Организация среды в виде объектно-ориентированного каркаса обеспечивает развитие, адаптацию и гибкое конфигурирование разработанных программных компонентов в составе целевых приложений. Благодаря выделенным интерфейсам разного уровня и предусмотренным точкам расширения среда допускает интеграцию со сторонними прикладными системами.

Ожидается, что разработанная инструментальная среда, а также связанный с ней метод построения целевых приложений глобального планирования движения позволят существенно сократить сроки и затраты. В дальнейшем планируется апробировать среду в ходе развития системы визуального

моделирования и планирования индустриальных проектов с функциями пространственно-временной верификации.

Список литературы

[1]. LaValle S.M. Planning Algorithms. Cambridge University Press, 2006.
[2]. Казаков К.А., Семенов В.А. Обзор современных методов планирования движения. Труды ИСП РАН том 28, 2016, выпуск 4. стр. 241–292. 10.15514/ISPRAS-2016-28(4)-14
[3]. 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.
[4]. Semenov V.A., Kazakov K.A., Zolotov V.A. Advanced spatio-temporal validation of construction schedules. ICCCBE. 2012.
[5]. Dubins L.E. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. 1957.
[6]. Reeds J.A., Shepp L.A. Optimal paths for a car that goes both forwards and backwards. Pacific J. Math. 1990. Vol. 145, № 2. pp. 367–393.
[7]. Chitsaz H., LaValle S.M., Balkcom D.J., Mason M.T. Minimum wheel-rotation paths for differential-drive mobile robots. Proc. - IEEE Int. Conf. Robot. Autom. 2006. Vol. 2006, № May. pp. 1616–1623.
[8]. Koenig S., Likhachev M. Fast Replanning for Navigation in Unknown Terrain Technology. 2002. Vol. XX, № May. pp. 968–975.
[9]. Naderi K., Rajamaki J., Hamalainen P. RT-RRT*: A Real-Time Path Planning Algorithm Based On RRT*. Proc. 8th ACM SIGGRAPH Conf. Motion Games - SA '15. 2015. pp. 113–118.
[10]. Gipson I., Gupta K., Greenspan M. MPK: An open extensible motion planning kernel. J. Robot. Syst. 2001. Vol. 18, № 8. pp. 433–443.
[11]. Diankov R. Automated Construction of Robotic Manipulation Programs Architecture. 2010. Vol. Ph.D. pp. 1–263.
[12]. Sucan I., Moll M., Kavraki L.E. The Open Motion Planning Library IEEE Robot. Autom. Mag. 2012. Vol. 19, № 4. pp. 72–82.
[13]. Казаков К.А., Морозов С.В., Семенов В.А., Тарлапан О.А. Применение топологических схем для глобального планирования движения в сложном динамическом окружении. ITSE, 2016.
[14]. Semenov V.A. et al. Global path planning in complex environments using metric and topological schemes. Proc. CIB W78-W102. 2011. pp. 26–28.
[15]. Mamou K., Ghorbel F. A simple and efficient approach for 3D mesh approximate convex decomposition. 16th IEEE Int. Conf. Image Process. 2009. pp. 3501–3504.
[16]. Золотов В.А., Семенов В.А. Перспективные схемы пространственно-временной индексации для визуального моделирования масштабных индустриальных проектов. Труды ИСП РАН том 26, вып. 2, стр. 175–196. 10.15514/ISPRAS-2014-26(2)-8
[17]. Bergen G. Van Den. Efficient Collision Detection of Complex Deformable Models using AABB Trees. J. Graph. Tools. 1997. Vol. 2. pp. 1–13.
[18]. Gottschalk S., Lin M.C., Manocha D. OBB Tree: A Hierarchical Structure for Rapid Interference Detection. Proc. SIGGRAPH 96. 1996. № 8920219. pp. 171–180.

[19]. Li H., Wu Y., Wiu Y. An improved dynamic-octree-based judging method of real-time node in moving geometry. Proceedings of the 2013 International Conference on Intelligent Control and Information Processing, ICICIP 2013. 2013. pp. 333–337.
[20]. Tracy D.J., Buss S.R., Woods B.M. Efficient large-scale sweep and prune methods with AABB insertion and removal. Proc. - IEEE Virtual Real. 2009. pp. 191–198.
[21]. van den Bergen G. Proximity queries and penetration depth computation on 3d game objects. Game Dev. Conf. 2001.
[22]. Gilbert E.G., Johnson D.W., S.S. K. A Fast Procedure for Computing Distance Between Complex Objects in Three-Dimensional Space. 1988.
[23]. Russell S.J., Norvig P. Artificial Intelligence: A Modern Approach. Neurocomputing. 1995. Vol. 9, pp. 215-218.
[24]. Karaman S., Frazzoli E. Sampling-based algorithms for optimal motion planning. Int. J. Robot., vol. 30, № 7, 2011, pp. 846–894.
[25]. Kuffner J.J., LaValle S.M. RRT-connect: An efficient approach to single-query path planning. Proc. IEEE Int. Conf. Robot. Autom. ICRA ’00. 2000. Vol. 2, № Icra. pp. 995--1001 vol.2.
[26]. Jaillet L., Yershova A. Lavalle S.M. Simeon T. Adaptive tuning of the sampling domain for dynamic-domain RRTs. 2005 IEEE/RSJ Int. Conf. Intell. Robot. Syst. IROS. 2005. pp. 4086–4091.
[27]. Ferre E., Laumond J.-P. An iterative diffusion algorithm for part disassembly. IEEE Int. Conf. Robot. Autom. 2004. Proceedings. ICRA ’04. 2004. Vol.3, pp. 3149–3154.
[28]. Geraerts R., Overmars M.H. Creating High-quality Paths for Motion Planning. Int. J. Rob. Res. 2007. Vol. 26, № 8. P. 845–863.
[29]. Jaillet L., Cortes J., Simeon T. Sampling-Based Path Planning on Costmaps Configuration-space. IEEE Trans. Robot. 2010. Vol. 26, № 4. P. 635–646.
[30]. Koenig S., Likhachev M., Furcy D. Lifelong Planning A*. Artif. Intell. 2004. Vol. 155, № 1–2. pp. 93–146.
[31]. Koenig S., Likhachev M. D* Lite. Proc. Eighteenth Natl. Conf. Artif. Intell. 2002. pp. 476–483.
[32]. Kuffner J.J. Effective sampling and distance metrics for 3D rigid body path planning. Proc. - IEEE Int. Conf. Robot. Autom. 2004. Vol. 4, pp. 3993--3998.
[33]. Amato N.M., Bayazit O.B., Dale L.K., Jones C., Vallejo D. OBPRM: An Obstacle-Based PRM for 3D Workspaces. Proc. Third Work. Algorithmic Found. Robot. Algorithmic Perspect. Algorithmic Perspect. 1998. pp. 155–168.
[34]. 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.
[35]. Hsu D., Jiang T., Reif J., Sun Z. The bridge test for sampling narrow passages with probabilistic roadmap planners. IEEE Int. Conf. Robot. Autom. 2003. Proceedings. ICRA ’03. 2003. Vol. 3. pp. 4420–4426.
[36]. Nissoux C., Simeon T., Laumond J.-P. Visibility based probabilistic roadmaps. 1999 IEEE/RSJ Int. Conf. Intell. Robot. Syst. 1999. IROS ’99. Proc. 1999. Vol. 3. pp. 1316–1321 vol.3.
[37]. Lien J.-M., Thomas S., Amato N.M. A general framework for sampling on the medial axis of the free space 2003 IEEE Int. Conf. Robot. Autom. (Cat. No.03CH37422). 2003. Vol. 3. pp. 4439–4444.
[38]. Berenson D., Srinivasa S., Ferguson D., Kuffner J.J. Manipulation Planning on Constraint Manifolds. Robotics and Automation, 2009. ICRA’09. 2009.

[39]. Berenson D., Srinivasa S., Kuffner J.J. Task Space Regions: A framework for pose-constrained manipulation planning. Int. J. Rob. Res. 2011. Vol. 30, № 12. pp. 1435–1460.
[40]. Redon S., Kheddar A., Coquillart S. Fast continuous collision detection between rigid bodies. Comput. Graph. Forum. 2002. Vol. 21, № 3. pp. 279–287.
[41]. Redon S., Lin M.C., Manocha D., Kim Y.J. Fast Continuous Collision Detection for Articulated Models. J. Comput. Inf. Sci. Eng. 2005. Vol. 5, № 2. pp. 126.
[42]. Devaurs D., Simeon T., Cortes J. Optimal Path Planning in Complex Cost Spaces with Sampling-Based Algorithms. IEEE Trans. Autom. Sci. Eng. 2016. Vol. 13, № 2. pp. 415–424.
[43]. Yershova A., LaValle S.M. Improving Motion Planning Alorithms by Efficient Nearest-Neighbor Searching. IEEE Trans. Robot. 2006. pp. 1–8.
[44]. Gipson B., Moll M., Kavraki L.E. Resolution Independent Density Estimation for motion planning in high-dimensional spaces. Proc. IEEE Int. Conf. Robot. Autom. 2013. pp. 2437–2443.
[45]. Fredriksson K. Geometric Near-neighbor Access Tree (GNAT) revisited. 2016.
[46]. Yianilos P.N. Data structures and algorithms for nearest neighbor search in general metric spaces. Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms. 1993. pp. 311–321.
[47]. Sanchez G., Latombe J.C. A single-query bi-directional probabilistic roadmap planner with lazy collision checking Robotics Research. 2003. pp. 403–417.
[48]. 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. 1996. Vol. 12, № 4. pp. 566–580.
[49]. 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.

Object-oriented framework for motion planning in complex dynamic environments

[1] *K.A. Kazakov <kazakov@ispras.ru>*
[1,2] *V.A. Semenov <sem@ispras.ru>*
[1] *Ivannikov 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. In this paper, we discuss principles of the organization and functioning of the software framework intended for development of models, methods and applications of motion planning theory. Developed within an object-oriented paradigm the framework includes a wide variety of ready-to-use components that provide the functionality required for automatic search for collision-free trajectories for robots moving in both static and dynamic complex 3D environments. The proposed software design provides extensibility, adaptation and flexible configuration of the developed program components as a part of target

applications. The developed architecture provides ability to integrate with third-party systems via interfaces of different level and extension points.

Keywords: motion planning; path planning; collision detection; software engineering; object-oriented programming.

DOI: 10.15514/ISPRAS-2017-29(5)-11

For citation: Kazakov K.A., Semenov V.A. Object-oriented framework for motion planning in complex dynamic environments. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 185-238 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-11

References

[1]. LaValle S.M. Planning Algorithms. Cambridge University Press, 2006.
[2]. 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. 241-292 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-14.
[3]. 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.
[4]. Semenov V.A., Kazakov K.A., Zolotov V.A. Advanced spatio-temporal validation of construction schedules. ICCCBE. 2012.
[5]. Dubins L.E. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. 1957.
[6]. Reeds J.A., Shepp L.A. Optimal paths for a car that goes both forwards and backwards. Pacific J. Math. 1990. Vol. 145, № 2. pp. 367–393.
[7]. Chitsaz H., LaValle S.M., Balkcom D.J., Mason M.T. Minimum wheel-rotation paths for differential-drive mobile robots. Proc. - IEEE Int. Conf. Robot. Autom. 2006. Vol. 2006, № May. pp. 1616–1623.
[8]. Koenig S., Likhachev M. Fast Replanning for Navigation in Unknown Terrain Technology. 2002. Vol. XX, № May. pp. 968–975.
[9]. Naderi K., Rajamaki J., Hamalainen P. RT-RRT*: A Real-Time Path Planning Algorithm Based On RRT*. Proc. 8th ACM SIGGRAPH Conf. Motion Games - SA '15. 2015. pp. 113–118.
[10]. Gipson I., Gupta K., Greenspan M. MPK: An open extensible motion planning kernel. J. Robot. Syst. 2001. Vol. 18, № 8. pp. 433–443.
[11]. Diankov R. Automated Construction of Robotic Manipulation Programs Architecture. 2010. Vol. Ph.D. pp. 1–263.
[12]. Sucan I., Moll M., Kavraki L.E. The Open Motion Planning Library IEEE Robot. Autom. Mag. 2012. Vol. 19, № 4. pp. 72–82.
[13]. Kazakov K.A., Morozov S.V., Semenov V.A., Tarlapan O.A. Application of topological schemes for global movement planning in a complex dynamic environment. ITSE, 2016 (in Russian).
[14]. Semenov V.A. et al. Global path planning in complex environments using metric and topological schemes. Proc. CIB W78-W102. 2011. pp. 26–28.
[15]. Mamou K., Ghorbel F. A simple and efficient approach for 3D mesh approximate convex decomposition. 16th IEEE Int. Conf. Image Process. 2009. pp. 3501–3504.

[16]. 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.
[17]. Bergen G. Van Den. Efficient Collision Detection of Complex Deformable Models using AABB Trees. J. Graph. Tools. 1997. Vol. 2. pp. 1–13.
[18]. Gottschalk S., Lin M.C., Manocha D. OBB Tree: A Hierarchical Structure for Rapid Interference Detection. Proc. SIGGRAPH 96. 1996. № 8920219. pp. 171–180.
[19]. Li H., Wu Y., Wiu Y. An improved dynamic-octree-based judging method of real-time node in moving geometry. Proceedings of the 2013 International Conference on Intelligent Control and Information Processing, ICICIP 2013. 2013. pp. 333–337.
[20]. Tracy D.J., Buss S.R., Woods B.M. Efficient large-scale sweep and prune methods with AABB insertion and removal. Proc. - IEEE Virtual Real. 2009. pp. 191–198.
[21]. van den Bergen G. Proximity queries and penetration depth computation on 3d game objects. Game Dev. Conf. 2001.
[22]. Gilbert E.G., Johnson D.W., S.S. K. A Fast Procedure for Computing Distance Between Complex Objects in Three-Dimensional Space. 1988.
[23]. Russell S.J., Norvig P. Artificial Intelligence: A Modern Approach. Neurocomputing. 1995. Vol. 9, pp. 215-218.
[24]. Karaman S., Frazzoli E. Sampling-based algorithms for optimal motion planning. Int. J. Robot., vol. 30, № 7, 2011, pp. 846–894.
[25]. Kuffner J.J., LaValle S.M. RRT-connect: An efficient approach to single-query path planning. Proc. IEEE Int. Conf. Robot. Autom. ICRA '00. 2000. Vol. 2, № Icra. pp. 995--1001 vol.2.
[26]. Jaillet L., Yershova A. Lavalle S.M. Simeon T. Adaptive tuning of the sampling domain for dynamic-domain RRTs. 2005 IEEE/RSJ Int. Conf. Intell. Robot. Syst. IROS. 2005. pp. 4086–4091.
[27]. Ferre E., Laumond J.-P. An iterative diffusion algorithm for part disassembly. IEEE Int. Conf. Robot. Autom. 2004. Proceedings. ICRA '04. 2004. Vol.3, pp. 3149–3154.
[28]. Geraerts R., Overmars M.H. Creating High-quality Paths for Motion Planning. Int. J. Rob. Res. 2007. Vol. 26, № 8. P. 845–863.
[29]. Jaillet L., Cortes J., Simeon T. Sampling-Based Path Planning on Costmaps Configuration-space. IEEE Trans. Robot. 2010. Vol. 26, № 4. P. 635–646.
[30]. Koenig S., Likhachev M., Furcy D. Lifelong Planning A*. Artif. Intell. 2004. Vol. 155, № 1–2. pp. 93–146.
[31]. Koenig S., Likhachev M. D* Lite. Proc. Eighteenth Natl. Conf. Artif. Intell. 2002. pp. 476–483.
[32]. Kuffner J.J. Effective sampling and distance metrics for 3D rigid body path planning. Proc. - IEEE Int. Conf. Robot. Autom. 2004. Vol. 4, pp. 3993--3998.
[33]. Amato N.M., Bayazit O.B., Dale L.K., Jones C., Vallejo D. OBPRM: An Obstacle-Based PRM for 3D Workspaces. Proc. Third Work. Algorithmic Found. Robot. Algorithmic Perspect. Algorithmic Perspect. 1998. pp. 155–168.
[34]. 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.
[35]. Hsu D., Jiang T., Reif J., Sun Z. The bridge test for sampling narrow passages with probabilistic roadmap planners. IEEE Int. Conf. Robot. Autom. 2003. Proceedings. ICRA '03. 2003. Vol. 3. pp. 4420–4426.

[36]. Nissoux C., Simeon T., Laumond J.-P. Visibility based probabilistic roadmaps. 1999 IEEE/RSJ Int. Conf. Intell. Robot. Syst. 1999. IROS '99. Proc. 1999. Vol. 3. pp. 1316–1321 vol.3.
[37]. Lien J.-M., Thomas S., Amato N.M. A general framework for sampling on the medial axis of the free space 2003 IEEE Int. Conf. Robot. Autom. (Cat. No.03CH37422). 2003. Vol. 3. pp. 4439–4444.
[38]. Berenson D., Srinivasa S., Ferguson D., Kuffner J.J. Manipulation Planning on Constraint Manifolds. Robotics and Automation, 2009. ICRA'09. 2009.
[39]. Berenson D., Srinivasa S., Kuffner J.J. Task Space Regions: A framework for pose-constrained manipulation planning. Int. J. Rob. Res. 2011. Vol. 30, № 12. pp. 1435–1460.
[40]. Redon S., Kheddar A., Coquillart S. Fast continuous collision detection between rigid bodies. Comput. Graph. Forum. 2002. Vol. 21, № 3. pp. 279–287.
[41]. Redon S., Lin M.C., Manocha D., Kim Y.J. Fast Continuous Collision Detection for Articulated Models. J. Comput. Inf. Sci. Eng. 2005. Vol. 5, № 2. pp. 126.
[42]. Devaurs D., Simeon T., Cortes J. Optimal Path Planning in Complex Cost Spaces with Sampling-Based Algorithms. IEEE Trans. Autom. Sci. Eng. 2016. Vol. 13, № 2. pp. 415–424.
[43]. Yershova A., LaValle S.M. Improving Motion Planning Alorithms by Efficient Nearest-Neighbor Searching. IEEE Trans. Robot. 2006. pp. 1–8.
[44]. Gipson B., Moll M., Kavraki L.E. Resolution Independent Density Estimation for motion planning in high-dimensional spaces. Proc. IEEE Int. Conf. Robot. Autom. 2013. pp. 2437–2443.
[45]. Fredriksson K. Geometric Near-neighbor Access Tree (GNAT) revisited. 2016.
[46]. Yianilos P.N. Data structures and algorithms for nearest neighbor search in general metric spaces. Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms. 1993. pp. 311–321.
[47]. Sanchez G., Latombe J.C. A single-query bi-directional probabilistic roadmap planner with lazy collision checking Robotics Research. 2003. pp. 403–417.
[48]. 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. 1996. Vol. 12, № 4. pp. 566–580.
[49]. 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.

Эволюционная разработка системы визуального планирования проектов на основе объектно-ориентированного каркаса

[1] *А.С. Аничкин <anton.anichkin@ispras.ru>*
[1,2] *С.В. Морозов <serg@ispras.ru>*
[1,3] *В.А. Семенов <sem@ispras.ru>*
[1,2] *О.А. Тарлапан <oleg@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991 ГСП-1, Москва, Ленинские горы*
[3] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*

Аннотация. В статье описывается практический опыт разработки перспективной системы визуального планирования проектов на основе объектно-ориентированного каркаса. Используемый каркас представляет собой систему классов и интерфейсов, предназначенных для программной реализации моделей, методов и приложений теории расписаний. Благодаря наличию готовых компонентов для решения типовых задач, а также предусмотренным механизмам их конфигурирования и расширения, создание приложений осуществляется относительно просто. Применение каркаса позволило реализовать в целевой системе необходимый функционал проектного планирования, а также обеспечить его последующее развитие путем обобщения условий задач и расширения арсенала алгоритмов, применяемых для их решения.

Ключевые слова: теория расписаний; календарно-сетевое планирование; программная инженерия; объектно-ориентированное программирование.

DOI: 10.15514/ISPRAS-2017-29(5)-12

Для цитирования: Аничкин А.С., Морозов С.В., Семенов В.А., Тарлапан О.А. Эволюционная разработка системы визуального планирования проектов на основе объектно-ориентированного каркаса. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 239-256. DOI: 10.15514/ISPRAS-2017-29(5)-12

1. Введение

Создание программных приложений теории расписаний представляет собой серьезную проблему, поскольку требует со стороны разработчиков как математической квалификации, необходимой для формализации прикладных задач и построения эффективных алгоритмов решения, так и знаний и опыта в области программной инженерии, необходимых для проектирования, реализации и интеграции сложных программных систем.

Универсальные математические библиотеки [1, 2, 3] позволяют решить некоторые типовые задачи теории расписаний известными алгоритмами, однако плохо приспособлены для прикладных задач, условия которых могут варьироваться в процессе эволюции целевого приложения. В подобных случаях вопросы выбора и реализации алгоритмов, релевантных вычислительной сложности решаемой прикладной задачи, становятся наиболее критичными [4].

С другой стороны, практика разработки программ заново для каждого приложения также является неприемлемой в силу значительных ресурсов, необходимых для реализации современных моделей и алгоритмов теории расписаний. Еще больших затрат требует конфигурирование и адаптация разработанных программ для согласованной работы в составе целевого приложения вместе с графическими и информационными компонентами.

Подход, предложенный в работе [5], предполагает создание и всестороннее применение единой программно-инструментальной среды для реализации моделей, методов и приложений теории расписаний. Данная среда сочетает в себе функции математической библиотеки и программного инструментария.

С одной стороны, такое сочетание предполагает наличие готовых к использованию программных компонентов для задания условий и решения типовых задач теории расписаний и, в частности, индустриально значимых задач высокой размерности в постановке проектного планирования Generally Constrained Project Scheduling Problem (GCPSP) [6, 7]. Поскольку многие задачи теории расписаний редуцируются к данной обобщенной постановке, среда предоставляет необходимые точные и приближенные средства решения.

С другой стороны, организация инструментальной среды в виде объектно-ориентированного каркаса обеспечивает повторное использование имеющихся компонентов при программной реализации новых моделей, методов и приложений теории расписаний при относительно низких затратах на доработку. При этом возможности развития, адаптации и конфигурации компонентов не препятствуют построению эффективных приложений, релевантных условиям и сложности решаемых прикладных задач.

Данные факторы предопределили выбор программно-инструментальной среды в качестве основного средства для построения системы визуального планирования проектов на основе ранее существовавшего приложения визуального моделирования Synchro [8]. Функции исходного приложения,

главным образом, ограничивались возможностями визуализации проектных планов и календарно-сетевых графиков на диаграмме Ганта и в окнах просмотра трехмерных сцен. Также приложение предоставляло средства верификации графиков, выявления пространственно-временных конфликтов и генерации сопутствующих отчетов и видеоматериалов.

Принцип работы Synchro состоял в консолидации календарно-сетевых графиков, импортируемых из традиционных систем управления проектами, таких как Oracle Primavera, Microsoft Project, Asta Powerproject, и трехмерных моделей, подготовленных в популярных CAD-системах, таких как AutoCAD, Revit, Sketchup, Microstation. В результате подобной консолидации формируется единая пространственно-временная модель проекта, которая затем может использоваться для визуализации, анализа и верификации. При обнаружении пространственно-временных конфликтов календарно-сетевой график проекта может быть скорректирован средствами приложения или с использованием сторонних систем в результате экспорта и импорта проектных данных.

Отсутствие средств построения расписаний являлось принципиальным недостатком исходного приложения, поскольку любая коррекция календарно-сетевого графика при обнаружении конфликтов могла потенциально нарушить его согласованность. В этом случае требовалось экспортировать календарно-сетевой график в стороннее приложение соответствующей функциональности, строить в нем согласованное расписание, а затем импортировать график в Synchro. Поскольку такая последовательность не гарантировала отсутствие новых пространственно-временных конфликтов в результирующем графике, требовались многократные действия. Наличие развитых средств построения расписаний превратило бы Synchro в полноценную систему управления проектами, причем с возможностями многофакторного планирования и визуального анализа.

В статье описывается практический опыт эволюционной разработки перспективной системы визуального планирования проектов на основе существующего приложения Synchro с использованием программно-инструментальной среды для реализации моделей, методов и приложений теории расписаний, который может быть востребован при создании других приложений. В разделе 2 кратко рассматриваются общие вопросы организации программно-инструментальной среды в виде объектно-ориентированного каркаса и методологии построения приложений теории расписаний на его основе. Процесс разработки целевой системы визуального планирования проектов обсуждается в разделе 3. Некоторые результаты вычислительных экспериментов представлены в разделе 4, в котором проводится сравнение показателей эффективности построенной системы с популярными системами управления проектами. В заключении подводятся

итоги проведенной работы и даются рекомендации по разработке и развитию приложений теории расписаний на основе каркаса.

2. Организация и назначение каркаса

Разработанный каркас представляет собой систему классов (в дальнейшем, учитывая его практическую реализацию на языке Си++, будем использовать принятые термины «класс», «конкретный класс», «абстрактный класс» и «интерфейс»). В организации каркаса (см. рисунок 1) выделим следующие группы или пакеты классов:

- пакет классов решателей (Solvers), реализующих общие алгоритмические схемы решения задач GCPSP (Schedulers), а также эвристики для поиска приближенных решений (Heuristics);
- пакет классов математических объектов (Mathematics), предназначенных для задания условий и получения результатов решения задач в обобщенной постановке GCPSP;
- пакет классов математической редукции (Reductions), обеспечивающих сведение прикладных задач составления расписаний к постановке GCPSP и соответствующую интерпретацию прикладных данных;
- пакет классов прикладных данных (Project Data), используемых для представления условий и результатов решения задач проектного планирования RCPSP в расширенных постановках.

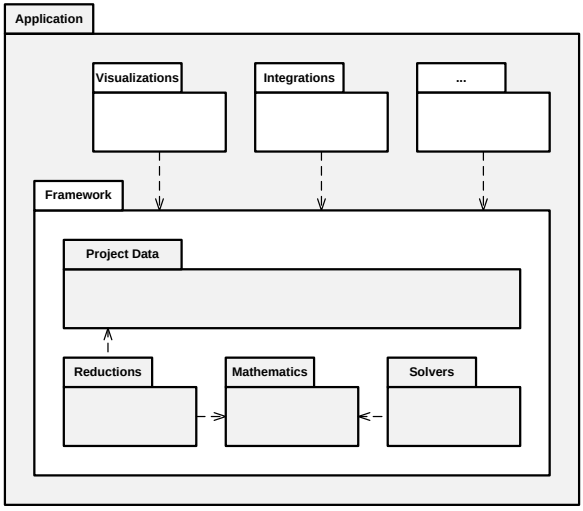


Рис. 1. Организация объектно-ориентированного каркаса для разработки приложений теории расписаний.
Fig. 1. Organization of the object-oriented framework for development of scheduling theory applications

Пакеты и классы каркаса подробно рассматриваются в [5]. Заметим, что часть из них реализуется в виде конкретных классов, допускающих непосредственное конструирование объектов. Другая часть представляет собой интерфейсы или абстрактные классы, которые по существу определяют точки расширения каркаса и позволяют предоставить их альтернативные реализации. Примечательно, что некоторые функции каркаса, в частности общие алгоритмические схемы, реализуются на уровне абстрактных классов без конкретизации частных условий решаемых задач и применяемых алгоритмов. Такой способ организации и реализации каркаса является рациональным с точки зрения повторного использования компонентов.

Сделаем некоторые комментарии, поясняющие представленную организацию и определяющие методологию построения целевых приложений на основе каркаса. Поскольку теория расписаний охватывает довольно много классов задач со своими алгоритмами, в состав каркаса включены, прежде всего, компоненты для решения обобщенных задач планирования в математически нейтральной постановке GCPSP, в которой определяются лишь тип целевой функции и вид алгебраических ограничений. Для того чтобы воспользоваться решателями, входящими в пакет Solvers, требуется предоставить собственные реализации данных абстракций, наследуя их от соответствующих интерфейсов пакета Mathematics. Тем не менее, поскольку многие задачи теории расписаний редуцируются к задачам проектного планирования RCPSP, более простым способом разработки приложений оказывается интерпретация условий исходных прикладных задач в терминах RCPSP, решение их имеющимися средствами и конвертация результатов в представление исходных задач. Пакет каркаса Project Data содержит необходимый набор классов для задания условий задач проектного планирования в расширенных постановках, что делает такую схему разработки приложений довольно простой. Более того, при создании систем проектного планирования и управления этот пакет определяет развитую модель данных, которая может быть положена в основу всей системы. В этом случае применимы и средства составления расписаний, поскольку пакет Reductions предоставляет необходимые реализации целевых функций и алгебраических ограничений, порождаемых постановками RCPSP.

Таким образом, разработка типового приложения с функциями проектного планирования может быть сведена к следующим работам:

- использование пакета Project Data для представления и хранения проектных данных, а также для задания условий задач планирования;
- конфигурирование классов решателей соответствующими целевыми функциями, ограничениями и эвристиками для эффективного решения задач планирования;
- разработка графического интерфейса пользователя целевого приложения для ввода, редактирования, визуализации проектных

данных, в том числе с использованием текстовых отчетов, графиков, диаграмм.

В частных случаях могут потребоваться дополнительные усилия для развития имеющихся пакетов:

- расширение пакета классов Project Data для представления специальных условий планирования и их сведения к расширенной постановке RCPSP;
- расширение пакета классов Reductions для приведения специальных условий планирования к обобщенной математической постановке GCPSP;
- расширение пакета классов Solvers для реализации новых алгоритмов и эвристик с учетом особенностей прикладных задач.

3. Разработка и развитие системы визуального планирования проектов

Разработанный каркас и связанная с ним методология разработки программных приложений теории расписаний были успешно апробированы в ходе построения системы визуального планирования проектов Synchro [8]. Система строилась на основе ранее существовавшего приложения визуального моделирования с основными функциями, перечисленными во введении. На рисунке 2 приведен снимок экрана, иллюстрирующий главное окно и основные элементы графического интерфейса пользователя системы Synchro, включая диаграмму Ганта и окна просмотра трехмерных сцен.

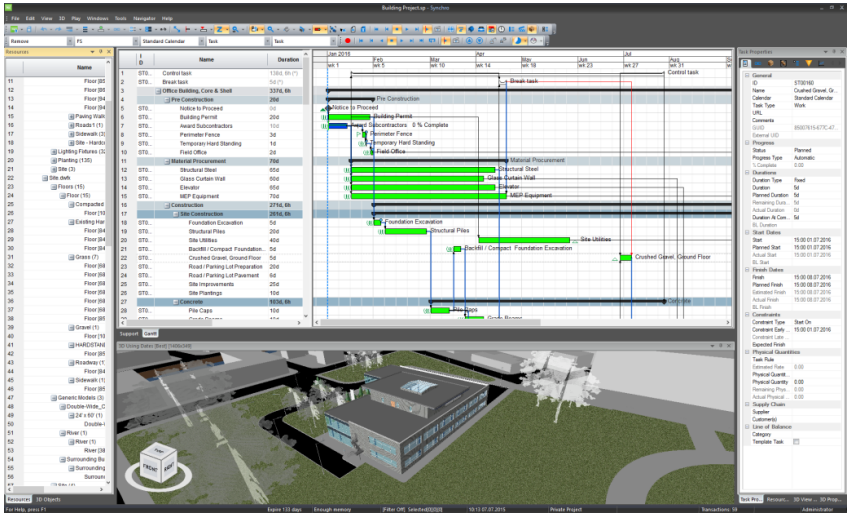


Рис. 2. Главное окно графического интерфейса пользователя системы Synchro.
Fig. 2. The main window of the graphical user interface of the Synchro system

Как отмечалось выше, отсутствие собственных средств построения расписаний являлось одним из главных недостатков приложения, поскольку ограничивало пользователя в разрешении обнаруженных конфликтов в результате согласованной коррекции календарно-сетевого графика. Наличие подобных средств превращало бы приложение в полноценную систему управления проектами, причем с возможностями визуального моделирования и многофакторного планирования.

Учет различных факторов осуществления проектной деятельности также являлся одним из принципиальных требований, предъявляемым к развиваемому приложению. При составлении расписаний должны учитываться не только временные условия, отношения предшествования, ресурсные ограничения и календарные правила, характерные для задач проектного планирования RCPSP, но и специфические требования пространственно-временной согласованности проектных работ, их финансового и логистического обеспечения. Данные требования важны для масштабных индустриальных программ, в которых риски технологических и организационных ошибок чрезвычайно высоки, а сроки и бюджеты жестко ограничены. Примерами специфических требований могут служить условия привлечения инвестиционных средств, ограничения по поставкам материалов, правила размещения и использования оборудования, особенности монтажа элементов конструкций возводимого сооружения, условия резервирования рабочих зон при организации проектных работ.

Заметим, что по мере развития целевой системы набор ограничений может пересматриваться в сторону обобщения и расширения, поэтому средства составления расписаний должны быть реализованы надлежащим образом, допуская поддержку новых типов ограничений без каких-либо серьезных изменений существующих программных компонентов.

Данные требования удалось удовлетворить с помощью представленного каркаса для построения приложений теории расписаний. Каркас не только предоставил готовые к использованию программные компоненты для задания условий и решения задач проектного планирования, но и обеспечил возможность добавления и поддержки новых видов ограничений в рамках описанной выше методологии.

Рассмотрим процесс эволюционной разработки системы визуального планирования проектов Synchro более подробно.

Так как каркас внедрялся в существующее приложение, а не использовался для создания новой системы, возникла необходимость развития унаследованной модели данных. Заметим, что приложение Synchro реализовано в рамках объектно-ориентированной парадигмы на языке Си++ и использует объектно-ориентированную модель данных для представления проекта, работ, связей, календарей, ресурсов, трехмерных объектов сцены. Упрощенная реализация данных концептов с использованием ограниченного

набора атрибутов диктовалась функциями визуального моделирования, для которых достаточно иметь данные, подлежащие отрисовке на диаграмме Ганта или в окнах просмотра трехмерных сцен. В частности, в классе работ определялись атрибуты имени, длительности, времен начала и конца, однако отсутствовали структуры индивидуального использования ресурсов. Класс календарей реализовывал простую модель рабочей недели с исключениями, но не применял более общую модель регулярных и исключительных правил. Класс ресурсов использовался, главным образом, в качестве делегата трехмерных объектов сцены, однако не предусматривал задание ресурсных атрибутов, таких как тип, статус возобновимости и количество доступных единиц.

В силу указанных причин было принято решение заменить соответствующие классы приложения каркасными реализациями. Такая замена не являлась трудоемкой и в большинстве случаев заключалась в коррекции некоторых названий методов доступа к атрибутам классов. Для поддержания прежней функциональности визуального моделирования и реализации новых функций проектного планирования, класс проектов Project, агрегирующий трехмерные данные (объекты, текстуры, камеры, анимации), был унаследован от соответствующего класса каркаса, агрегирующего необходимые проектные данные. Аналогичная схема консолидации применена в классе ресурсов, что позволило объединить данные, необходимые для ресурсного планирования, и данные, необходимые для их связи с трехмерными объектами.

При описанной схеме наследования в приложении Synchro стало возможным составлять расписания непосредственно средствами каркаса. В частности, появилась возможность расчета критического пути и оценки временных резервов в постановке задачи проектного планирования без ресурсов. Также стали доступны алгоритмы составления расписаний в расширенных постановках. Важно, что для подобных целей не потребовалась разработка дополнительных конвертеров исходных данных и результатов расчетов, поскольку алгоритмические реализации Solvers рассчитаны для работы с проектными данными при надлежащей конфигурации компонентов Project Data, Reductions и Mathematics.

Для задания дополнительных параметров планирования, выполнения алгоритмов и визуализации расчетов потребовалось расширить графический интерфейс пользователя. В частности, в окно параметров проекта добавлены элементы для установки планируемых дат начала и завершения проекта, необходимых для составления расписания и расчета временных резервов работ. В окно параметров работ добавлены элементы для установки временных ограничений, количеств используемых ресурсов, задержек между взаимосвязанными работами. В диалоге задания опций предусмотрен раздел настроек для алгоритмов составления расписаний, который позволяет, например, установить правила выравнивания дат работ, пороговые значения временных резервов для интерпретации работ как критических и

субкритических, способы разрешения нарушенных ограничений в процессе актуализации проектных данных. Наконец, в панель инструментов добавлена кнопка для запуска пересчета расписания. На рисунке 3 представлен экранный снимок приложения Synchro с описанными развитыми функциями проектного планирования.

Заметим, что ревизии подверглись и средства визуализации результатов расчетов. В диаграмме Ганта появилась возможность отобразить критический путь, наложенные временные ограничения, а также фактические взаимосвязи, определяющие даты планируемых работ. В таблицу проектных работ (слева от диаграммы Ганта) добавлены столбцы со значениями временных резервов, которые в сочетании с универсальными средствами сортировки и фильтрации позволяют пользователю выделить относительно небольшое, но исключительно важное подмножество работ, влияющих на ход и успешное завершение всего проекта к планируемой дате. Данная возможность особенно важна для управления масштабными индустриальными программами, в которых детальный контроль выполнения каждой индивидуальной работы не представляется возможным.

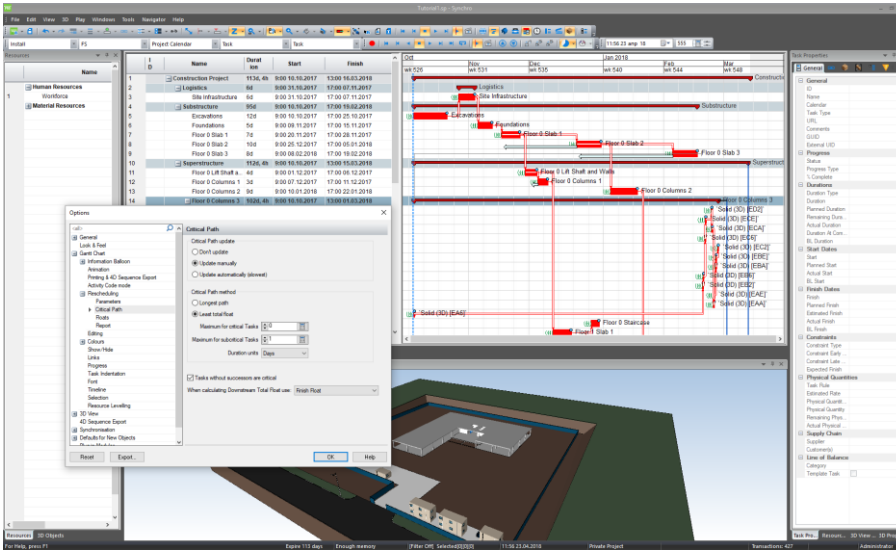


Рис. 3. Главное окно графического интерфейса пользователя системы Synchro с развитыми возможностями проектного планирования.
Fig. 3. The main window of the graphical user interface of the Synchro system with advanced features for project planning

Поскольку планирование ресурсов является одним из ключевых элементов проектного управления, в графический интерфейс пользователя были добавлены графики потребления ресурсов, которые позволяют пользователю

идентифицировать конфликты в календарно-сетевом графике, связанные с превышением доступного количества единиц того или иного ресурса. Сгенерированные приложением расписания являются согласованными в том смысле, что при корректно заданных условиях наложенные ограничения, в том числе и ресурсные, оказываются удовлетворенными. Однако при задании или коррекции календарно-сетевого графика непосредственно пользователем часть ограничений может быть нарушена и необходимы средства их идентификации. На рисунке 4 приведён пример графиков потребления ресурсов по времени, причем слева представлены кривые, устанавливающие превышение доступного количества ресурсов в графике, а справа — кривые полностью согласованного графика.

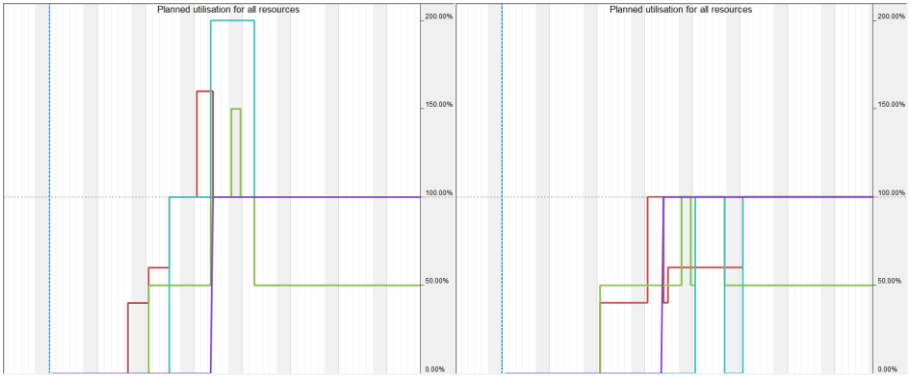


Рис. 4. Примеры графиков потребления ресурсов в несогласованном (слева) и согласованном (справа) календарно-сетевом графике.
Fig. 4. Examples of resource utilisation charts in the uncoordinated (left) and coordinated (right) calendar-network schedule

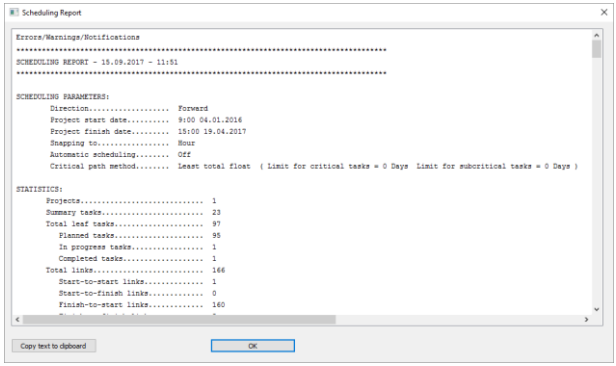


Рис. 5. Пример отчёта о построении календарно-сетевого графика.
Fig. 5. An example calendar-network schedule construction report

Наконец, для верификации условий задач проектного планирования и генерации отчетов был разработан и добавлен в систему соответствующий инструмент. На рисунке 5 показано окно для просмотра текстовых отчетов, в которое выводятся общие статистические данные о проекте (количество работ, ресурсов, взаимосвязей, календарей и т.п.), параметры проектного планирования, а также некорректно заданные условия, связанные с наличием циклических взаимосвязей между работами, превышением потребляемых ресурсов индивидуальными работами, переопределенными временными условиями.

Перечисленные выше функции календарно-сетевого планирования являются традиционными для систем управления проектами. Использование компонентов каркаса лишь упростило и ускорило их реализацию в составе целевой системы Synchro, которая в результате объединила в себе функционал визуального моделирования и проектного планирования.

Вместе с тем, с консолидацией проектных данных появились качественно новые возможности для решения задач планирования в расширенных постановках с учетом пространственных факторов. Такими факторами могут быть рабочие зоны, которые в случае пересечения или перегруженности порождают конфликты на проектной площадке и мешают осуществлению проектных работ [9]. Другими пространственными факторами, влияющими на реализуемость графика, могут быть коллизии размещаемого оборудования, конфликтность планируемых путей доставки материалов, отсутствие опор или подвесов при монтаже конструкций [10]. Математическая формализация задачи проектного планирования с учетом перечисленных пространственных факторов, а также метод её решения приводятся в работе [7].

Важно, что данные пространственные факторы должны учитываться вместе с другими видами ограничений в рамках общей вычислительной схемы составления расписания. Введенный класс задач проектного планирования, а также применяемая схема последовательной диспетчеризации позволяют осуществить это в результате задания специальных условий и приведения их к обобщенной математической постановке GCPSP. С этой целью в пакет каркаса Project Data были добавлены необходимые классы представления рабочих зон и трехмерных объектов, а в реализации класса Project учтены новые пространственные условия. Для данных условий были разработаны и включены в пакет Reductions соответствующие классы приведения к алгебраическим ограничениям общего вида в рамках постановки GCPSP. Для приоритезации работ с назначенными рабочими зонами была разработана новая эвристика, учитывающая требуемый объем рабочей зоны, а ее реализация включена в состав пакета Solvers. Заметим, что для подобных целей можно было бы использовать и стандартную эвристику, основанную на уровнях потребления ресурсов, однако интерпретация объема рабочей зоны в качестве ресурса не всегда корректна [11].

Рассматривались возможности использования других вычислительных схем и алгоритмов проектного планирования, в частности, параллельной схемы генерации расписаний, одно- и многопроходных схем, схем сэмплирования [12]. Несмотря на их поддержку каркасом, в целевой системе был использован компонент, реализующий алгоритм последовательной диспетчеризации с расширенным набором эвристик. В силу невысокой вычислительной сложности данный алгоритм хорошо себя зарекомендовал при решении широкого класса задач проектного планирования высокой размерности [7], что являлось принципиальным требованием к разрабатываемой целевой системе. Поддержка возможностей выбора специализированных алгоритмов для решения частных классов задач была принята нецелесообразной.

Заметим, что использование объектно-ориентированного каркаса значительно упростило развитие приложения. Сам каркас предоставил готовые классы проектных данных и решатели задач построения расписаний в обобщенной постановке GCPSP. Адаптация каркаса к соответствующим прикладным задачам свелась лишь к реализации классов в пакете Reductions для поддержки специальных типов условий, возникающих в задачах проектного планирования. Трудозатраты (в строках программного кода) реализации данных классов приведены в таблице 1.

Табл. 1. Оценка трудозатрат развития приложения.
Table 1. An estimate of labor costs of application development

Тип ограничения	Кол-во строк кода	Доля от общего кол-ва строк
Учет предшествования работ	211	0,16%
Учет календарей	374	0,28%
Явные временные ограничения	170	0,13%
Обязательные временные ограничения	153	0,11%
Выравнивание начала или конца работ	97	0,07%
Учет использования ресурсов	669	0,49%
Учет использования рабочих зон	2033	1,5%

Таким образом, реализация каждого из типов ограничений заключалась в создании одного нового класса, описываемого в среднем 530 строками программного кода, что составляет 0,39% от общего количества строк кода инструментальной среды.

4. Сравнительный анализ производительности системы

Достигнутая общность в программной реализации алгоритмов составления расписаний, а также принятая параметризация условий задач проектного планирования в расширенных постановках могут существенно влиять на

производительность целевой системы. Поэтому важным было убедиться, что негативный эффект от обобщенной реализации программных компонентов каркаса не столь существенен и разработанная система визуального планирования Synchro может конкурировать с популярными системами управления проектами при решении индустриально значимых задач высокой размерности.

Оценка производительности систем производилась по критерию затраченного процессорного времени на составление расписания в зависимости от размерности задачи (числа переменных и ограничений). В качестве тестов использовались синтезированные наборы проектных данных с разным количеством работ и связей между ними. Поскольку синтезированные планы имели фиксированные соотношения глубины и кратности связей, размерность задач в каждом тестовом наборе определялась лишь числом работ. Планы реальных индустриальных проектов использовались лишь для подтверждения полученных результатов. Тестирование проводилось на одном и том же компьютере с типовой конфигурацией (процессор: Intel Core i7, количество ядер: 4, объём оперативной памяти: 16 ГБ, операционная система: MS Windows 10 x64), поэтому для проводимого сравнительного анализа систем значение имели лишь относительные показатели производительности.

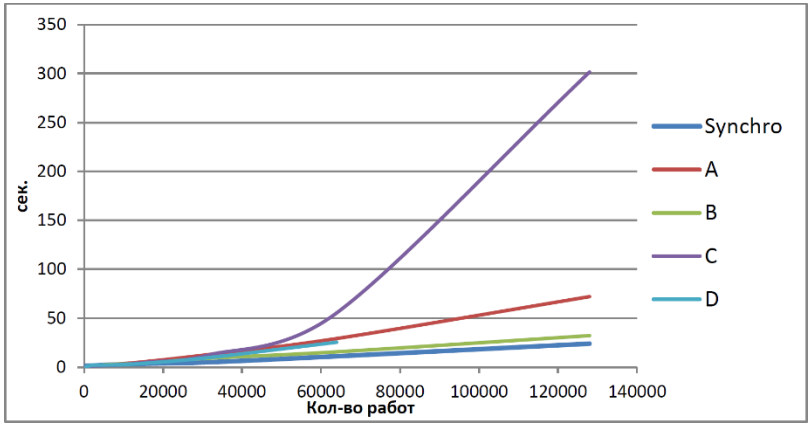


Рис. 6. Зависимость процессорного времени составления расписания в постановке CPM от размерности задачи.
Fig. 6. The dependence of computation time of schedule construction in CPM formulation on the problem dimensionality

Первая серия испытаний предназначалась для анализа производительности систем в зависимости от размерности задач планирования в постановке расчета критического пути (CPM). Для каждого теста в наборе составлялось расписание с помощью пяти различных систем, включая Synchro, и

замерялось затраченное процессорное время. Поскольку используемые в экспериментах системы управления проектами являются коммерческими и их упоминание в результатах сравнительного анализа запрещается в соответствии с лицензионными соглашениями, мы присвоили данным системам символические имена A, B, C, D. Результаты испытаний качественно не менялись при переходе к новому тестовому набору, поэтому на рисунке 6 приведены типовые графики зависимости процессорного времени от количества работ в проектном плане. Заметим, что кривая производительности для системы D обрывается при достижении размерности тестовых данных, составляющей 64000 работ. Это связано с невозможностью данной системы обрабатывать проектные планы большей размерности. С ростом размерности задач планирования система Synchro демонстрировала наилучшие результаты.

Вторая серия испытаний предназначалась для аналогичных целей, но расписание строилось в постановке проектного планирования с учетом ресурсов RCPSP. Поскольку системы реализуют приближенные алгоритмы, имеющие разную вычислительную сложность и обеспечивающие разное качество найденных решений, сравнительный анализ производительности здесь не всегда корректен.

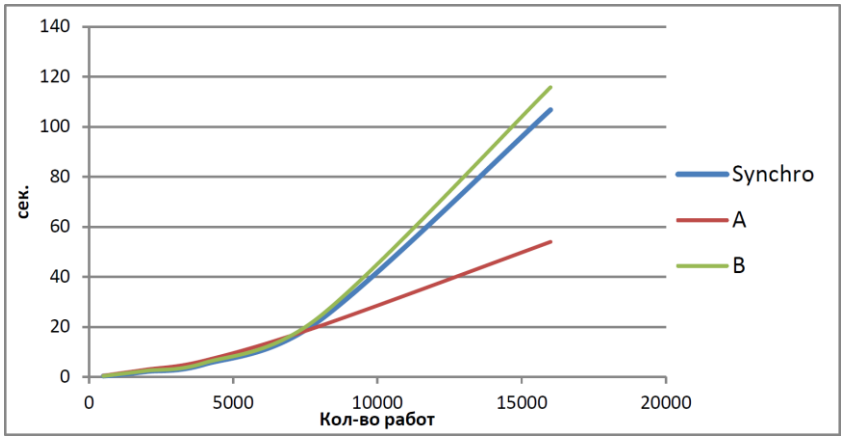


Рис. 7. Зависимость времени поиска расписания от размерности плана (с учётом ресурсов).
Fig. 7. The dependence of schedule search time on the dimensionality of the plan (considering resources)

На рисунке 7 представлены характерные результаты сравнения производительности популярных систем управления проектами. Системы C и D не участвовали в данном испытании по причине отсутствия в них соответствующих алгоритмических средств. Более тщательный анализ показывает, что алгоритм, реализованный в системе A, демонстрирует

линейную сложность от размерности задачи, а в двух других системах, включая Synchro — квадратичную. Это объясняется тем, что первый алгоритм строит согласованное расписание путем решения задачи в постановке расчета критического пути, а затем — последовательно выравнивает уровни потребления ресурсов. Это может быть выполнено за линейное время, однако качество найденного решения в общем случае будет хуже вследствие возможного увеличения длительности проектов за счет выравнивания ресурсов. Более качественный результат (меньшая длительность планируемого проекта) достигается при поиске оптимального расписания приближенными методами, например, с помощью реализованного в объектно-ориентированном каркасе классического алгоритма последовательной диспетчеризации, который имеет квадратичную сложность, поскольку выбор приоритетной работы на каждом шаге требует применения эвристик для всех работ в активном списке. По сравнению с системой B, также реализующей алгоритм поиска расписания квадратичной сложности, разработанная система демонстрирует лучшую производительность.

Тем самым, предоставляя широкие функциональные возможности, разработанная система Synchro не уступает по производительности популярным системам управления проектами при решении задач планирования в аналогичных постановках и аналогичными методами.

5. Заключение

Таким образом, применение объектно-ориентированного каркаса позволило не только ускорить разработку системы визуального планирования проектов, но и обеспечило ее динамичное функциональное развитие, причем с относительно небольшими затратами. Важно, что сами классы каркаса не претерпели каких-либо изменений, а для расширения функций системы потребовалась лишь реализация специализированных классов с предопределенными интерфейсами. Главным образом, доработка была обусловлена необходимостью обобщения условий решаемых задач и настройки алгоритмических компонентов. Эволюционные возможности среды позволили со временем поддержать развитые наборы временных условий, взаимосвязей задач, моделей привлечения ресурсов, пространственных ограничений и, тем самым, обеспечить конкурентные преимущества целевой системы над традиционными системами управления проектами.

Проведённые серии экспериментов показали, что достигнутая общность в реализации программных компонентов не препятствует высокой производительности, которая оказалась сопоставима с аналогичными показателями популярных систем управления проектами.

Успешная апробация каркаса в ходе разработки, сопровождения, развития и коммерческого использования целевой системы Synchro на протяжении десяти лет показала правильность принятых проектных решений, а также

перспективность использования каркаса для эволюционной разработки приложений теории расписаний.

Список литературы

[1]. Kolisch R., Sprecher A. PSPLIB — A project scheduling library. European Journal of Operational Research, vol. 96, issue 1, 1997, pp. 205-216.

[2]. Kolisch R., Hartmann S. Experimental Investigation of Heuristics for Resource-Constrained Project Scheduling: An Update. European Journal of Operational Research, vol. 174, 2006, pp. 23-37.

[3]. Lemmen R. Modeling Resource Alternatives in Project Scheduling. Munich University of Applied Sciences, March 29, 2005.

[4]. Лазарев А. А., Гафаров Е. Р. Теория расписаний. Задачи и алгоритмы. МГУ им. М. В. Ломоносова, Москва, 2011 г., 222 с.

[5]. Аничкин А.С., Семенов В.А. Объектно-ориентированный каркас для программной реализации приложений теории расписаний. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 247-296. DOI: 10.15514/ISPRAS-2017-29(3)-14.

[6]. Аничкин А.С., Семенов В.А. Об обобщенной математической постановке задач проектного планирования. ИТНОУ: Информационные технологии в науке, образовании и управлении (под ред. Глориозова Е.Л.), вып. 2, 2017 г., стр. 74-86.

[7]. Аничкин А.С., Семенов В.А. Математическая формализация задач проектного планирования в расширенной постановке. Труды ИСП РАН, том 29, вып. 2, 2017 г., стр. 231-256. DOI: 10.15514/ISPRAS-2017-29(2)-9.

[8]. Интернет-ресурс: «Synchro Software». Официальный Интернет-сайт продукта Synchro, http://synchroltd.com

[9]. Аничкин А.С., Казаков К.А., Семенов В.А. Календарно-сетевое планирование индустриальных проектов с учетом перегруженности рабочих зон. Информационные и математические технологии в науке и управлении: Труды XX Байкальской Всероссийской конференции, в 3 томах (под ред. Массель Л.В.), том 1, 2015 г., стр. 7-14. Иркутск, ИСЭМ СО РАН.

[10]. Семенов В.А., Аничкин А.С., Морозов С.В., Тарлапан О.А., Золотов В.А. Комплексный метод составления расписаний для сложных индустриальных программ с учетом пространственно-временных ограничений. Труды ИСП РАН (под ред. Иванникова В.П.), том 26, вып. 1, 2014 г., стр. 457-482. ISSN 2220-6426.

[11]. Yeoh K.W., Chua David K.H. Mitigating Workspace Congestion: A Genetic Algorithm Approach. Proceedings of the EPPM 2012 Conference, 2012, pp. 107-118.

[12]. Kolisch R., Hartmann S. Heuristic algorithms for solving the resource-constrained project scheduling problem — Classification and computational analysis. Chapter in the book «Handbook on recent advances in project scheduling», ed. Weglarz J., 1999, pp. 147-178.

Аничкин А.С., Морозов С.В., Семенов В.А., Тарлапан О.А. Эволюционная разработка системы визуального планирования проектов на основе объектно-ориентированного каркаса. *Труды ИСП РАН*, 2017 г., том 29, вып. 5, стр. 239-256

Evolutionary development of a visual planning system using object-oriented framework

[1] *A.S. Anichkin <anton.anichkin@ispras.ru>*
[1,2] *S.V. Morozov <serg@ispras.ru>*
[1,3] *V.A. Semenov <sem@ispras.ru>*
[1,2] *O.A. Tarlapan <oleg@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*
[3] *Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*

Abstract. The article describes the practical experience of developing a prospective visual planning system based on an object-oriented framework. The used framework is a system of classes and interfaces intended for software implementation of models, methods and applications of scheduling theory. Due to the availability of ready-to-use components for the solution of typical problems as well as the mechanisms for their configuration and extension, development of applications becomes a relatively simple process. The application of the framework allows to implement the necessary functional for project planning in the target system as well as to provide its subsequent evolution by generalizing the statements of the problems and expanding the arsenal of algorithms used to solve them. The described experience can be claimed when building other applications of scheduling theory. The paper discusses the general issues of organizing a software toolkit in the form of the object-oriented framework, a methodology for creation of scheduling theory applications on its basis, as well as a process of developing a target system for visual planning of projects based on the methodology and the framework. Results of computational experiments comparing the performance of the developed system with some popular project management systems are also presented in the paper. Recommendations on the development and evolution of scheduling theory applications based on the framework are summarized in conclusion.

Keywords: scheduling theory; project planning and scheduling; software engineering; object-oriented programming.

DOI: 10.15514/ISPRAS-2017-29(5)-12

For citation: Anichkin A.S., Morozov S.V., Semenov V.A., Tarlapan O.A. Evolutionary development of a visual planning system using object-oriented framework. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 239-256 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-12

References

[1]. Kolisch R., Sprecher A. PSPLIB — A project scheduling library. European Journal of Operational Research, vol. 96, issue 1, 1997, pp. 205-216.

[2]. Kolisch R., Hartmann S. Experimental Investigation of Heuristics for Resource-Constrained Project Scheduling: An Update. European Journal of Operational Research, vol. 174, 2006, pp. 23-37.

[3]. Lemmen R. Modeling Resource Alternatives in Project Scheduling. Munich University of Applied Sciences, March 29, 2005.

[4]. Lazarev A. A., Gafarov E. R. Scheduling theory. Tasks and algorithms. Lomonosov Moscow State University, Moscow, 2011, 222 p. (in Russian).

[5]. Anichkin A.S., Semenov V.A. Object-oriented framework for software development of scheduling applications. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017, pp. 247-296 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-14.

[6]. Anichkin A.S., Semenov V.A. About the generalized mathematical formulation of project scheduling problems. ITNOU: Informacionnye tehnologii v nauke, obrazovanii i upravlenii/ITSEM: Information technologies in science, education and management (ed. Gloriozov E.L.), issue 2, 2017, pp. 74-86 (in Russian).

[7]. Anichkin A.S., Semenov V.A. Mathematical formalization of project scheduling problems. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 2, 2017, pp. 231-256 (in Russian). DOI: 10.15514/ISPRAS-2017-29(2)-9.

[8]. Internet: «Synchro Software». Official website of the product Synchro, http://synchroltd.com

[9]. Anichkin A.S., Kazakov K.A., Semenov V.A. Calendar-network planning of industrial projects taking into account the congestion of workspaces. Proceedings of the 20th Baikal All-Russian Conference, in 3 volumes (ed. Massel L.V.), vol. 1, 2015, pp. 7-14. Irkutsk, MESI SB RAS.

[10]. Semenov V.A., Anichkin A.S., Morozov S.V., Tarlapan O.A., Zolotov V.A. Effective method for scheduling complex industrial programs under spatio-temporal constraints. Trudy ISP RAN/Proc. ISP RAS (ed. Ivannikov V.P.), vol. 26, issue 1, 2014, pp. 457-482. ISSN 2220-6426.

[11]. Yeoh K.W., Chua David K.H. Mitigating Workspace Congestion: A Genetic Algorithm Approach. Proceedings of the EPPM 2012 Conference, 2012, pp. 107-118.

[12]. Kolisch R., Hartmann S. Heuristic algorithms for solving the resource-constrained project scheduling problem — Classification and computational analysis. Chapter in the book «Handbook on recent advances in project scheduling», ed. Weglarz J., 1999, pp. 147-178.

Моделирование программно-аппаратных систем и анализ их безопасности

[1] *С.А. Зеленова <sophia@ispras.ru>*
[1,2] *С.В. Зеленов <zelenov@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Национальный исследовательский университет Высшая школа экономики, 101000, Москва, ул. Мясницкая, д. 20.*

Аннотация. В данной статье демонстрируется целесообразность применения языка моделирования программно-аппаратных систем AADL и его расширения Error Model Annex для описания требований безопасности проектируемой системы. Наиболее важным аспектом здесь является возможность описания требований безопасности в терминах, используемых в теории безопасности, таких, как марковские цепи или логико-вероятностные функции, т.к. за годы развития теории было накоплено большое количество весьма полезных результатов. Различные подходы к оценке безопасности систем не конкурируют, но дополняют друг друга, так что наличие некоторой универсальности в описании требований безопасности является весьма ценным качеством.

Ключевые слова: моделирование программно-аппаратных систем; безопасность; анализ дерева неисправностей; анализ видов и последствий отказов; марковский анализ.

DOI: 10.15514/ISPRAS-2017-29(5)-13

Для цитирования: Зеленов С.В., Зеленова С.А. Моделирование программно-аппаратных систем и анализ их безопасности. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 257-282. DOI: 10.15514/ISPRAS-2017-29(5)-13

1. Введение

Бурное развитие техники в XX веке поставило вопросы защиты человека от всевозможных проблем (аварий, техногенных катастроф) связанных со сложными техническими системами. Увеличение количества техники, усложнение как внутреннего строения, так и внешних связей технических систем затрудняет поиск проблемных «тонких» мест и требует автоматизированных методов оценки надежности и безопасности системы. Кроме того, в силу тех же причин отказы и аварии часто влекут большие экономические и людские потери. Все это привело к возникновению

совершенно новых дисциплин — теории надежности, теории безопасности, теории живучести [3], [8], [15], [21], [22], [26].

Термины надежности, безопасности, живучести хотя и близки, однако имеют большие смысловые отличия:

- *надежность* — способность системы в нормальных условиях работать безотказно;
- *безопасность* — способность системы функционировать не переходя в опасное состояние;
- *живучесть* — способность системы функционировать под воздействием поражающих факторов, возникающих в результате каких-либо экстраординарных событий (взрыв, пожар, землятрясение и т. п.).

Из этих определений видно, что понятия надежности и безопасности структурно очень похожи, различие между ними состоит в том, что в основе понятия надежности лежит понятие отказа, а в основе понятия безопасности — понятие опасного состояния. Опыт показывает, что различия в определениях существенны для реальных систем, так, надежная система может быть и опасной, и неживучей. Корабль, который плавает не один десяток лет, проходит все ремонты и технически исправен, может затонуть при хорошей погоде за несколько минут при получении пробоины — это пример надежной, но неживучей системы (крушение теплохода «Адмирал Нахимов» в 1986 году [6]). Другой пример — склад боеприпасов: трудно представить, какой отказ в нем может произойти в нормальных условиях, т.е. эта система надежна, но между тем, очевидно, что такая система становится опасной в случае взрыва боеприпасов.

В нашей стране теории надежности, безопасности и живучести сложно структурированных систем получили свое развитие в послевоенные годы. Исследованием данных вопросов занимались как инженеры (Дедков В.К., Рябинин И.А., Северцев Н.А., Ильичев А.В., и др), так и математики, (Гнеденко Б.В., Беляев Ю.К., Коваленко И.Н., Каштанов В.А., Соловьев А.Д. и др.) [2], [3], [4], [7]. [8], [9], [10], [26]. В настоящей работе мы упомянем две методологии, использующиеся в данных теориях (см. раздел 2).

Размеры и сложность современных систем таковы, что их ручной анализ был бы чрезвычайно трудоемким и дорогостоящим, а в некоторых случаях и просто невозможным. Таким образом, привлечение автоматизированных средств и, как следствие, разработка формальных моделей системы, становится насущной необходимостью.

Одним из современных средств описания архитектуры программно-аппаратных систем является Architecture Analysis & Design Language (AADL) [18], [33]. Для спецификации модели сбоев системы предназначено специальное расширение Error Model Annex [34] языка AADL.

Разработчики стандарта AADL предоставляют инструмент OSATE [31], который позволяет редактировать и анализировать AADL-модели, в частности выполнять отдельные аспекты анализа рисков [17]. Однако, следует отметить, что OSATE предоставляет лишь некоторую базовую поддержку языка, а в плане анализа моделей в основном полагается на сторонние инструменты, такие как OpenFTA [30] для анализа дерева неисправностей и PRISM [24] для марковского анализа. OSATE производит разбор AADL-модели и подготовку необходимых входных данных для запуска внешнего анализатора. В ряде случаев функциональность анализа рисков в OSATE имеет серьезные ограничения, например, обрабатывается только один уровень вложенности компонентов модели, а также отсутствует поддержка комбинаций различных сбоев в компоненте.

В настоящей статье мы представляем алгоритмы полнофункционального, лишенного указанных недостатков, анализа рисков: анализ дерева неисправностей, анализ видов и последствий отказов, марковский анализ [32]. Алгоритмы реализованы в разрабатываемом в ИСП РАН инструменте MASIW [1], [23], [29] для поддержки автоматизированного проектирования и анализа программно-аппаратных систем.

Основная часть статьи построена следующим образом. В разделе 2 дан обзор методов оценки безопасности. В разделе 3 представлены возможности языка AADL для описания архитектуры системы и для спецификации модели сбоев системы. В разделе 4 приводятся алгоритмы анализа рисков на основе описаний AADL. В разделе 5 рассмотрен пример моделирования и анализа безопасности системы передачи данных. В разделе 6 подводятся итоги статьи.

2. Элементы теории безопасности

2.1 Логико-вероятностный анализ

Основным методом логико-вероятностного анализа (ЛВА) [7], [9], [10], [13], [19], [25] является сопряжение методов математической логики и теории вероятности.

Общая схема может быть описана следующим образом. Исходя из поставленной цели (изучение надежности, безопасности) для исследуемой системы определяются высказывания об элементах системы и вероятность того, что данное высказывание верно (неверно). Например, для имеющегося соединительного провода высказывание может звучать как «провод цел» или «провод оборван». Вероятность истинности такого высказывания берется из соображений технического и опытного характера. На этом этапе первую часть анализа составляет структуризация системы, приводящая к выделению отдельных элементов, высказывания о которых приобретают характер терминальных. Так, можно говорить об отказе или опасном состоянии целого блока не вдаваясь в подробности его устройства, если имеются основания рассматривать данный блок как единое целое. Важно заметить, что об одном

элементе системы можно сделать несколько высказываний («провод оборван», «провод перегорел», «провод перекушен кусачками злоумышленником»), у каждого из которых будет своя вероятность истинности.

Далее, исходя из общей структуры системы, терминальные высказывания об отдельных ее элементах организуются в логическую формулу, которая уже будет говорить об отказе/опасном состоянии системы в целом. То есть в математическом виде прописывается логика возникновения отказа/опасного состояния. Важно понимать взаимосвязи терминальных элементов в смысле зависимости или независимости их вероятностных характеристик. Не всегда можно считать вероятности истинности терминальных элементов независимыми, это создает дополнительные трудности.

На следующем этапе логическая формула отказа/опасного состояния системы должна быть преобразована в вероятностную функцию. Факторы, усложняющие этот процесс: немонотонность логической функции, повторение аргументов (терминальных элементов) и др. Для упрощения обычно используются различные алгоритмы ортогонализации.

Логико-вероятностный анализ широко используется в самолетостроении, часто в виде различных упрощений и модификаций. Примером такой модификации является анализ дерева неисправностей [12], [16], [27], когда логическая функция строится для какого-то одного события верхнего уровня. Отталкиваясь от этого события, исследователь-аналитик постепенно спускается на нижележащие уровни, выискивая всевозможные причины, приводящие к событиям верхних уровней, и, таким образом, строя искомую логическую функцию.

Достоинства ЛВА:

- простота и доступность для понимания;
- наглядность;
- не слишком большой объем;
- возможность использования различных уровней абстракции;
- возможность автоматизации значительной части анализа при наличии формальной модели системы.

Недостатки ЛВА:

- отсутствие понятия времени в описании логической функции и, как следствие этого, необходимость строгого регламентирования всех событий и трудности описания специфических ситуаций, зависящих от времени (повторные отказы, перемежающиеся отказы, наложение одних событий на другие, самовосстанавливающиеся системы);
- трудности построения вероятностной функции в некоторых случаях (повторные терминальные символы, немонотонная логическая функция).

2.2 Вероятностные автоматы и марковские цепи

Другой подход к оценке надежности/безопасности/живучести системы — использование теории марковских процессов [11]. Здесь необходимо ввести одно понятие из теории конечных автоматов.

Вероятностный или *стохастический автомат* — это конечный автомат, в котором нет входных и выходных символов, а переходы между состояниями осуществляются с заданной вероятностью.

Исследуемая на предмет безопасности система может быть описана в виде вероятностного автомата, содержащего особые состояния, соответствующие отказам (или опасным состояниям). Вероятность перехода в такое состояние — это вероятность возникновения отказа/опасного состояния.

Во многих случаях можно считать, что будущее технической системы определяется ее текущим состоянием, так что построенный вероятностный автомат является марковской цепью и к нему применимы все результаты теории марковских цепей.

Марковский анализ [11], [14], [20] целевой системы дает достаточно полную информацию о её надежности и безопасности. Основная проблема в том, что полученная модель может быть очень объемной.

Достоинства марковского анализа:

- наличие понятия временной последовательности событий в самой модели и, вследствие этого, возможность оценки поведения системы в разные моменты времени;
- простота описания циклических событий;
- возможность автоматизации значительной части анализа при наличии формальной модели системы.

Недостатки марковского анализа:

- большой объем модели;
- достаточно сложный математический аппарат.

Как видно, ни марковский анализ, ни логико-вероятностный анализ не имеют неоспоримых преимуществ друг перед другом, и должны использоваться как взаимно дополняющие.

2.3 Специфические методы анализа

Кроме общих методов анализа безопасности, таких как, ЛВА и марковский анализ, имеется большое число специфических методов, направленных на анализ конкретных инженерных решений. Эти методы несомненно полезны, так как хорошо учитывают специфику ситуации, но они не обладают общностью, что сужает область их применения.

3. Язык моделирования архитектуры AADL

Язык AADL (Architecture Analysis and Design Language) [33] в настоящее время де-факто является стандартом для моделирования архитектуры и требований по безопасности в аэрокосмической области. В данном разделе мы продемонстрируем, как на базе выразительных возможностей AADL можно автоматизировать различные виды анализа безопасности, и проиллюстрируем процесс моделирования архитектуры и требований по безопасности на примере фрагмента реальной системы.

3.1 Моделирование архитектуры

Язык AADL предназначен для формального описания архитектуры программно-аппаратных систем. В AADL имеются встроенные средства для описания следующих сущностей и отношений между ними:

- на аппаратном уровне: элементарное устройство, процессор, память, порт, шина;
- на функциональном уровне: процесс, подпрограмма, соединение (канал передачи информации);
- есть возможность описать систему на некотором виртуальном абстрактном уровне: виртуальный процессор, виртуальная шина, поток исполнения.

AADL-описание отражает внутреннюю структуру, подчиненную строгой иерархии: система состоит из подчиненных подсистем и элементов, у этих подсистем могут быть свои подсистемы и элементы и т. д. Кроме того, функциональные компоненты могут быть привязаны к аппаратным напрямую или через несколько виртуальных уровней, что позволяет рассматривать систему на нескольких уровнях абстракции.

3.2 Описание требований по безопасности

Для задания требований безопасности разработчиками языка AADL было создано специальное расширение этого языка — Error Model Annex (EM) [34]. Для каждого компонента системы, описанного на языке AADL, можно указать особые требования, накладываемые на этот компонент, которые могут интерпретироваться как требования безопасности. Error Model Annex включает следующие изобразительные возможности:

- для каждого компонента может быть задан конечный автомат, состояния которого — это штатные и нештатные (опасные/отказные) состояния данного компонента;
- влияние сбоев компонентов системы на другие компоненты описывается посредством указания логических условий распространения ошибок между различными типами компонентов в различных состояниях;

- переходы между состояниями компонента осуществляются при выполнении некоторых логических условий, в зависимости от наличия сбоев — как внутренних, так и «наведенных», т.е. распространившихся в данный компонент извне;
- для внутренних сбоев задаются вероятности их возникновения;
- модель ошибок компонента, составленного из подкомпонентов, описывается как композиция моделей ошибок подкомпонентов, при этом для составления этой композиции используются логические операции.

Таким образом, Error Model Annex использует для описания требований безопасности и термины логико-вероятностного анализа, и термины марковского анализа.

3.3 Математическая модель требований по безопасности

Пусть C — множество всех компонентов данной модели. Рассмотрим отдельный компонент *c*. Связанные с компонентом *c* объекты (множества, функции) будем записывать с префиксом «c.». Напомним, что для множества M запись 2^M означает множество всех подмножеств множества M.

Пусть c.S — множество состояний компонента *c*, c.E — множество внутренних сбоев, c.Q — множество сбоев, приходящих в *c* извне, c.R — множество сбоев, распространяющихся из *c* вовне. Пусть задана функция переходов между состояниями c.F: $c.S \times 2^{c.E} \times 2^{c.Q} \to c.S$, а также функция распространения сбоев в данном состоянии c.G: $c.S \times 2^{c.E} \times 2^{c.Q} \to 2^{c.R}$. Пусть также задана функция st: $C \to \{ c.S \}$, которая для данного компонента возвращает его текущее состояние. Кроме того, пусть для всех внутренних сбоев задана функция p вероятности их возникновения.

В EM-описании функции c.F и c.G имеют следующий вид. Переход из состояния $c.s_i$ в состояние $c.s_j$ осуществляется, если выполняется некоторое заданное логическое условие $c.f_{i,j}(c.e_1, \dots , c.e_L, c.q_1, \dots , c.q_M)$ от внутренних и внешних сбоев. Аналогично, в состоянии $c.s_j$ осуществляется распространение сбоя $c.r_k$, если выполняется некоторое заданное логическое условие $c.g_{j,k}(c.e_1, \dots , c.e_L, c.q_1, \dots , c.q_M)$ от внутренних и внешних сбоев. Терминальные высказывания относительно сбоев в условиях $c.f_{i,j}$ и $c.g_{j,k}$ имеют вид «произошел сбой».

Если компонент *c* является композицией компонентов $d_1, \dots , d_n$, то его состояния задаются функцией c.H: $d_1.S, \dots , d_n.S \to c.S$. В EM-описании функция c.H имеет следующий вид. Компонент *c* находится в состоянии $c.s_j$, если выполняется некоторое заданное логическое условие $c.h_j(d_1.s_1, \dots , d_n.s_n)$. Терминальные высказывания относительно состояний подкомпонентов d_i в условиях $c.h_j$ имеют вид «d_i находится в состоянии $d_i.s_i$», т.е. «$st(d_i) = d_i.s_i$».

4. Анализ рисков на основе описаний AADL

4.1 Анализ дерева неисправностей

4.1.1 Построение дерева неисправностей

Заметим, что каждое логическое условие можно однозначно представить в виде дерева, где листьям соответствуют терминальные операнды, а внутренним узлам — логические операции. Будем называть такое дерево LF-деревом. Верно и обратное: для любого LF-дерева однозначно восстанавливается соответствующее логическое условие.

Дерево неисправностей — это LF-дерево, соответствующее условию перехода некоторого компонента c в состояние $c.s_j$.

Шаг построения дерева неисправностей начинается с рассмотрения целевого состояния $c.s_j$ компонента *c*. На одном шаге построения требуется выявить локальные причины перехода компонента *c* в состояние $c.s_j$. Пусть в строящемся дереве есть узел A, соответствующий условию «$st(c) = c.s_j$».

Если EM-описание для $c.s_j$ имеет вид композиции, то происходит поиск соответствующего условия $c.h_j$ от состояний подкомпонентов. Дерево t логического условия $c.h_j$ вставляется в дерево неисправностей, так что корень t становится узлом A. Далее построение дерева неисправностей продолжается для фигурирующих в $c.h_j$ терминальных высказываний относительно состояний подкомпонентов.

Если EM-описание для $c.s_j$ имеет вид функции перехода из другого состояния, то ищутся все соответствующие исходные состояния $c.s_i$ и условия $c.f_{i,j}$. Далее строится дерево t' для логического условия

$$OR_i \ (st(c) = c.s_i) \text{ AND } c.f_{i,j} ,$$

где «OR_i» означает оператор «ИЛИ» по всем индексам *i*, и вставляется в дерево неисправностей, так что его корень становится узлом A.

Далее построение дерева неисправностей продолжается для терминальных высказываний:

- для условия «$st(c) = c.s_i$» производится следующий шаг построения дерева;
- для условия «в компоненте *c* произошел внутренний сбой c.e» в дереве оставляется соответствующий листовой узел;
- для условия «в компоненте *c* произошел внешний сбой c.q» происходит поиск источников этого внешнего сбоя.

Поиск источников внешнего сбоя c.q для компонента c происходит так. Для всех компонентов b_i, которые распространяют сбой $b_i.r_k = c.q$ в компонент *c*,

ищутся все соответствующие состояния $b_i.s_j$ и условия $b_i.g_{j,k}$. Далее строится дерево t" для логического условия

$$OR_i \ OR_j \ OR_k \ (st(b_i) = b_i.s_j) \text{ AND } b_i.g_{j,k}$$

и вставляется в дерево неисправностей, так что его корень заменяет узел для рассматриваемого высказывания «в компоненте *c* произошел внешний сбой c.q». Построение дерева неисправностей для терминальных высказываний дерева t" продолжается аналогично.

В результате в листьях дерева будут содержаться только условия вида «в компоненте c_i произошел внутренний сбой $c_i.e_{l_i}$»

4.1.2 Минимальные сечения

После построения дерева неисправностей можно вычислить вероятность перехода исходного компонента *c* в состояние $c.s_j$. Для этого необходимо сперва упростить логическую формулу, соответствующую данному дереву, и привести ее к некоторому каноническому виду. Прежде чем приступить к описанию этого канонического вида, напомним следующее определение.

Минимальным сечением называется такая наименьшая комбинация внутренних сбоев, что если все эти сбои произойдут, то это повлечет возникновение в дереве события верхнего уровня (т.е. переход компонента *c* в состояние $c.s_j$).

Вычисление минимальных сечений для данной логической формулы происходит путем приведения формулы к дизъюнктивной форме с использованием булевских законов дистрибутивности и поглощения. В результате формула примет вид дизъюнкции конъюнкций, причем каждая конъюнкция будет соответствовать какому-то минимальному сечению.

Пусть все внутренние сбои являются независимыми событиями. Тогда вероятность любой их конъюнкции равна произведению вероятностей множителей. Что касается вероятности дизъюнкции, то для ее точного вычисления необходимо применять формулу включений-исключений. Однако, поскольку в реальности вероятности внутренних сбоев достаточно малы (по меньшей мере порядка $10^{-4}..10^{-6}$), членами высших порядков в формуле включений-исключений пренебрегают.

Таким образом, если логическая формула для данного дерева неисправностей приведена к дизъюнкции конъюнкций, соответствующих минимальным сечениям, то вероятность события верхнего уровня (приближенно) равна соответствующей сумме произведений вероятностей внутренних сбоев из минимальных сечений.

4.1.3 Ранжирование сбоев

Все внутренние сбои, встречающиеся в данном дереве неисправностей, можно ранжировать по величине их вклада в вероятность возникновения в дереве события верхнего уровня (т.е. перехода компонента *c* в состояние $c.s_j$).

Ранжирование первичных событий в соответствии с их значимостью в смысле наступления события верхнего уровня может производиться разными методами [27]. Пусть S – событие верхнего уровня, E – первичное событие, $\bar{E}$ – отрицание E, p – функция вероятности событий. Напомним, что если A и B – два события, то p(A|B) – это вероятность наступления события A при условии, что наступило событие B.

Мера значимости «*Risk Achievement Worth*» – «стоимость возрастания риска»: p(S|E) / p(S). Эта величина показывает, насколько увеличивается риск при условии, что событие E происходит постоянно (т. е. соответствующий компонент абсолютно не надежен). Это индикатор, насколько важно поддерживать текущий уровень надежности соответствующего компонента.

Мера значимости «*Risk Reduction Worth*» – «стоимость уменьшения риска»: p(S) / p(S|$\bar{E}$). Эта величина показывает, максимальное возможное уменьшение риска, которое можно ожидать при увеличении надежности соответствующего компонента (вплоть до его абсолютной надежности, когда событие E никогда не происходит).

Мера значимости *по Бинбауму* (Birnbaum): $\partial p(S)/\partial p(E) = p(S|E) - p(S|\bar{E})$. Эта величина показывает, в какой степени изменение надежности E влияет на повышение надежности S.

Мера значимости *по Фусселу-Весли* (Fussel-Vesely): $(p(S) - p(S|\bar{E})) / p(S)$. Эта величина показывает степень критичности события E, т.е. относительную величину вклада события E в величину вероятности наступления S.

Мера значимости «*важность диагностики*»: $p(E|S) = p(E)\cdot p(S|E) / p(S)$. Эта величина показывает вероятность сбоя E при условии сбоя S, т.е. насколько приоритетно проверять, был ли сбой E, при возникновении сбоя S.

4.2 Анализ видов и последствий отказов

Пусть модель состоит из компонентов c_1, ... , c_N. Состояние всей модели полностью определяется состояниями всех ее компонентов: $c_1.S \times ... \times c_N.S$.

Рассмотрим отказ следующего вида. Пусть каждый из компонентов c_i находится в некотором состоянии $c_i.s_i$, и пусть в каждом из компонентов c_i произошли некоторые внутренние сбои $\{c_i.e_{i,j}\}$.

В результате этого отказа компоненты модели могут поменять свое состояние и начать распространять какие-то сбои вовне. Далее, в условиях прихода этих внешних сбоев в зависимые компоненты, компоненты модели опять могут поменять свое состояние и начать распространять еще какие-то сбои вовне. И так далее, модель будет эволюционировать до тех пор, пока не достигнет некоторого стабильного состояния.

Требуется найти, в каком состоянии модель стабилизируется. Это состояние модели будет выражать последствия исходного отказа.

Рассмотрим подробно один шаг эволюции модели. В общем случае, в начале шага каждый компонент c_i находится в некотором состоянии $c_i.s_i$ и имеет сбои: внутренние $\{c_i.e_{i,j}\}$ и внешние $\{c_i.q_{i,j}\}$. Применяя для соответствующих компонентов сперва функцию $c_i.F$, а затем функцию $c_i.H$, найдем, в какое новое состояние $c_i.s_i'$ при текущих условиях перейдет каждый компонент. Затем, применяя для каждого компонента функцию $c_i.G$, найдем, какие сбои $\{c_i.r_{i,j}\}$ станет распространять компонент c_i в состоянии $c_i.s_i'$ при условии сбоев $\{c_i.e_{i,j}\}$ и $\{c_i.q_{i,j}\}$. После этого, определим компоненты-приемники найденных сбоев $\{r_{i,j}\}$ и таким образом получим, новый набор внешних сбоев $\{c_i.q_{i,j}'\}$.

Как видим, шаг эволюции модели зависит не только от состояний компонентов, но также и от наборов сбоев $\{c_i.e_{i,j}\}$ и $\{c_i.q_{i,j}\}$. Таким образом, стабильным можно считать такое состояние модели, когда в результате шага эволюции в компонентах не меняются как состояния, так и наборы сбоев.

Поскольку все заданные множества ($c_i.S$, $c_i.E$, $c_i.Q$) являются конечными, процесс эволюции на некотором шаге придет в такое состояние, в котором модель уже находилась ранее. После этого модель начнет эволюционировать по циклу. В том случае, если этот цикл является петлей из одного состояния, модель стабилизируется в этом состоянии. Если же этот цикл содержит более одного состояния модели, можно говорить, что модель задана некорректно. Вопрос изучения критериев корректности модели выходит за рамки настоящей работы.

4.3 Построение марковской цепи

Давно разработан классический математический аппарат [11], [14], [20] для проведения марковского анализа по заданной марковской цепи. Здесь мы опишем алгоритм построения марковской цепи на основе EM-описания.

Для данной модели, состоящей из компонентов c_1, ... , c_N, рассмотрим вероятностный автомат. Состояниями этого автомата являются кортежи состояний всех компонентов $s_1 \times ... \times s_N$. Переход автомата в другое состояние происходит при возникновении некоторого набора внутренних сбоев в компонентах. Для данного состояния s автомата и данного набора внутренних сбоев $\{e_{i,j}\}$ можно запустить процесс анализа последствий этого отказа модели и найти состояние s', в котором модель стабилизируется. В этом случае будем говорить, что в вероятностном автомате имеется переход $s \rightarrow s'$ с вероятностью $\prod_{i,j} p(e_{i,j})$.

Если в модели имеется N компонентов и L различных внутренних сбоев, причем каждый компонент может находиться не менее чем в двух состояниях, то вероятностный автомат будет содержать не менее чем 2^N состояний, и из каждого состояния будет выходить 2^L переходов.

Однако, на практике не все состояния вероятностного автомата являются достижимыми из некоторого начального состояния. Таким образом, если строить вероятностный автомат постепенно, по мере построения переходов из уже достигнутых состояний, то количество полученных состояний будет существенно меньше экспоненты.

Кроме того, в реальных системах отдельные сбои имеют довольно низкую вероятность, порядка 10^{-4} .. 10^{-12}. Соответственно, вероятность одновременного возникновения нескольких сбоев является исчезающе малой величиной. Таким образом, на практике наборы сбоев $\{e_{i,j}\}$ можно ограничить парами или тройками, а значит количество переходов из каждого состояния вероятностного автомата будет ограничено полиномом степени 2 или 3.

Более подробное описание алгоритмов построения марковской цепи и проведения марковского анализа на основе EM-описания можно найти в [5].

5. Пример моделирования и анализа системы

Процесс моделирования системы с использованием поддержки требований безопасности включает следующие шаги:

- анализ общей ситуации
- формулирование различных проблем на уровне системы
- выделение компонентов, ответственных за ситуацию
- классификация и построение описания проблем, связанных с отдельными компонентами виртуального уровня
- анализ связей между виртуальным и физическим уровнем
- классификация и построение описания проблем, связанных с отдельными компонентами физического уровня

Проиллюстрируем процесс моделирования и анализа безопасности на примере сети передачи данных между датчиками, электроприводами, вычислительные модули, дисплеями и т.п. внутри авиационного судна. Абоненты сети, которым требуется передавать друг другу информацию, связаны между собой проводами. Несколько десятилетий назад, когда электрооборудования на самолетах было немного, для повышения безопасности проектировщики руководствовались правилом: каждому каналу передачи информации — свой провод. В настоящее время количество абонентов сети, а с ними и количество каналов выросло настолько, что буквальное следование этому принципу привело бы к непомерному росту суммарного веса необходимых проводов. Решением проблемы стало использование одних и тех же проводов несколькими каналами. Уменьшение количества проводов потребовало более сложной топологии сети, в частности применения коммутаторов (switch) в местах разветвления.

Один из подходов, реализующих указанный принцип, называется Avionics Full-Duplex Switched Ethernet (AFDX) [28] и в настоящее время де-факто является стандартом в этой области.

Канал передачи информации в AFDX-сети не является, как раньше, единоличным владельцем провода, однако на логическом уровне каждый канал моделируется так называемым виртуальным каналом (virtual link). Таким образом, проектировщик мысленно продолжает жить в старом подходе («один канал — один провод»), оперируя виртуальными каналами, а в реальности каждый виртуальный канал использует какой-то свой физический маршрут, состоящий из некоторого набора проводов и коммутаторов сети (см. рис.1).

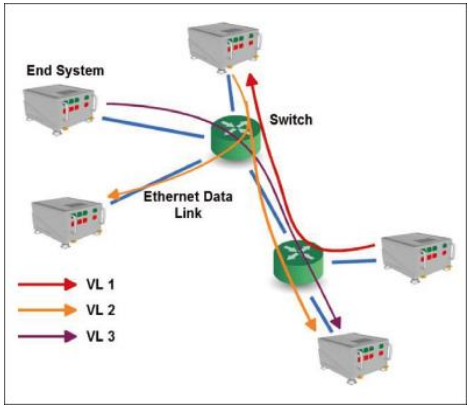


Рис. 1. Виртуальные каналы

Fig. 1. Virtual links

Для обеспечения безопасности в AFDX применяется резервирование: для каждого виртуального канала используются два независимых физических маршрута, по каждому из которых передаются идентичные копии сообщения.

5.1 Моделируемая система

Рассмотрим процесс моделирования и анализа сети передачи данных.

Пусть абонентами сети являются:

- датчик (sensor),
- вычислительный модуль (computer),
- электропривод (actuator),
- дисплей (monitor).

Перечислим логические соединения между абонентами (см. рис. 2):

- датчик отправляет свои текущие показания на вычислительный модуль,

- вычислительный модуль анализирует полученные показания датчика и отправляет управляющие команды на электропривод,
- также вычислительный модуль отправляет текущую информацию о параметрах полета на дисплей,
- кроме того, электропривод в случае возникновения в нем аварийной ситуации отправляет предупреждающий сигнал на дисплей.

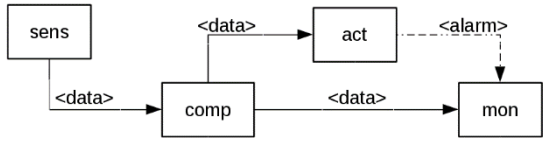


Рис. 2. Функциональная модель

Fig. 2. Functional model

Моделирование сети AFDX состоит в описании виртуальных каналов. Каждому логическому соединению соответствует один виртуальный канал в сети AFDX, а для обеспечения резервирования для каждого вируального канала определены два виртуальных маршрута «a» и «b», каждому из которых соответствует свой набор аппаратных компонентов системы.

Пусть абоненты нашей сети выполняются каждый на своем процессоре, и каждый процессор использует для выхода в AFDX-сеть свою сетевую карту. В местах разветвления информационных потоков в сети используются коммутаторы. На рис.3 представлена аппаратная часть архитектуры системы без резервирования. Собственно сеть (провода и коммутаторы) на рисунке выделена пунктирной рамкой. Для обеспечения резервирования в системе мы организуем два идентичных экземпляра представленной на рисунке сети.

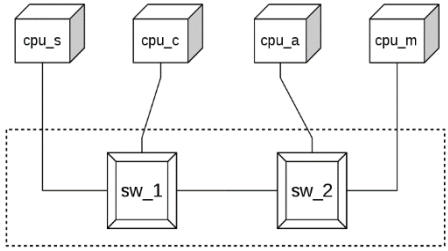


Рис. 3. Аппаратная часть: процессоры и сеть

Fig. 3. Hardware: processors and net

Архитектура системы в целом показана на рис.4 и состоит из трех частей. В верхней части представлена функциональная модель с абонентами сети и логическими соединениями. В средней части представлена модель сети AFDX,

которая содержит четыре элемента — виртуальные каналы для каждого логического соединения. В нижней части представлена аппаратная модель HW, которая содержит процессоры, два экземпляра сети net_a и net_b (чтобы не загромождать рисунок, внутреннее содержимое сетей скрыто), а также энергосистема power. Энергосистема состоит из нескольких батарей, каждая из которых обеспечивает энергией какие-то два устройства (процессоры и комммутаторы).

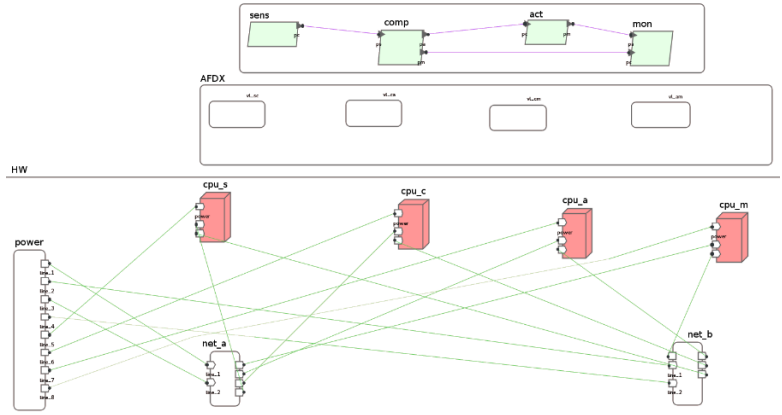


Рис. 4. Модель системы

Fig. 4. System model

На рис.5 подробно показана привязка одного логического соединения к аппаратным компонентам посредством виртуального канала и виртуальных путей.

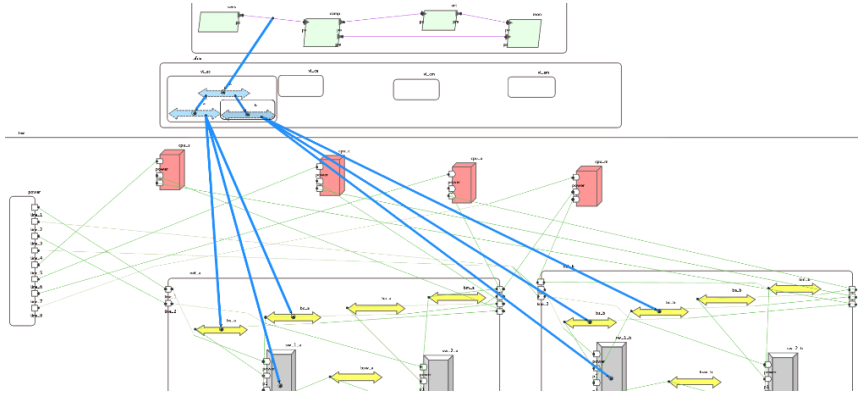


Рис. 5. Привязка логического соединения к аппаратным компонентам

Fig. 5. Binding of a connection to hardware components

5.2 Спецификация опасных состояний компонентов

Изучим возможные повреждения отдельных компонентов и их влияние на работоспособность других компонентов.

Прежде всего сформулируем возможные проблемы, которые могут представлять опасность для системы в целом:

- отказ какого-то процесса-абонента,
- потеря информации для управления самолетом,
- потеря предупреждающей сигнализации.

Далее необходимо выразить перечисленные функциональные опасности через комбинации отказов компонентов на аппаратном уровне.

Отказ процесса-абонента происходит в случае отказа соответствующего процессора.

Потеря информации на логическом соединении происходит в случае если одновременно отказали оба виртуальных пути соответствующего виртуального канала, поскольку виртуальные пути в рамках одного виртуального канала включены параллельно. Отказ виртуального пути происходит в случае отказа хотя бы одного из аппаратных компонентов, к которым привязан этот виртуальный путь, поскольку эти аппаратные компоненты в рамках виртуального пути включены последовательно.

Теперь перейдем собственно к описанию повреждений отдельных компонентов.

Изначально, каждый компонент находится в рабочем (Operational) состоянии. Если в компоненте или в его окружении возникнут какие-то повреждения, то данный компонент может частично или полностью терять свою работоспособность.

Для системы в целом и для процессов-абонентов введем три опасных состояния, соответствующих перечисленным выше функциональным опасностям: Failed_NoService, Failed_AppMsg, Failed_AlarmMsg.

Для остальных компонентов (виртуальные каналы и пути, процессоры, коммутаторы, батареи) введем одно опасное состояние Failed, когда компонент полностью теряет способность выполнять заданное назначение.

EM-описание условия нахождения каждого компонента в конкретном состоянии может иметь одну из двух форм:

- явное описание переходов между состояниями;
- неявное описание переходов в зависимости от состояний подкомпонентов.

Для иллюстрации неявной формы рассмотрим нашу систему в целом. В соответствии с декомпозицией, система переходит, например, в состояние Failed_NoService, если хотя бы один из процессов-абонентов перешел в состояние Failed_NoService.

В EM-описании состояние компонента-композиции задается в виде логической функции от состояний подкомпонентов:

```
composite error behavior
      states
             [   sens.Failed_NoService
              or comp.Failed_NoService
              or act.Failed_NoService
              or mon.Failed_NoService
             ]-> Failed_NoService;
```

Явная форма задания состояний компонента включает следующие описания:

- внутренние сбои (events);
- собственно переходы между состояниями компонента (transitions);
- влияние на соседние компоненты (propagations).

Для примера рассмотрим EM-описание процессора. Внутренним сбоем здесь является поломка всего процессора:

```
events
      Failure : error event;
```

Далее опишем переходы между состояниями процессора. В результате поломки процессора происходит его переход в состояние Failed. В это состояние он также переходит и при отказе энергосистемы:

```
transitions
      Operational -[ Failure ]-> Failed;
      Operational -[ power{NoPower} ]-> Failed;
```

Наконец, опишем влияние процессора в различных его состояниях на соседние компоненты. Отказ процессора повлияет на работоспособность привязанного к нему процесса:

```
propagations
      Failed -[]-> bindings( NoService );
```

5.3 Анализ безопасности

Проведем анализ безопасности описанной выше системы.

Пусть интенсивности внутренних сбоев компонентов равны значениям из следующей таблицы:

Компонент	**Сбой**	**Интенсивность**
Коммутатор	Отказ	$2.5 \cdot 10^{-5}$
Процессор	Отказ	$2.5 \cdot 10^{-5}$
Батарея	Разряжена	$1.35 \cdot 10^{-5}$

Результаты анализа деревьев неисправностей для сформулированных в начале функциональных опасностей системы приведены в следующей таблице:

Функциональная опасность	**Логическое выражение (по дереву неисправностей)**	**Логическое выражение (по минимальным сечениям)**	**Вероятность**
Потеря предупреждающей сигнализации	[⩾2] ((sw_2_a.Failure ⋁ battery_2.Depleted) , (sw_2_b.Failure ⋁ battery_2.Depleted))	(battery_2.Depleted + (sw_2_a.Failure * sw_2_b.Failure))	0.0000135
Потеря информации для управления	([⩾2] ((sw_1_b.Failure ⋁ battery_1.Depleted) , (sw_1_a.Failure ⋁ battery_1.Depl ...	(battery_1.Depleted + battery_2.Depleted + (sw_1_a.Failure * sw_1_b.Failure) + (sw_1_a.Failu ...	0.000027
Отказ	((cpu_s.Failure ⋁ battery_3.Depleted) ⋁ cpu_c.Failure ⋁ (cpu_a.Failure ⋁ battery_4.Depleted) ⋁ cpu_m.Failure)	(battery_3.Depleted + battery_4.Depleted + cpu_a.Failure + cpu_c.Failure + cpu_m.Failure + cpu_s.Failure)	0.000127

Как видно, для опасностей «Потеря предупреждающей сигнализации» и «Потеря информации для управления» вероятности имеют порядок 10^{-5}, хотя из соображений резервирования виртуальных каналов эти вероятности должны иметь порядок $10^{-9}..10^{-10}$ (поскольку вероятности внутренних сбоев имеют порядок 10^{-5}).

Меры значимости внутренних сбоев для опасности «Потеря предупреждающей сигнализации» приведены в следующей таблице:

ID	**Description**	**RAW**	**RRW**	**Birnbaum**	**Fussel-Vesely**	**Diagnostic**
battery_2. Depleted	Батарея разряжена	74071.15	21601.4	1.0	0.9999537	0.9999537
sw_2_a. Failure	Отказ	2.851	1.000046	0.0000245	0.0000463	0.0000713
sw_2_b. Failure	Отказ	2.851	1.000046	0.0000245	0.0000463	0.0000713

Как видно, сбой «Батарея разряжена» имеет чрезвычайно большую значимость для данной функциональной опасности. Анализируя логическое выражение по минимальным сечениям для опасности «Потеря предупреждающей сигнализации», видно, что одно из минимальных сечений состоит из единственного сбоя «Батарея разряжена». Это подтверждает и анализ видов и последствий отказов:

Item(s)	Initial failure mode(s)	End effect	Sev	P
sw_2_a	Отказ	a.[propagation]	9	0.000125
		sw_2_a.Failed	9	0.000125
sw_2_b	Отказ	b.[propagation]	9	0.000125
		sw_2_b.Failed	9	0.000125
battery_2	Батарея разряжена	Model. Failed_AppMsg	10	0.0000675

Из этой таблицы следует, что внутренний сбой в коммутаторе приводит лишь к отказу в коммутаторе и к распространению ошибки на уровне виртуального пути («a» или «b»), а внутренний сбой батареи battery_2 приводит к опасному состоянию Failed_AppMsg («Потеря предупреждающей сигнализации») уровня всей системы с максимальным значением «Важность» (Severity) = 10.

Причина такого положения дел кроется в следующей архитектурной ошибке, которая сводит на нет резервирование сети. А именно, коммутаторы sw_1_a и sw_1_b (резервирующие друг друга) подключены к одной батарее battery_1 (см. рис.6), а коммутаторы sw_2_a и sw_2_b (тоже резервирующие друг друга) подключены к одной батарее battery_2.

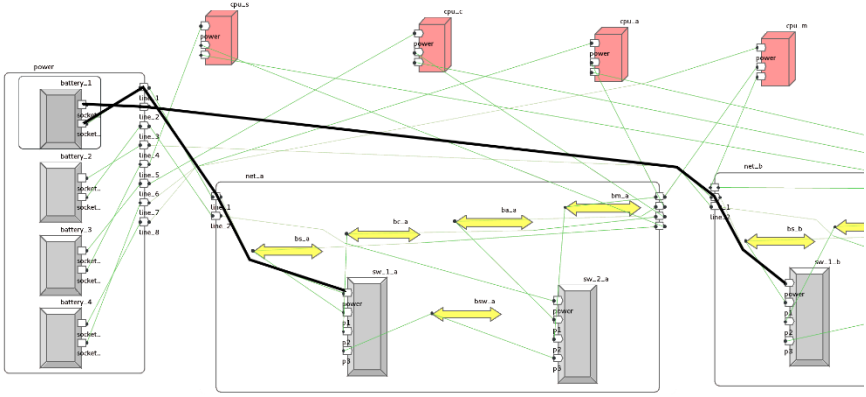


Рис. 6. Два резервирующих друг друга коммутатора подключены к одной батарее

Fig. 6. Two corresponding redundant switches connected to one battery

Для исправления этой ошибки требуется переподключить коммутаторы к батареям, так чтобы резервирующие друг друга коммутаторы были подключены к разным батареям, например, как это изображено на рис.7.

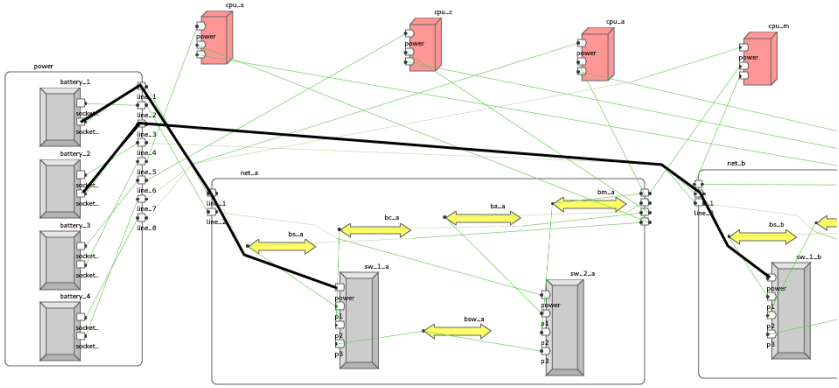


Рис. 7. Правильное подключение коммутаторов к батареям

Fig. 6. Good connecting of switches to batteries

После такой корректировки модели результаты анализа дерева неисправностей будут такими:

Функциональная опасность	**Логическое выражение (по дереву неисправностей)**	**Логическое выражение (по минимальным сечениям)**	**Вероятность**
Потеря предупреждающей сигнализации	[⩾2] ((sw_2_b.Failure ∨ battery_2.Depleted) , (sw_2_a.Failure ∨ battery_1.Depleted))	((battery_1.Depleted * battery_2.Depleted) + (battery_1.Depleted * sw_2_b.Failure) + (battery_2.Depleted * sw_2_a.Failure) + (sw_2_a.Failure * sw_2_b.Failure))	1.4822E-9
Потеря информации для управления	([⩾2] ((sw_1_a.Failure ∨ battery_1.Depleted) , (sw_1_b.Failure	((battery_1.Depleted * battery_2.Depleted) + (battery_1.Depleted * sw_1_b.Failure)	4.03216E-9

	∨ battery_2.Depl ...	+ (...	
Отказ	((cpu_s.Failure ∨ battery_3.Depleted) ∨ cpu_c.Failure ∨ (cpu_a.Failure ∨ battery_4.Depleted) ∨ cpu_m.Failure)	(battery_3.Depleted + battery_4.Depleted + cpu_a.Failure + cpu_c.Failure + cpu_m.Failure + cpu_s.Failure)	0.000127

Теперь для опасностей «Потеря предупреждающей сигнализации» и «Потеря информации для управления» вероятности имеют порядок 10^{-9}, а все минимальные сечения имеют по два элемента. Меры значимости внутренних сбоев для опасности «Потеря предупреждающей сигнализации» таковы:

ID	Description	RAW	RRW	Birnbaum	Fussel-Vesely	Diagnostic
battery_1. Depleted	Батарея разряжена	25974.95	1.54	0.0000385	0.35	0.35
battery_2. Depleted	Батарея разряжена	25974.95	1.54	0.0000385	0.35	0.35
sw_2_b. Failure	Отказ	25974.65	2.85	0.0000385	0.65	0.65
sw_2_a. Failure	Отказ	25974.65	2.85	0.0000385	0.65	0.65

Значимость сбоя «Батарея разряжена» стала сравнима со значимостью отказа коммутатора.

Результаты анализа видов и последствий отказов таковы:

Item(s)	Initial failure mode(s)	End effect	Sev	P
sw_2_a	Отказ	a.[propagation]	9	0.000125
		sw_2_a.Failed	9	0.000125
sw_2_b	Отказ	b.[propagation]	9	0.000125
		sw_2_b.Failed	9	0.000125
battery_1	Батарея разряжена	a.[propagation]	9	0.0000675
		sw_1_a.Failed	9	0.0000675
		sw_2_a.Failed	9	0.0000675
		battery_1.Failed	9	0.0000675
battery_2	Батарея разряжена	b.[propagation]	9	0.0000675
		sw_1_b.Failed	9	0.0000675
		sw_2_b.Failed	9	0.0000675
		battery_2.Failed	9	0.0000675

Теперь единичный сбой в батарее не приводит к опасному состоянию уровня всей системы.

Таким образом, используя анализ рисков, удалось выявить, локализовать и устранить ошибку проектирования, причем сделать это на этапе моделирования системы.

Еще один недостаток проектирования рассмотренной здесь системы выражается в том, что вероятность отказа в каком-либо процессе (функциональная опасность «Отказ») имеет порядок 10^{-4}, что является слишком большим значением. Этот недостаток можно устранить, например, если дублировать процессы, причем выполнять их на разных независимых процессорах. Однако, подробное рассмотрение этого сценария выходит за рамки настоящей статьи.

Следует также отметить, что поскольку для рассмотренной здесь системы не предусмотрено восстановление от ошибок, то использование марковского анализа в данном случае не дает существенной дополнительной информации по сравнению с результатами анализа дерева неисправностей.

6. Заключение

AADL и Error Model Annex позволяют систематически разрабатывать формальные модели состем и формально описывать требования по безопасности. AADL-модели со спецификациями на Error Model Annex позволяют автоматизировать многие разновидности анализа рисков, которые в частности требуются для сертификации самолетов гражданской авиации [32].

В настоящей статье мы представили алгоритмы для следующих видов анализа рисков на основе AADL и Error Model Annex: анализ дерева неисправностей, анализ видов и последствий отказов, марковский анализ.

Представленные в статье алгоритмы реализованы в инструменте MASIW. Их использование позволяет выявлять и локализовывать ошибки проектирования системы на этапе моделирования системы.

Список литературы

[1] Д.В. Буздалов, С.В. Зеленов, Е.В. Корныхин, А.К. Петренко, А.В. Страх, А.А. Угненко, А.В. Хорошилов. Инструментальные средства проектирования систем интегрированной модульной авионики. Труды ИСП РАН, том 26, вып. 1, 2014, стр. 201-230. DOI: 10.15514/ISPRAS-2014-26(1)-6

[2] Б.В. Гнеденко, Ю.К. Беляев, И.Н. Коваленко. Математические вопросы теории надежности. Итоги науки. Сер. Теор. вероятн. Мат. стат. Теор. Кибернет. 1964, 1966. 7-53.

[3] Гнеденко Б.В., Беляев Ю.К., Соловьев А.Д. Математические методы в теории надежности. М.: Наука, 1965.

[4] В.К. Дедков, А.С. Проников, А.Н. Терпиловский. Надежность сложных технических систем. Методы определения и обеспечения надежности промышленной продукции. Под ред. Г. Н. Бобровникова. М.:АНХ 1983.

[5] Карнов А.А., Зеленов С.В. Стохастические методы анализа комплексных программно-аппаратных систем. Труды ИСП РАН, том 29, вып. 4, 2017 г., стр. 191-202. DOI: 10.15514/ISPRAS-2016-29(4)-12
[6] Никольский В.И. Некоторые аварии и катастрофы отечественных и пассажирских судов. СПб.: СПГУВК, 2011.
[7] Рябинин И.А. Концепция логико-вероятностной теории безопасности.// М., «Приборы и системы управления», №10, 1993, с.6-9.
[8] Рябинин И.А. Надежность и безопасность структурно-сложных систем. Спб., Политехника, 2000.
[9] Рябинин И.А. Логико-вероятностный анализ проблем надежности живучести и безопасности. Новочеркасск, Южно-Российский государственный университет. Новочеркасск: Лик, 2009. -600с.
[10] Рябинин И.А., Черкесов Г.Н. Логико-вероятностные методы исследования надежности структурно-сложных систем.// Изд. "Радио и связь", М., 1981.
[11] Альберт Ширяев. Вероятность. 4-е издание, переработанное и дополненное. М.:МЦНМО. 2007.
[12] ГОСТ Р 27.302-2009. Надежность в технике. Анализ дерева неисправностей.
[13] K.K. Aggarwal, J.S. Gupta, and K.B. Misra. A new method for system reliability evaluation. Microelectronics Reliability, 12(5):435–440, Nov 1973.
[14] U.M. Ascher and L.R. Petzold. Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1998.
[15] E.E. Barlow, F. Proschan, and L.C. Hunter. Mathematical Theory of Reliability. Wiley, New York-London-Sydney, 1965.
[16] R.G. Bennetts. On the analysis of fault trees. IEEE Transactions on Reliability, R-24(3):175–185, Aug 1975.
[17] J. Delange, P. Feiler, D. Gluch, J. Hudak. AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. CMU/SEI-2014-TR-020, 2014.
[18] Peter H. Feiler, David P. Gluch. Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley Professional, 2012.
[19] L. Fratta and U.G. Montanari. A boolean algebra method for computing the terminal reliability in a communication network. IEEE Transactions on Circuit Theory, 20(3):203–211, 1973.
[20] J. Hadamard. Lectures on Cauchy's Problem in Linear Partial Differential Equations. Dover phoenix editions. Dover Publications, 2003.
[21] E.J. Henley and H. Kumamoto. Reliability engineering and risk assessment. Prentice-Hall, 1981.
[22] E.J. Henley and H. Kumamoto. Designing for reliability and safety control. Prentice-Hall International Series in Industrial and Systems Engineering. Prentice-Hall, 1985.
[23] Alexey Khoroshilov, Dmitry Albitskiy, Igor Koverninskiy, Mikhail Olshanskiy, Alexander Petrenko, and Alexander Ugnenko. AADL-based toolset for IMA system design and integration. SAE Int. J. Aerosp., 5:294–299, Oct 2012.
[24] M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In Proc. 23rd International Conference on Computer Aided Verification (CAV11), ser. LNCS, volume 6806, pages 585–591. Springer, 2011.
[25] Nils J. Nilsson. Probabilistic logic. Artif. Intell., 28(1):71–88, February 1986.

[26] I.A. Ryabinin. Reliability of Engineering Systems. Principles and Analysis. MIR, Moscow, 1976.
[27] W. Vesely, J. Dugan, J. Fragola, Minarick, and J. Railsback. Fault Tree Handbook with Aerospace Applications. Handbook, National Aeronautics and Space Administration, Washington, DC, 2002.
[28] ARINC 664 part 7, Avionics Full Duplex Switched Ethernet (AFDX) network, 2005.
[29] MASIW: Modular Avionics System Integrator Workplace, 2016. https://forge.ispras.ru/projects/masiw-oss/.
[30] OpenFTA, 2005. http://openfta.com/.
[31] OSATE: Open Source AADL2 Tool Environment, 2016. http://osate.org/.
[32] SAE International standard ARP4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, 1996. http://standards.sae.org/arp4761/.
[33] SAE International standard AS5506C, Architecture Analysis & Design Language (AADL), 2004. Rev. 2017, http://standards.sae.org/as5506c/.
[34] SAE International standard AS5506/1A, Architecture Analysis & Design Language (AADL), Annex E: Error Model Annex, 2011. Rev. 2015, http://standards.sae.org/as5506/1a/.

Modeling and Risk Analysis of Hardware-Software Systems

[1] *S.A. Zelenova <sophia@ispras.ru>*
[1,2] *S.V. Zelenov <zelenov@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *National Research University Higher School of Economics (HSE) 20 Myasnitskaya Ulitsa, Moscow, 101000, Russia.*

Abstract. Hardware-software systems are widely used now and must be safe and reliable. Manual analysis of risks for structural complex systems is very expensive, so formal automated methods are required. The most important aspect here is the possibility to describe safety requirements in terms used in safety theory, such as Markov chains or logic-probabilistic functions, since for the decades of development of the theory, a large number of very useful results have been accumulated. Different approaches to assessing safety of systems do not compete, but complement each other, so having some universality in describing safety requirements is a very valuable quality.

In this article, we demonstrate the advisability of using the AADL modeling language and its extension Error Model Annex to describe safety requirements of a system under design.

First, we describe a mathematical model of safety requirements expressible in AADL Error Model Annex.

Next, we present algorithms to perform the following automated risk analysis on the base of AADL models: Fault Tree Analysis (including calculation of minimal cut sets and ranking of primary events with respect to different relevant importance measures), Failure Mode and Effects Analysis, and Markovian Analysis.

At last, we consider an example of a real avionic system. We present an architecture of an AADL model of the system under design and describe how to develop Error Model Annex specifications for the model. With the help of risk analysis, we show how one can identify, localize and fix a bug in the architecture of the system on the design stage of the system development.

All presented algorithms are implemented in MASIW framework for design of modern avionics systems.

Keywords: risk analysis; reliability; safety; fault tree analysis; failure mode and effects analysis; markovian analysis.

DOI: 10.15514/ISPRAS-2017-29(5)-13

For citation: Zelenova S.A., Zelenov S.V. Modeling and Risk Analysis of Hardware-Software Systems. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 257-282 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-13

References

[1] D. V. Buzdalov, S. V. Zelenov, E. V. Kornykhin, A. K. Petrenko, A. V. Strakh, A. A. Ugnenko, and A. V. Khoroshilov. Tools for system design of integrated modular avionics. Trudy ISP RAN/Proc. ISP RAS, volume 26, issue 1, pages 201–230, 2014. DOI: 10.15514/ISPRAS-2014-26(1)-6 (Russian)

[2] Gnedenko, B. V.; Beljaev, Ju. K.; Kovalenko, I. N. Mathematical problems in the theory of reliability. (Russian) 1966 Theory of Probability, Math. Statist., Theoret. Cybernet. 1964 (Russian) pp. 7–53 Akad. Nauk SSSR Inst. Naučn. Informacii, Moscow.

[3] B.V. Gnedenko, Y.K. Belyayev, and A.D. Solovyev. Mathematical methods of reliability theory. Nauka, Moscow, 1965. (Russian)

[4] V.K. Dedkov, A.S. Pronikov, A.N. Terpilovskij. Reliability of complex technical systems. Methods for determining and ensuring the reliability of industrial products. Academy of National Economy, Moscow, 1983. (Russian)

[5] Karnov A.A., Zelenov S.V. Stochastic Methods for Analysis of Complex Hardware-Software Systems. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 4, 2017, pp. 191-202. DOI: 10.15514/ISPRAS-2016-29(4)-12

[6] Nikolskij V.I. Some accidents and disasters of domestic passenger ships. St. Petersburg State University of Water Communications, St.Petersburg, 2011. (Russian)

[7] I.A. Ryabinin. The concept of the logic-probabilistic theory of safety. Devices and control system, 10:6–9, 1993. (Russian)

[8] I.A. Ryabinin. Reliability and Safety of Structural Complex Systems. Politechnika, St.Petersburg, 2000. (Russian)

[9] I.A. Ryabinin. Logic-probabilistic Analysis of Problems of Safety, Survivability and Safety. South Russian State University, Lik, Novocherkassk, 2009. (Russian)

[10] I.A. Ryabinin and G.N. Cherkesov. The logic-probabilistic research methods of structure-complex systems reliability. Radio and communication, Moscow, 1981. (Russian)

[11] Albert Nikolaevich Shiryaev. Probability. 2nd edition, 1995.

[12] State Standard 27.302-2009. Dependability in technics. Fault tree analysis. Moscow, Standartinform Publ., 2011. (In Russian)

[13] K.K. Aggarwal, J.S. Gupta, and K.B. Misra. A new method for system reliability evaluation. Microelectronics Reliability, 12(5):435–440, Nov 1973.

[14] U.M. Ascher and L.R. Petzold. Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1998.
[15] E.E. Barlow, F. Proschan, and L.C. Hunter. Mathematical Theory of Reliability. Wiley, New York-London-Sydney, 1965.
[16] R.G. Bennetts. On the analysis of fault trees. IEEE Transactions on Reliability, R-24(3):175–185, Aug 1975.
[17] J. Delange, P. Feiler, D. Gluch, J. Hudak. AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. CMU/SEI-2014-TR-020, 2014.
[18] Peter H. Feiler, David P. Gluch. Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley Professional, 2012.
[19] L. Fratta and U.G. Montanari. A boolean algebra method for computing the terminal reliability in a communication network. IEEE Transactions on Circuit Theory, 20(3):203–211, 1973.
[20] J. Hadamard. Lectures on Cauchy's Problem in Linear Partial Differential Equations. Dover phoenix editions. Dover Publications, 2003.
[21] E.J. Henley and H. Kumamoto. Reliability engineering and risk assessment. Prentice-Hall, 1981.
[22] E.J. Henley and H. Kumamoto. Designing for reliability and safety control. Prentice-Hall International Series in Industrial and Systems Engineering. Prentice-Hall, 1985.
[23] Alexey Khoroshilov, Dmitry Albitskiy, Igor Koverninskiy, Mikhail Olshanskiy, Alexander Petrenko, and Alexander Ugnenko. AADL-based toolset for IMA system design and integration. SAE Int. J. Aerosp., 5:294–299, Oct 2012.
[24] M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In Proc. 23rd International Conference on Computer Aided Verification (CAV11), ser. LNCS, volume 6806, pages 585–591. Springer, 2011.
[25] Nils J. Nilsson. Probabilistic logic. Artif. Intell., 28(1):71–88, February 1986.
[26] I.A. Ryabinin. Reliability of Engineering Systems. Principles and Analysis. MIR, Moscow, 1976.
[27] W. Vesely, J. Dugan, J. Fragola, Minarick, and J. Railsback. Fault Tree Handbook with Aerospace Applications. Handbook, National Aeronautics and Space Administration, Washington, DC, 2002.
[28] ARINC 664 part 7, Avionics Full Duplex Switched Ethernet (AFDX) network, 2005.
[29] MASIW: Modular Avionics System Integrator Workplace, 2016. https://forge.ispras.ru/projects/masiw-oss/.
[30] OpenFTA, 2005. http://openfta.com/.
[31] OSATE: Open Source AADL2 Tool Environment, 2016. http://osate.org/.
[32] SAE International standard ARP4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, 1996. http://standards.sae.org/arp4761/.
[33] SAE International standard AS5506C, Architecture Analysis & Design Language (AADL), 2004. Rev. 2017, http://standards.sae.org/as5506c/.
[34] SAE International standard AS5506/1A, Architecture Analysis & Design Language (AADL), Annex E: Error Model Annex, 2011. Rev. 2015, http://standards.sae.org/as5506/1a/.

Распределённые алгоритмы на корневых неориентированных графах

Игорь Бурдонов <igor@ispras.ru>
Александр Косачев <kos@ispras.ru>
Александр Сортов <sortov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Рассматриваются распределённые алгоритмы решения задач на неориентированных графах. В разделе 2 определяется используемая модель, особенностью которой является наличие корня, с которого начинается и в котором заканчивается работа алгоритма. Описываются синхронная и асинхронная разновидности модели. В разделе 3 предлагаются алгоритмы решения любых задач, основанные на сборе информации о всём графе в корне или в каждой вершине, а также, если необходимо, разметке графа (его вершин и/или рёбер). Акцент сделан на времени работы алгоритма, а при минимальном времени – на экономии памяти в вершинах и суммарном объёме пересылаемых сообщений. В остальной части статьи рассматриваются оптимизации для конкретных задач: построение максимального независимого множества (MIS – Maximal Independent Set), поиск множества всех мостов в графе (FSB – Finding Set of Bridges), построение минимального остовного дерева во взвешенном графе (MST – Minimum Spanning Tree). В разделе 4 предлагается модификация общих алгоритмов для этих задач, уменьшающая оценки размера памяти вершин и сообщений. Раздел 5 содержит нижние оценки сложности решения этих задач. В разделе 6 для синхронной модели уменьшается время работы алгоритмов с разметкой графа до нижней границы для задач с однозначным решением, зависящим только от простых циклов графа, в частности, FSB, MST и задачи поиска гальмильтонова цикла. В разделе 7 рассматриваются оптимальные по времени алгоритмы для FSB и MST для обеих моделей: синхронной и асинхронной. Заключение подводит итоги и намечает направления дальнейших исследований.

Ключевые слова: корневой неориентированный граф; распределённые алгоритмы; задачи на графах; максимальное независимое множество; минимальное остовное дерево; поиск мостов.

DOI: 10.15514/ISPRAS-2017-29(5)-14

Для цитирования: Бурдонов И., Косачев А., Сортов А. Распределённые алгоритмы на корневых неориентированных графах. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 283-310. DOI: 10.15514/ISPRAS-2017-29(5)-14

1. Введение

Исследование алгоритмов решения задач на графах имеет давнюю историю. Первоначально такие алгоритмы были чисто последовательными, но со второй половины XX века всё больше внимания стало уделяться методам распараллеливания, позволяющим уменьшить время решения. Общий принцип параллельных графовых алгоритмов заключается в том, что граф подразделяется на меньшие локальные области, в которых задачи решаются параллельно, выполняя идентичный алгоритм. Вначале использовались модели типа *PRAM* (*Parallel Random-Access Machine*) [1], в которой процессоры имеют разделяемую общую память, через которую взаимодействуют друг с другом. Это модель, скорее, параллельной, чем распределённой обработки. Впоследствии исследования распространились на случай распределенных алгоритмов на компьютерных кластерах, когда общей памяти нет, но процессоры взаимодействуют друг с другом с помощью обмена сообщениями, используя заданную сеть связи. В конечном итоге такую сеть стали считать совпадающей с графом, на котором решается графовая задача. В вершинах графа находятся вычислительные единицы, которые назывались по-разному: процессами, процессорами, автоматами. Они не имеют общей памяти, но могут обмениваться между собой сообщениями, передаваемыми по рёбрам графа. В ориентированном графе сообщения передаются по рёбрам в направлении их ориентации, а в неориентированном графе – в обоих направлениях. Можно сказать, что такая графовая сеть исследует саму себя, решая те или иные задачи на этом графе, а «локальные области», на которые разбивается граф, сужаются до одиночных вершин, автоматы которых работают параллельно, выполняя идентичный алгоритм.

В качестве характерного примера проследим краткую историю решения задачи о максимальном независимом множестве. Подмножество вершин неориентированного графа называется *независимым*, если никакие две вершины из подмножества не соединены ребром. *Максимальным независимым множеством*, сокращённо *MIS* (*Maximal Independent Set*), называется независимое множество, являющееся максимальным элементом в семействе всех независимых множеств по отношению вложенности. Отметим, что в русскоязычной литературе часто используется не вполне удачная терминология, когда такое множество называется наибольшим, а максимальным называется наибольшее множество, в котором наибольшее число вершин (соответствует англ. термину *MaxIS* – *Maximum Independent Set*).

Проблема построения MIS – одна из основных проблем в теории графов и теории распределённых алгоритмов на графах. Многие другие проблемы могут быть сведены к проблеме MIS, например, раскраска графа и поиск максимального паросочетания. Другие проблемы, хотя и не сводятся к проблеме MIS, но также тесно связаны с ней, поскольку алгоритмы их решения используют алгоритм построения MIS как подпрограмму. Примером могут служить задача о вершинном покрытии или поиск максимальной клики.

Следует отметить, что проблемы построения MaxIS и перечисления всех MIS NP-трудные. В противоположность этому проблема MIS легко решается тривиальным последовательным алгоритмом: сначала линейно упорядочиваем множество вершин графа, затем строим множество, добавляя к нему вершину, если она не смежна ни с какой вершиной множества. Результатом будет MIS, которое называется *лексикографическим MIS* (*LFMIS*), поскольку при построении используется заданный порядок вершин.

Задача поиска LFMIS P-полна, поэтому сначала думали, что эта задача трудна для распараллеливания. Однако было показано, что детерминированное параллельное решение можно получить с помощью NC^1 редукции из решения проблемы максимальной упаковки множеств или проблемы максимального паросочетания, либо же редукцией из решения проблемы 2-*SAT* (задача выполнимости булевых формул в 2-конъюнктивной нормальной форме) [2][3]. Напомним, что через NC^i обозначается класс сложности проблем, разрешимых за время $\boldsymbol{O}(log^i n)$ на параллельном компьютере с полиномиальным числом процессоров, где n – число вершин графа. В 1984 Karp и др. показали, что для модели PRAM детерминированное параллельное решение проблемы MIS принадлежит классу сложности NC^4 [4]. Их алгоритм находит MIS за время $\boldsymbol{O}(log^4 n)$, используя $\boldsymbol{O}((n/logn)^3)$ процессоров, где n – число вершин графа. В той же статье предложено рандомизированное параллельное решение с временем выполнения $\boldsymbol{O}(log^4 n)$ и числом процессоров $\boldsymbol{O}(n^2)$. Через 2 года после этого Luby и Alon и др. независимо улучшили этот результат, перенеся проблему MIS в область NC^2 с временем выполнения $\boldsymbol{O}(log^2 n)$ и число процессоров $\boldsymbol{O}(mn^2)$, где m – количество рёбер в графе [5][6]. Они предложили рандомизированный алгоритм, который использует $\boldsymbol{O}(m)$ процессоров, но может быть дерандомизирован с дополнительными $\boldsymbol{O}(n^2)$ процессорами для каждого из первоначальных m процессоров.

Работа Luby и Alon и др. инициировала исследования по распределённым алгоритмам [7][8][9]. В [7] Peleg предложил модель распределённых алгоритмов LOCAL, в которой нет общей памяти процессоров и которая является графово-ориентированной: процессоры находятся в вершинах графа взаимодействуют друг с другом только обменом сообщениями, передаваемым по рёбрам графа. Тем самым проблема MIS решается для графа, который является графом связи процессоров в модели LOCAL. Предлагавшиеся алгоритмы имели размер сообщения с нижней границей $\boldsymbol{O}(logn)$ битов, и требовали знания дополнительных характеристик графа. Например, должен быть известен размер графа, или для данной вершины можно было узнать максимальную степень соседних вершин. В 2010 Métivier и др. сумели уменьшить размер сообщения до $\boldsymbol{O}(1)$, который минимален, и устранили необходимость любого дополнительного знания о графе [10].

Одним из последних достижений в этой области стала работа Ghaffari, доложенная на симпозиуме SODA и опубликованная в 2016 г. [11], но выложенная в интернет на сайте Корнеллского университета в 2015 г. [12]. В

рамках модели LOCAL его рандомизированный алгоритм с вероятностью не менее 1-1/n строит MIS за время $\boldsymbol{O}(log\Delta)+2^{\boldsymbol{O}(\sqrt{(loglogn)})}$, где Δ – максимальная степень вершины. Это лучше, чем предыдущий результат $\boldsymbol{O}(log^2\Delta)+2^{\boldsymbol{O}(\sqrt{(loglogn)})}$ в [13], и уже близок к нижней оценке $\boldsymbol{\Omega}(min\{log\Delta, \sqrt{(logn)}\})$, доказанной в [14].

LOCAL является синхронной моделью: автоматы во всех вершинах срабатывают одновременно, принимая все посланные им сообщения и посылая новые сообщения; это называется *раундом*. При этом время перемещения сообщения по ребру фиксировано (1 такт), а временем срабатывания автомата, т.е. временем вычислений в вершинах пренебрегают. Время работы алгоритма считается в числе раундов (или тактов). В асинхронных моделях сообщение может двигаться по ребру произвольное ограниченное сверху (1 тактом) время, поэтому время работы алгоритма считается в тактах и в «наихудшем» случае. Для некоторых алгоритмов это существенно: существуют синхронные алгоритмы, которые в асинхронной модели либо не работают, либо имеют другие временные оценки. В данной статье мы рассматриваем оба типа модели, хотя некоторые предлагаемые алгоритмы применимы только в синхронном варианте.

Но более важной особенностью как системы LOCAL, так и некоторых асинхронных моделей является предположение об одновременном начале работы автоматов в вершинах. В противоположность этому существуют модели, в которых работа начинается с одной вершины, называемой *корнем* графа. Остальные вершины «спят», а «просыпаются» только при получении сообщений, посылаемых в конечном счёте от корня. В данной статье мы рассматриваем как раз такие *корневые* модели. В этих моделях существенно то, что время работы алгоритма учитывает время начального распространения сообщений от корня до остальных вершин (*broadcast*). Это время в синхронной модели равно, а в асинхронной не превышает эксцентриситета корня d_0 (максимального расстояния от корня до других вершин). Более того, мы будем считать, что работа алгоритма не только начинается, но и заканчивается в корне. Это значит, что, используя завершающее распространение сообщений от всех вершин к корню (*convergecast*), корень должен «узнать» о завершении решения задачи в каждой вершине, что требует такого же времени d_0. Тем самым, время работы алгоритма для задач, решение которых зависит от всего графа (в частности, проблемы MIS), имеет нижнюю границу $2d_0 = \boldsymbol{O}(n)$. Это делает неактуальным для корневых моделей многие алгоритмы, разработанные для модели LOCAL.

Данная статья – вторая в серии статей, начатой нами в [15]. Там был предложен общий подход к распределенному решению задач на графах, основанный на систематизации способов распространения сообщений по графу и построения типовых конструкций, в частности, остова графа.

В разделе 2 описывается общая корневая модель распределённых алгоритмов для неориентированных графов в синхронном и асинхронном вариантах, отмечаются её существенные отличия от других моделей, прежде всего,

модели LOCAL. В разделе 3 рассматриваются алгоритмы решения любых задач на неориентированных нумерованных графах, основанные на сборе информации о всём графе в корне (*broadcast+convergecast*) или в каждой вершине (*broadcast+all-to-all-broadcast+convergecast*). Акцент сделан на времени работы алгоритма, а при минимальном времени – на экономии памяти автоматов в вершинах и суммарном объёме пересылаемых сообщений как по одному ребру, так и по всем рёбрам, как одновременно, так и за всё время работы алгоритма. Эти алгоритмы различаются, прежде всего, типом модели: синхронная или асинхронная. Кроме того, задача решается без разметки графа или с разметкой графа. В первом случае корень формирует ответ в виде сообщения вовне графа. Например, для MIS – как множество номеров вершин из MIS. Во втором случае решение задачи – это разметка графа, при которой помечаются некоторые вершины и/или рёбрам графа, а корень сообщает вовне только о том, что задача решена и разметка выполнена. Например, для MIS помечаются вершины, входящие в MIS.

Общие алгоритмы, пригодные для решения любых задач на графах, конечно, могут оказаться не оптимальными по времени и/или памяти при решении той или иной конкретной задачи. В остальной части статьи рассматриваются различные оптимизации на примере трёх важных графовых задач. Одна из них – это проблема MIS. Вторая – задача поиска множества всех мостов в графе – FSB (*Finding Set of Bridges*). Напомним, что мост – это ребро графа, удаление которого увеличивает число компонентов связности графа. Третья – задача построения минимального остовного дерева во взвешенном графе – MST (*Minimum Spanning Tree*). Напомним, что MST – это остовное дерево графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер.

В разделе 4 предлагается модификация общих алгоритмов для этих трёх задач, уменьшающая оценки размера памяти вершин и сообщений. Раздел 5 содержит нижние оценки сложности алгоритмического решения задач MIS, FSB и MST. В разделе 6 показано, что в синхронной модели можно уменьшить время работы алгоритмов с разметкой графа до нижней границы для задач с однозначным решением, зависящим только от простых циклов графа. К таким задачам относятся, в частности, задачи FSB, MST и задача поиска гальмильтонова цикла. В разделе 7 в обеих моделях (синхронной и асинхронной) рассмотрены алгоритмы для задач FSB и MST, в которых удаётся совместить начальное (*broadcast*) и завершающее (*convergecast*) распространение сообщений с решением задачи и разметкой графа за минимально возможное время. Заключение подводит итоги и намечает направления дальнейших исследований.

2. Две модели распределённых алгоритмов

Используемая нами модель похожа на модель LOCAL: автоматы находятся в вершинах графа и обмениваются между собой сообщениями, которые

пересылаются по рёбрам графа. Для краткости вместо «автомат в вершине» мы будем часто писать просто «вершина». Граф предполагается неориентированным, связным, без кратных рёбер и петель, с выделенной вершиной – корнем графа.

Ребро понимается как дуплексный канал передачи сообщений. Для того, чтобы вершина могла указать ребро, по которому она посылает сообщение, все инцидентные вершине рёбра считаются пронумерованными от 1 до степени вершины. Такой граф называется *упорядоченным* [15][16]. При посылке сообщения по ребру вершина указывает номер ребра. Получая сообщение, вершина «узнаёт» номер ребра, по которому сообщение получено.

Основное отличие от системы LOCAL в том, что алгоритм начинает работать не одновременно во всех вершинах графа, а только с корня. Другие вершины подключаются к решению задачи после того, как получат сообщения, передаваемые по путям от корня до этих вершин. Соответственно, задача считается решённой только после того, как об этом «узнал» корень. Будем считать, что корень связан с «окружением» фиктивным ребром для получения извне сообщения, инициирующего решение задачи, и для посылки вовне сообщения о завершении решения задачи. Такую модель будем называть *корневой* (rooted).

Мы будем рассматривать две разновидности корневой модели: синхронную – $\mathcal{RSM}$ (Rooted Synchronous Model) и асинхронную – $\mathcal{RAM}$ (Rooted Asynchronous Model). В синхронной модели, так же как в системе LOCAL, сообщение перемещается по ребру фиксированное время – 1 такт. В асинхронной модели сообщение передаётся по ребру, вообще говоря, произвольное время, ограниченное сверху 1 тактом. В $\mathcal{RAM}$ время передачи сообщения по ребру может быть различным для разных рёбер, для разных направлений на одном ребре, для разных сообщений на одном ребре в одном направлении и, вообще говоря, меняться со временем.

В обеих моделях за одно срабатывание автомата в вершине (*раунд*) принимаются все дошедшие до вершины сообщения и посылается одно или несколько сообщений по одному или нескольким рёбрам. Предполагается, что сообщения не теряются, не генерируются рёбрами, и сообщения, посланные по одному ребру и ещё не принятые другим его концом, не обгоняют друг друга, т.е. выстраиваются в очередь на ребре. Как и в системе LOCAL мы пренебрегаем временем срабатывания автоматов в вершинах графа. Поэтому посылка по одному ребру нескольких сообщений в одном раунде эквивалентна посылке одного «склеенного» сообщения.

В $\mathcal{RSM}$ как и в LOCAL между раундами проходит ровно один такт, и время работы алгоритма можно считать как число раундов (*time complexity* в [9]). В $\mathcal{RAM}$ время между раундами может быть произвольным в интервале (0,1], поэтому нет смысла считать время работа алгоритма в раундах, оно оценивается в тактах в «наихудшем случае» в зависимости от времени передачи по рёбрам тех или иных сообщений в те или иные моменты времени.

Будем считать, что сообщение представляет собой набор параметров, а память вершины (автомата в вершине) – набор переменных, сохраняемых между раундами, т.е. без учёта памяти для приёма сообщений и формирования сообщений перед посылкой (эта память ограничена размером сообщения). Имена параметров будем писать *курсивом строчными буквами*, а имена переменных – ***жирным курсивом строчными буквами***.

Мы будем давать следующие оценки алгоритмов: T – время работы алгоритма, A – размер памяти вершины как сумма размеров переменных без учёта памяти для принимаемых и посылаемых сообщений, M_1 – максимальный суммарный размер сообщений, находящихся на одном ребре одновременно, M_{all} – максимальный суммарный размер сообщений, находящихся на всех рёбрах одновременно, S_1 – максимальный суммарный размер сообщений, проходивших через одно ребро за всё время работы алгоритма, S_{all} – максимальный суммарный размер сообщений, проходивших через все рёбра за всё время работы алгоритма (если сообщение проходит k рёбер, оно считается k раз).

Эти оценки будут даваться как функции от следующих параметров графа: n – число вершин, Δ – максимальная степень вершины, D – длина максимального пути, D_0 – длина максимального пути от корня, d – диаметр графа, т.е. максимальное расстояние между вершинами, где расстояние между вершинами – это длина кратчайшего пути между вершинами, d_0 – эксцентриситет корня, т.е. максимальное расстояние от корня до вершины. Очевидно, $\Delta \le n\text{-}1$, $d_0 \le d \le 2d_0$, $D_0 \le D \le 2D_0$, $d_0 \le D_0$, $d \le D \le n\text{-}1$ [15] и, если $d_0 > 1$ и $\Delta > 2$, то $d_0+\Delta\text{-}1 \le n \le ((\Delta\text{-}1)^{d_0+1}\text{-}1)/(\Delta\text{-}2)$.

В предлагаемых ниже алгоритмах граф, как правило, будет предполагаться *нумерованным* [15][16]: его вершинам присвоены номера от 1 до n. Номер вершины хранится в её переменной с самого начала.

3. Общий случай

В этом разделе мы рассмотрим общие алгоритмы для решения любой задачи на графе в моделях $\mathcal{RSM}$ и $\mathcal{RAM}$. Мы будем различать решения без разметки графа и решения с разметкой графа. Если задача решается без разметки графа, то решение содержится в завершающем сообщении, которое корень посылает вовне в конце работы алгоритма. Если задача решается с разметкой графа, то решение – это глобальная разметка графа: вершинам и рёбрам присваиваются некоторые метки, а завершающее сообщение, посылаемое корнем вовне, только сообщает о том, что разметка выполнена. При этом каждая вершина выполняет локальную разметку: инициализирует метку вершины, задаваемую специальной переменной в вершине, и метки инцидентных вершине рёбер, задаваемые в другой переменной, содержащей отображение номера ребра в этой вершине на значение метки ребра. Поскольку ребро инцидентно двум

вершинам (концам ребра), оно получает две метки, по одной в каждом из своих концов (обычно эти метки одинаковые).

Мы предлагаем алгоритмы с возможно меньшим временем работы T, а при данном T – с возможно меньшими размерами памяти вершины и сообщений.

3.1 Три типа алгоритмов и пять типов сообщений

Мы предложим три типа общих алгоритмов:

1. *Сбор в корне*. Из корня рассылается во все вершины сообщение, инициирующее сбор информации о графе. В корне графа собирается информация о всём графе, после этого в корне решается задача и посылается вовне сообщение с решением. Граф не размечается. Например, для MIS завершающее сообщение содержит множество номеров вершин из MIS, а для MST или FSB – множество рёбер из MST или множество мостов, соответственно.

2. *Сбор в корне и разметка из корня*. Выполняется сбор информации о графе в корне, решается задача, а затем выполняется глобальная разметка графа в соответствии с найденным решением. Для разметки из корня рассылается во все вершины информация о разметке, по которой каждая вершина выполняет свою локальную разметку, после чего извещает корень о завершении локальной разметки в этой вершине. Глобальная разметка завершена, когда корень получит извещения от всех вершин. Корень посылает вовне завершающее сообщение о том, что задача решена и граф размечен. Например, для MIS помечаются вершины из MIS, а для MST или FSB – рёбра из MST или мосты, соответственно.

3. *Сбор и разметка во всех вершинах*. Из корня рассылается во все вершины сообщение, инициирующее сбор информации о графе. Информация о графе собирается не только в корне, а в каждой вершине графа. Собрав всю информацию о графе, каждая вершина самостоятельно решает задачу и выполняет свою локальную разметку, после чего извещает корень о завершении локальной разметки в этой вершине. Важное требование: алгоритм должен гарантировать, что все вершины находят одно и то же решение задачи. Получив извещения от всех вершин, корень посылает вовне завершающее сообщение о том, что задача решена и граф размечен.

Информация о графе, собираемая в корне (алгоритмы типа **1** и **2**) или в каждой вершине (алгоритмы типа **3**), в общем случае описывает все вершины и рёбра графа. Некоторые задачи требуют взвешенных графов, в которых вершинам и/или рёбрам приписаны веса. Для определённости будем считать, что вес – целое число от 1 до максимального значения w. В дальнейшем оценки алгоритмов приводятся для случая невзвешенных графов, для взвешенных графов размер описания вершин и/или рёбер увеличивается не более чем в $\log w$ раз. Также в дальнейшем считается, что метка вершины и/или ребра принимает два значения: «помечено», «не помечено». В более общем случае

размер информации о разметке увеличивается не более чем в *logu* раз, где *u* – число различных значений метки.

Алгоритмы этих трёх типов используют сообщения пяти типов:

1. ***Старт*** – сообщение от корня в каждую вершину (*Broadcast*), инициирующее сбор информации о графе.
2. ***ИнфоКорню*** – сообщение от каждой вершины в корень (*Convergecast*), содержащее информацию о соседях вершины.
3. ***Разметка*** – сообщение от корня в каждую вершину (*Broadcast*), содержащее информацию о разметке: какие рёбра и вершины нужно пометить.
4. ***ИнфоВсем*** – сообщение от каждой вершины в каждую вершину (*All-to-All Broadcast*), содержащее информацию о соседях вершины.
5. ***Финиш*** – сообщение от каждой вершины в корень (*Convergecast*) об окончании локальной разметки в этой вершине.

Сообщения, используемые разными типами алгоритмов:

1. *Сбор в корне*: ***Старт*** + ***ИнфоКорню***.

2. *Сбор в корне и разметка из корня*: ***Старт*** + ***ИнфоКорню*** + ***Разметка*** + ***Финиш***.

3. *Сбор и разметка во всех вершинах*: ***Старт*** + ***ИнфоВсем*** + ***Финиш***.

3.2 Пять классов сообщений по способу их передачи

В [15] определены 9 способов передачи сообщений. В данной статье нам достаточно 5 из них, мы определим их в этом подразделе. Оценки этих способов приведены в 0, где *m* означает максимальный размер сообщения. Как функции от *m* и *n* эти оценки достигаются на графе на Рис. 1.

Табл. 1. Оценки способов передачи сообщений.
Table 1. Estimations of the methods of transmitting messages.

<table>
<tr><th></th><th colspan="2">P</th><th colspan="2">O</th><th>C</th><th>П</th><th colspan="2">$M_к$</th><th colspan="2">$M_в$</th></tr>
<tr><td>MODEL</td><td>RSM</td><td>RAM</td><td colspan="2">RSM</td><td>RSM</td><td>RSM</td><td>RSM</td><td>RAM</td><td>RSM</td><td>RAM</td></tr>
<tr><td>A</td><td colspan="3">$O(log\Delta) = O(logn)$</td><td></td><td>$O(\Delta) = O(n)$</td><td colspan="4">$O(n)$</td></tr>
<tr><td>M_1 and S_1</td><td colspan="2">$O(m)$</td><td colspan="2" rowspan="2">$O(nm)$</td><td>$O(m)$</td><td colspan="4">$O(nm)$</td></tr>
<tr><td>M_{all}</td><td colspan="2" rowspan="2">$O(n\Delta m)$ = $O(n^2m)$</td><td rowspan="2">$O(nm)$</td><td colspan="4" rowspan="2">$O(n^2\Delta m) = O(n^3m)$</td></tr>
<tr><td>S_{all}</td><td colspan="2">$O(nd_0m) = O(n^2m)$</td></tr>
<tr><td>T</td><td>$= d_0$</td><td>$\le d_0$</td><td colspan="3">$= d_0$</td><td>$\le d_0$</td><td>$= d \le 2d_0$</td><td>$\le d \le 2d_0$</td></tr>
<tr><td>T*</td><td colspan="2">$\le d_0+1$</td><td colspan="7"></td></tr>
</table>

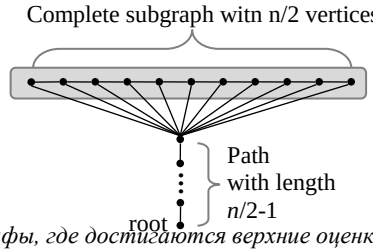


Рис. 1. Графы, где достигаются верхние оценки M_1, S_1, M_{all}, S_{all}.

Fig. 1. Graphs where the upper bounds M_1, S_1, M_{all}, S_{all} are reached.

1. Класс ***P*** – рассылка без повторения из корня. Вначале сообщение создаётся корнем и посылается по всем инцидентным корню рёбрам. Когда некорневая вершина получает сообщение первый раз по некоторому ребру, она пересылает его по всем инцидентным вершине рёбрам. Поскольку в одном раунде вершина может получить сообщения сразу по нескольким рёбрам, порядок их обработки вершиной, в том числе, выбор «первого» из них, недетерминированы. Повторно получаемые вершиной сообщения дальше не пересылаются. Для различения первого и повторных сообщений используется булевская переменная ***было***; вначале ***было*** = ***false***, а при получении первого сообщения ***было*** := ***true***. Для определения номеров рёбер, по которым нужно посылать сообщение, используется переменная ***степень*** вершины размером $\boldsymbol{O}(log\Delta)$.

Рёбра, по которым сообщение впервые попадает в вершины, образуют остов графа. Если для ребра *ab* этого остова вершина *a* расположена по остову ближе к корню, чем вершина *b*, то есть вершина *b* получила первое сообщение от вершины *a*, это ребро будем называть *прямым* в вершине *a* и *обратным* в вершине *b*. Каждой некорневой вершине инцидентно ровно одно обратное ребро, и каждой нелистовой вершине (включая корень) инцидентно хотя бы одно прямое ребро. Остальные рёбра графа будем называть *хордами* остова.

Класс ***P*** реализует *broadcast* от корня до всех вершин с минимальным временем T, которое в $\mathcal{RSM}$ равно, а в $\mathcal{RAM}$ не превосходит d_0. Для того чтобы сообщение прошло по всем рёбрам, нужно ещё не более 1 такта, общее время $T^* \le T+1$. По каждому ребру в каждом направлении пройдёт ровно одно сообщение.

В описываемых ниже алгоритмах используются дополнительные опции класса ***P***, которые мы будем вводить по ходу изложения.

2. Класс ***O*** – пересылка по обратным рёбрам до корня. Сообщение этого класса проходит путь по обратным рёбрам от вершины, где оно создано, до корня. Для этого в каждой некорневой вершине сохраняется ***номер обратного ребра*** размером $\boldsymbol{O}(log\Delta)$. Эту переменную инициализирует опция *O* класса ***P*** как номер ребра, по которому вершина первый раз получает сообщение класса ***P***.

Остов, создаваемый обратными рёбрами, в $\mathcal{RSM}$ является деревом кратчайших путей и имеет высоту d_0. Однако в $\mathcal{RAM}$ это не обязательно так, там этот остов может иметь высоту до D_0. Поэтому класс ***O*** нам пригодится только для алгоритмов в $\mathcal{RSM}$, где он реализует *convergecast* с минимальным временем $T = d_0$. Для оценок в 0 предполагается, что каждая некорневая вершина создаёт сообщение, т.е. всего будет n-1 сообщений, а в графе на Рис. 1 все вершины полного подграфа могут создать сообщения одновременно. Поэтому по некоторым рёбрам остова может пройти, в том числе одновременно, $\boldsymbol{O}(n)$ сообщений, и каждое из таких сообщений может пройти путь длиной до d_0.

3. Класс ***C*** – сбор по остову. Это способ передачи сообщений от всех вершин к корню по обратным рёбрам, но в отличие от класса ***O***, с проходом по каждому обратному ребру ровно одного сообщения: сообщение посылается из вершины по обратному ребру только тогда, когда получены сообщения по всем прямым рёбрам. Для этого в каждой некорневой вершине имеется ***номер обратного ребра*** и ***число прямых рёбер*** оба размером $\boldsymbol{O}(log\Delta)$. Первую переменную инициализирует опция *O* класса ***P*** как номер ребра, по которому вершина первый раз получает сообщение класса ***P***. Вторую переменную инициализирует опция *C* класса ***P***. Для этого в сообщении класса ***P*** есть булевский параметр *признак остова*, который равен ***true*** тогда, когда сообщение посылается по обратному ребру. Вершина подсчитывает число полученных сообщений класса ***P*** с *признак остова* = ***true*** в переменной ***число прямых рёбер***. (Вместо опции *C* класса ***P*** может использоваться опция *П*, устанавливающая шкалу прямых рёбер, которую потом можно превратить в ***число прямых рёбер***. См. ниже класс ***П*** и алгоритмы типа **2** в 0.)

Остов из обратных рёбер в $\mathcal{RSM}$ является деревом кратчайших путей и имеет высоту d_0. Однако в $\mathcal{RAM}$ это не обязательно так, там этот остов может иметь высоту до D_0. Поэтому класс ***C*** нам пригодится только для алгоритмов в $\mathcal{RSM}$, где он реализует *convergecast* с минимальным временем $T = d_0$.

4. Класс ***П*** – рассылка по прямым рёбрам от корня. Сообщение этого класса рассылается, начиная с корня, по прямым рёбрам до листовых вершин остова. Для этого в каждой нелистовой вершине (включая корень) имеется битовая ***шкала прямых рёбер*** размером $\boldsymbol{O}(\Delta)$, содержащая «1» в *i*-ом разряде, если ребро с номером *i* прямое. Эту переменную инициализирует опция *П* класса ***P***. Сообщение класса ***P***, как и в случае опции *C*, имеет булевский параметр *признак остова*, который равен ***true*** тогда, когда сообщение посылается по обратному ребру. Получая по ребру сообщение класса ***P*** с *признак остова* = ***true***, вершина отмечает это ребро как прямое в шкале прямых рёбер.

Остов, создаваемый прямыми рёбрами, в $\mathcal{RSM}$ является деревом кратчайших путей и поэтому имеет высоту d_0. Однако в $\mathcal{RAM}$ это не обязательно так, там этот остов может иметь высоту до D_0. Поэтому класс $\boldsymbol{\Pi}$ нам пригодится только для алгоритмов в $\mathcal{RSM}$, где он реализует *broadcast* с минимальным временем $T = d_0$. По каждому ребру остова пройдёт ровно одно сообщение и только в направлении «прямое» (от корня).

5. Класс $\boldsymbol{M}$ – множественная рассылка. Множественная рассылка – это рассылка без повторения, которая параллельно ведётся, начиная с нескольких вершин, которые мы назовём *инициаторами*. Сообщение должно дойти либо из каждой вершины в каждую вершину (класс $\boldsymbol{M_в}$), либо из каждой вершины в корень (класс $\boldsymbol{M_к}$). Для класса $\boldsymbol{M_в}$ инициаторы – это все вершины, а для класса $\boldsymbol{M_к}$ – все вершины, кроме корня. Для того чтобы различать сообщения от разных инициаторов, используется ***номер инициатора***, который хранится в каждой вершине как её номер и имеет размер $\boldsymbol{O}(\mathit{logn})$, т.е. граф предполагается нумерованным. *Номер инициатора* также является параметром сообщения. Кроме того, в вершине для каждого инициатора должна быть своя переменная ***было***, т.е. вершина хранит множество номеров инициаторов, сообщения от которых были в вершине, в виде ***шкалы инициаторов*** размером $\boldsymbol{O}(n)$.

Класс $\boldsymbol{M}$ реализует *all-to-all-broadcast* с минимальным временем $T = d$. По каждому ребру в каждом направлении пройдёт ровно одно сообщение от каждого инициатора. Для оценок предполагается, что в графе на Рис. 1 все вершины полного подграфа могут создать сообщения одновременно.

3.3 Алгоритмы

Оценки алгоритмов приведены в 0. Обозначения: $f = min\{n,\Delta \mathit{logn}\}$, P – размер описания графа, необходимого для решения задачи, R – размер информации о разметке. Оценки как функции от n даны для общего случая, когда $P = R = \boldsymbol{O}(nf)$, они достигаются на графе на Рис. 1.

Табл. 2. Оценки алгоритмов.
Table 2. Estimations of the algorithms.

	1		**2**	**2a**	**2**	**2a**	**3**	
MODEL	*RSM*	*RAM*	*RSM*		*RAM*		*RSM*	*RAM*
Classes, ***P***(*options*)	$\boldsymbol{P}(O)+\boldsymbol{O}$	$\boldsymbol{P}+\boldsymbol{M_к}$	$\boldsymbol{P}(O\Pi)+\boldsymbol{O}$ $+\boldsymbol{\Pi}+\boldsymbol{C}$	$\boldsymbol{P}(O)+\boldsymbol{O}$ $+\boldsymbol{P}(OC)+\boldsymbol{C}$	$\boldsymbol{P}+\boldsymbol{M_к}$ $+\boldsymbol{P}(Б)+\boldsymbol{M_к}$	$\boldsymbol{P}+\boldsymbol{M_к}$ $+\boldsymbol{P}+\boldsymbol{M_к}$	$\boldsymbol{P}(OC)$ $+\boldsymbol{M_в}+\boldsymbol{C}$	$\boldsymbol{P}$ $+\boldsymbol{M_в}+\boldsymbol{M_к}$
A (not in root)	$\boldsymbol{O}(f)$ или $\boldsymbol{O}(\log n)$ *	$\boldsymbol{O}(n)$	$\boldsymbol{O}(\Delta\log n)$ $=\boldsymbol{O}(n\log n)$	$\boldsymbol{O}(f)$ или $\boldsymbol{O}(\log n)$ *	$\boldsymbol{O}(n+\Delta\log n)$ $=\boldsymbol{O}(n\log n)$	$\boldsymbol{O}(n)$	$\boldsymbol{O}(P+$ $+n+\Delta\log n)$ $=\boldsymbol{O}(n^2)$	
A (in root)	$\boldsymbol{O}(P+$ $+\log n)$ $=\boldsymbol{O}(n^2)$	$\boldsymbol{O}(P+$ $+n)$ $=\boldsymbol{O}(n^2)$	$\boldsymbol{O}(P+$ $+\Delta\log n)$ $=\boldsymbol{O}(n^2)$	$\boldsymbol{O}(P+$ $+\log n)$ $=\boldsymbol{O}(n^2)$	$\boldsymbol{O}(P+$ $+n+\Delta\log n)$ $=\boldsymbol{O}(n^2)$	$\boldsymbol{O}(P+$ $+n)$ $=\boldsymbol{O}(n^2)$		
M_1 and S_1	$\boldsymbol{O}(nf)=\boldsymbol{O}(n^2)$		$\boldsymbol{O}(R+nf)=\boldsymbol{O}(n^2)$				$\boldsymbol{O}(nf)=\boldsymbol{O}(n^2)$	
M_{all}	$\boldsymbol{O}(n\Delta\log n)$ $=\boldsymbol{O}(n^2\log n)$	$\boldsymbol{O}(\Delta n^2 f)$ $=\boldsymbol{O}(n^4)$	$\boldsymbol{O}(nR+$ $+n\Delta\log n)$ $=\boldsymbol{O}(n^3)$	$\boldsymbol{O}(n\Delta R+$ $+n\Delta\log n)$ $=\boldsymbol{O}(n^4)$	$\boldsymbol{O}(n\Delta R+$ $+\Delta n^2 f)$ $=\boldsymbol{O}(n^4)$		$\boldsymbol{O}(\Delta n^2 f)$ $=\boldsymbol{O}(n^4)$	
S_{all}	$\boldsymbol{O}(n\Delta\log n+$ $+d_0 nf)$ $=\boldsymbol{O}(n^3)$		$\boldsymbol{O}(n\Delta\log n+$ $+nR+d_0 nf)$ $=\boldsymbol{O}(n^3)$	$\boldsymbol{O}(n\Delta\log n+$ $+n\Delta R+d_0 nf)$ $=\boldsymbol{O}(n^4)$				
T	$\le 2d_0+1$		$\le 4d_0+1$	$\le 4d_0+2$	$\le 4d_0+1$	$\le 4d_0+2$	$\le 2d_0+d+1$ **	

* – при «экономной модификации», ** – очевидно, что $3d_0+1 \le 2d_0+d+1 \le 4d_0+1$.
Тёмно-серым фоном выделены оценки, в которые входят величины P или R.

3.3.1 Алгоритмы типа 1: Сбор в корне

Цель: *broadcast* с проходом по каждому ребру и *convergecast* за минимальное суммарное время $2d_0+1$. Используются сообщения ***Старт*** и ***ИнфоКорню***, которые имеют следующие классы и опции. В *RSM*: ***Старт*** – ***P*** с опцией *O* сохранения обратного ребра, ***ИнфоКорню*** – ***O***. В *RAM*: ***Старт*** – ***P*** без опций, ***ИнфоКорню*** – $\boldsymbol{M_к}$.

Граф предполагается нумерованным, каждая вершина хранит свой ***номер вершины*** размером $\boldsymbol{O}(\log n)$. Сообщение ***Старт*** имеет параметр: *номер вершины*, из которой оно посылается.

В каждой некорневой вершине имеется переменная ***множество соседей***, в которую помещается номер соседней вершины при получении сообщения ***Старт***. Это множество задаётся либо битовой шкалой вершин размером $\boldsymbol{O}(n)$, в которой i-ый разряд равен 1, если вершина с номером i является соседом вершины a, либо списком соседей размером $\boldsymbol{O}(\Delta\log n)$, в котором j-ый элемент списка содержит номер вершины на другом конце ребра, имеющего номер j в a. Переменная ***множество соседей*** имеет размер $\boldsymbol{O}(f)$, где $f=\min\{n,\Delta\log n\}$.

Сообщение ***ИнфоКорню*** вершина создаёт тогда, когда получит сообщение ***Старт*** от всех соседей, что определяется сравнением мощности множества соседей и степени вершины. Это сообщение имеет параметры: *номер вершины*, которая его создала, и *множество соседей*, равное значению переменной ***множество соседей*** этой вершины.

Корень сохраняет информацию о графе в виде ***списка множеств соседей*** размером $P = \boldsymbol{O}(nf)$, в котором i-ый элемент списка – это множество соседей вершины с номером i; это множество пусто, если соседи вершины ещё не известны. Длина списка равна максимальному номеру известной корню вершины. Первоначально корню известен только его номер, а некорневая вершина становится известной корню, когда корень получает от неё сообщение ***Старт*** или ***ИнфоКорню***, или когда она описана как соседняя вершина в сообщении ***ИнфоКорню***. Когда корень r получает сообщение ***Старт*** от вершины j, он добавляет j в своё множество соседей в списке. Когда корень получает сообщение ***ИнфоКорню*** от вершины i, i-ое множество соседей из списка объединяется с множеством соседей из сообщения.

Список множеств соседей описывает весь граф, если корень получил сообщение ***Старт*** по каждому инцидентному ему ребру, и от каждой известной корню некорневой вершины, т.е. от некорневой вершины с номером i от 1 до длины списка, получено сообщение ***ИнфоКорню***, т.е. i-ое множество соседей в списке не пусто. В этом случае корень решает задачу и посылает её решение вовне в завершающем сообщении.

Оценка времени $T = 2d_0+1$ достигается на любом графе, в котором есть ребро, соединяющее две вершины, каждая на расстоянии d_0 от корня. Такие вершины и рёбра будем называть *периферийными*. Если периферийных рёбер нет, то $T = 2d_0$. Максимальный размер сообщения: для ***Старт*** $m = \boldsymbol{O}(logn)$, для ***ИнфоКорню*** $m = \boldsymbol{O}(f)$.

«Экономная модификация» алгоритма в $\mathcal{RSM}$. В $\mathcal{RSM}$ можно уменьшить память вершины, если сообщение ***ИнфоКорню*** посылать «без накопления» информации о соседях, т.е. сразу при получении сообщения ***Старт***. Переменная ***множество соседей*** не нужна, а параметр *множество соседей* сообщения ***ИнфоКорню*** равен множеству номеров вершин, от которых в данном раунде вершина получила сообщение ***Старт***. Если вершина находится на расстоянии r от корня, то её соседи находятся от корня на расстоянии r-1, r или r+1. Поэтому сообщение ***ИнфоКорню*** создаётся вершиной не более чем на трёх раундах, и суммарный размер этих сообщений всё равно равен $\boldsymbol{O}(f)$. Поскольку число таких раундов колеблется от 1 до 3, последнее сообщение от данной вершины ***ИнфоКорню*** маркируется булевским параметром *последнее сообщение*. Для этого вершина подсчитывает число полученных ею сообщений ***Старт*** в переменной ***счётчик соседей*** размером $\boldsymbol{O}(log\Delta)$: сообщение ***ИнфоКорню*** последнее, если при его создании счётчик соседей равен степени вершины. Корень дополнительно запоминает, было ли полученное сообщение ***ИнфоКорню*** последним от создавшей его вершины i, в переменной ***шкала последних*** размером $\boldsymbol{O}(n)$, i-ый разряд которой соответствует вершине i. При определении конца сбора информации корень учитывает, что от каждой некорневой вершины должно быть получено последнее сообщение ***ИнфоКорню***. Таким образом, за счёт нескольких (но не больше трёх)

сообщений ***ИнфоКорню*** от одной вершины мы экономим память вершины: вместо $A = \boldsymbol{O}(f)$ будет $A = \boldsymbol{O}(\log n)$.

Заметим, что в *RAM* такая модификация не имеет смысла. Дело в том, что в *RAM* на каждом раунде вершина может получать не более одного сообщения ***Старт***. Если создавать сообщение ***ИнфоКорню*** каждый раз при получении сообщения ***Старт***, то число таких сообщений ***ИнфоКорню*** может достигать степени вершины, а их суммарный размер достигать $\boldsymbol{O}(\Delta f)$, что увеличивает оценки M_1, M_{all}, S_1, S_{all} в $\Delta = \boldsymbol{O}(n)$ раз. Кроме того, в *RAM* сообщение ***ИнфоКорню*** посылается множественной рассылкой, а это предполагает либо одно сообщение от вершины, либо увеличение числа инициаторов в Δ раз, то есть, по сути, инициатором становится не вершина, а ребро, и размер шкалы инициаторов увеличивается с $\boldsymbol{O}(n)$ до $\boldsymbol{O}(\Delta n)$.

3.3.2 Алгоритмы типа 2: Сбор в корне и разметка из корня

Цель: сбор информации – *broadcast* с проходом по каждому ребру и *convergecast* за минимальное суммарное время $2d_0+1$, и разметка от корня – *broadcast* и *convergecast* за минимальное суммарное время $2d_0$. Используются сообщения ***Старт***, ***ИнфоКорню***, ***Разметка*** и ***Финиш***, которые имеют следующие классы и опции: В *RSM*: ***Старт*** – ***Р*** с опцией *О* сохранения обратного ребра и опцией *П* создания шкалы прямых рёбер, ***ИнфоКорню*** – ***О***, ***Разметка*** – ***П***, ***Финиш*** – ***С***. В *RAM*: ***Старт*** – ***Р*** без опций, ***ИнфоКорню*** – $\boldsymbol{M_к}$, ***Разметка*** – ***Р*** с опцией *Б*, ***Финиш*** – $\boldsymbol{M_к}$. Опция *Б* состоит в том, что сообщение не посылается по обратному ребру.

Граф предполагается нумерованным, каждая вершина хранит свой номер размером $\boldsymbol{O}(\log n)$.

Сбор информации в корне выполняется аналогично тому, как это делается в алгоритмах типа **1**, за исключением следующего. Во-первых, для удобства последующей разметки при обработке сообщения ***Старт*** вершина сохраняет отображение номера ребра на номер соседа на другом конце этого ребра в переменной ***соответствие*** размером $\boldsymbol{O}(\Delta \log n)$. Из-за этого переменная ***множество соседей*** становится лишней, поскольку её значение восстанавливается по переменной ***соответствие***. Во-вторых, в *RSM* сообщение ***Старт*** имеет опцию *П* для формирования ***шкалы прямых рёбер***, которая затем используется для рассылки сообщения ***Разметка*** класса ***П***. В-третьих, поскольку в памяти вершины уже есть переменная ***соответствие*** размером $\boldsymbol{O}(\Delta \log n)$, не имеет смысла «экономная модификация» в *RSM* из 0. В-четвёртых, когда в *RSM* вершина получает сообщение ***Разметка*** и пересылает его дальше по прямым рёбрам, она на месте переменной ***шкала прямых рёбер*** создаёт переменную ***число прямых рёбер***, которая в дальнейшем используется при обработке сообщения ***Финиш*** класса ***С***.

После сбора информации решается задача, формируется и рассылается сообщение ***Разметка***. Сообщение ***Разметка*** содержит в качестве параметра

информацию о разметке размером $R = \boldsymbol{O}(nf)$. Информация о разметке вершин задаётся шкалой вершин, которые нужно пометить, размером $\boldsymbol{O}(n)$, а информация о разметке рёбер задаётся в виде списка множеств соседей размером $\boldsymbol{O}(nf)$, в котором для каждой вершины i в её множестве соседей оставлены только те вершины j, для которых нужно пометить ребро ij. Получив первое сообщение ***Разметка***, вершина с номером i выполняет свою локальную разметку, определяемую i-ым разрядом шкалы вершин и i-ым элементом списка множеств соседей. Для пометки ребра ij вершина должна узнать его номер в i, что делается по отображению номера ребра в номер соседа в переменной ***соответствие***.

В *RSM* сообщение ***Финиш*** не имеет параметров и относится к классу $\boldsymbol{C}$, поэтому по каждому прямому ребру в корень придёт ровно одно сообщение ***Финиш***. Когда корень получит все эти сообщения, глобальная разметка закончена. В *RAM* сообщение ***Финиш*** относится к классу $\boldsymbol{M_к}$ и содержит номер инициатора, поэтому корень может использовать шкалу инициаторов: число «1» в ней равно числу вершин, от которых пришли сообщения ***Финиш***. Глобальная разметка закончена, когда это число становится равным числу некорневых вершин, т.е. на 1 меньше длины списка множеств соседей. После окончания глобальной разметки корень посылает вовне сообщение о завершении разметки и заканчивает работу.

Максимальный размер сообщения: для ***Старт*** $m = \boldsymbol{O}(log n)$, для ***ИнфоКорню*** $m = \boldsymbol{O}(f)$, для ***Разметка*** $m = R = \boldsymbol{O}(nf)$, для ***Финиш*** в *RSM* $m = \boldsymbol{O}(1)$, а в *RAM* $m = \boldsymbol{O}(log n)$.

3.3.3 Алгоритмы типа 2a: Уменьшение памяти вершины

Цель: модификация **2a** алгоритмов типа **2** для уменьшения памяти некорневой вершины за счёт удаления переменной ***соответствие*** размером $\boldsymbol{O}(\Delta log n)$. Для этого достаточно сообщение ***Разметка*** посылать способом ***P*** без опции *Б* с дополнительным параметром *номер вершины*-отправителя размером $\boldsymbol{O}(log n)$. Каждая вершина получит это сообщение по каждому инцидентному ей ребру. Получая сообщение ***Разметка*** от вершины j по ребру k, вершина i, во-первых, узнаёт соответствие j и k, а, во-вторых, по информации о разметке узнаёт, нужно ли помечать ребро ij. Тем самым, она может пометить ребро с номером k нужной меткой.

Сообщения имеют следующие классы и опции. В *RSM*: ***Старт*** – ***P*** с опцией *O* сохранения обратного ребра и опцией *C* подсчёта числа прямых рёбер, ***ИнфоКорню*** – ***O***, ***Разметка*** – ***P*** с опциями *O* и *C*, ***Финиш*** – ***C***. В *RAM*: ***Старт*** – ***P*** без опций, ***ИнфоКорню*** – $\boldsymbol{M_к}$, ***Разметка*** – ***P*** без опций, ***Финиш*** – $\boldsymbol{M_к}$.

Поскольку мы удаляем переменную ***соответствие***, снова нужна переменная ***множество соседей*** размером $\boldsymbol{O}(f)$, кроме случая «экономной модификации» в *RSM* (0). Эта модификация снова имеет смысл: сообщение ***ИнфоКорень*** можно посылать «без накопления» информации о соседях, т.е. сразу при

получении сообщения ***Старт***, используются переменные ***счётчик соседей*** и, в корне, ***шкала последних***, а также параметр *последнее сообщение* в сообщении ***ИнфоКорень***.

По сравнению с типом **2** время работы увеличивается не более чем на 1 такт (или остаётся тем же, если нет периферийных рёбер). Максимальные размеры сообщений m такие же как для типа **2**.

3.3.4 Алгоритмы типа 3: Сбор и разметка во всех вершинах

Цель: сбор информации в каждой вершине – *broadcast* с проходом по каждому ребру и *all-to-all-broadcast* за минимальное суммарное время d_0+1+d, и сообщение корню о завершении разметки – *convergecast* за минимальное время d_0. Используются сообщения ***Старт***, ***ИнфоВсем*** и ***Финиш***, которые имеют следующие классы и опции. В $\mathcal{RSM}$: ***Старт*** – ***P*** с опцией *O* сохранения обратного ребра и *C* подсчёта числа прямых рёбер, ***ИнфоВсем*** – $\boldsymbol{M_в}$, ***Финиш*** – ***C***. В $\mathcal{RAM}$: ***Старт*** – ***P*** без опций, ***ИнфоВсем*** – $\boldsymbol{M_в}$, ***Финиш*** – $\boldsymbol{M_к}$.

Граф предполагается нумерованным, каждая вершина хранит свой номер размером $\boldsymbol{O}(\mathit{logn})$. Как и в алгоритмах типа **2** сообщение ***Старт*** содержит *номер вершины*, из которой оно посылается, а при обработке сообщения ***Старт*** вершина сохраняет отображение номера ребра на номер соседа на другом конце этого ребра в переменной ***соответствие*** размером $\boldsymbol{O}(\Delta\mathit{logn})$. Как и в алгоритмах типа **2** после того как вершина получит сообщение ***Старт*** от всех своих соседей, она формирует сообщение, но не ***ИнфоКорню***, а ***ИнфоВсем***, помещая в него *номер вершины* из переменной ***номер вершины*** и *множество соседей*, формируемое по переменной ***соответствие***. Кроме этого, вершина поддерживает ***список множеств соседей*** и определяет конец сбора информации о графе так же, как корень в алгоритмах типа **2** и **2a**. После того как информация о графе собрана, вершина решает задачу и выполняет свою локальную разметку. Напомним, что алгоритм решения задачи должен гарантировать, что в разных вершинах будет получаться одно и то же решение. После этого вершина посылает корню сообщение ***Финиш*** аналогично алгоритмам типа **2** и **2a**. Заметим, что в $\mathcal{RAM}$ это сообщение посылается множественной рассылкой ($\boldsymbol{M_к}$) так же как сообщение ***ИнфоВсем*** ($\boldsymbol{M_в}$), поэтому в каждой вершине нужно иметь две шкалы инициаторов – по одной для каждого из этих двух типов сообщений: ***Финиш*** и ***ИнфоВсем***.

Корень определяет конец работы, когда, во-первых, сам получит информацию о всём графе, и, во-вторых, получит все нужные сообщения ***Финиш***, что определяется так же как в алгоритмах типа **2** и **2a**.

Заметим, что «экономная модификация» в $\mathcal{RSM}$ для сбора информации не имеет смысла, поскольку она уменьшает память некорневой вершины с $\boldsymbol{O}(f)$ до $\boldsymbol{O}(\mathit{logn})$, но в алгоритмах типа *3* все вершины имеют память размера $\boldsymbol{O}(P+n+\Delta\mathit{logn})$.

Максимальный размер сообщения: для ***Старт*** $m = \boldsymbol{O}(logn)$, для ***ИнфоВсем*** $m = \boldsymbol{O}(f)$, для ***Финиш*** в $\mathcal{RSM}$ $m = \boldsymbol{O}(1)$, а в $\mathcal{RAM}$ $m = \boldsymbol{O}(logn)$.

4. Экономия памяти для задач FSB, MST и MIS

В алгоритмах для общего случая размер памяти вершины имеет порядок $\boldsymbol{O}(P) = \boldsymbol{O}(nf) = \boldsymbol{O}(n^2)$ в корне или во всех вершинах за счёт хранения списка множеств соседей. Такой же по порядку размер имеет информация о разметке. Для некоторых задач можно уменьшить эти размеры, используя вместо списка множеств соседей другую структуру меньшего размера.

4.1 Определение конца сбора информации о графе и конца работы

Список множества соседей используется также для определения конца сбора информации о графе. Можно предложить альтернативный способ: используются две битовые шкалы вершин, каждая размером $\boldsymbol{O}(n)$: ***шкала всех вершин***, содержащая корень и вершины, от которых получены сообщения ***ИнфоКорню*** в алгоритмах типа **1** и **2** или вершины, от которых получены сообщения ***ИнфоВсем*** в алгоритмах типа **3**, и ***шкала всех соседей*** этих вершин. Конец сбора информации определяется по совпадению этих шкал. В алгоритмах типа **2** и **3** с этого момента времени корень знает число вершин (оно равно числу «1» в этих шкалах), что используется в $\mathcal{RAM}$ для определения конца приёма сообщений ***Финиш*** от всех некорневых вершин.

4.2 FSB

Мост – это ребро, не входящее в цикл, поэтому для определения мостов можно использовать следующий способ. Вместо списка множеств соседей хранится лес деревьев, вначале пустой. При получении вершиной *a* сообщения ***ИнфоКорню*** или ***ИнфоВсем***, в котором указана соседняя вершина *b*, проверяется ребро *ab*. Если при добавлении этого ребра к лесу цикл не образуется, ребро добавляется к лесу. В противном случае все рёбра этого цикла помечаются как «не мосты», а ребро *ab* к лесу не добавляется. В конце работы все рёбра леса, не помеченные как «не мосты», – это все мосты графа.

Для хранения остова достаточно памяти размером $\boldsymbol{O}(nlogn) = \boldsymbol{o}(nf)$ при $\Delta \to \infty$ (и, поскольку $\Delta \le n\text{-}1$, также $n \to \infty$). Информация о разметке содержит описание мостов как части рёбер остова, также размером $\boldsymbol{O}(nlogn) = \boldsymbol{o}(nf)$. Получаются варианты алгоритмов с оценками из 0 для $R = P = \boldsymbol{O}(nlogn)$.

4.3 MST

Для построения MST можно использовать известный критерий Тарьяна минимальности остовного дерева: остовное дерево минимально тогда и только тогда, когда любое ребро не из дерева является максимальным по весу на

цикле, который образуется при его добавлении в дерево. Как и для FSB вместо списка множеств соседей хранится лес деревьев, вначале пустой. При получении от вершины *a* сообщения ***ИнфоКорню*** или ***ИнфоВсем***, в котором указана соседняя вершина *b*, проверяется ребро *ab*. Если при добавлении этого ребра к лесу цикл не образуется, ребро добавляется к лесу. В противном случае проверяется, какое ребро в этом цикле максимально по весу. Если это ребро $a`b` \neq ab$, оно удаляется из леса, а ребро *ab* добавляется к лесу.

Задача MST формулируется для взвешенного графа, поэтому список множеств соседей с указанием весов рёбер до них имеет размер $\boldsymbol{O}(nf_w)$, где $f_w = min\{nlogw,\Delta logn w\}$. Для хранения взвешенного остова достаточно памяти размером $\boldsymbol{O}(nlognw) = \boldsymbol{o}(nf_w)$ при $\Delta\to\infty$ (и, поскольку $\Delta \le n\text{-}1 \le w$, также $n\to\infty$ и $w\to\infty$). Информация о разметке содержит описание рёбер остова размером $\boldsymbol{O}(nlogn) = \boldsymbol{o}(nf)$. Получаются варианты алгоритмов общего типа с оценками из 0 для $P = \boldsymbol{O}(nlognw)$ и $R = \boldsymbol{O}(nlogn)$, и заменой *f* на f_w.

4.4 MIS

Для построения MIS можно использовать хорошо известную стандартную процедуру. Вместо списка множеств соседей в корне используется ***шкала вершин независимого множества***, представляющая независимое множество вершин *IS*. Вначале шкала содержит только корень. При получении корнем сообщения ***ИнфоКорню*** или ***ИнфоВсем*** от вершины *x* по множеству соседей в этом сообщении проверяется, смежна ли вершина *x* с какой либо вершиной из *IS* или нет. Если *x* не смежна ни с какой вершиной из *IS*, то *x* добавляется к множеству *IS*, т.е. в ***шкалу вершин независимого множества***. Однако применение такой процедуры имеет два ограничения.

Во-первых, в $\mathcal{RSM}$ в алгоритмах типа **1** и **2a** нельзя использовать «экономную модификацию», так как нельзя определить, смежна или не смежна вершина *x* с какой либо вершиной из *IS*, до получения всех сообщений ***ИнфоКорню*** от *x*.

Во-вторых, для алгоритмов типа **3** эта процедура не применима. Дело в том, что MIS в отличие от MST и FSB определяется в графе, вообще говоря, неоднозначно. Процедура основана на линейном упорядочении всех вершин в соответствии с порядком получения от них сообщений ***ИнфоКорню*** или ***ИнфоВсем***. Однако такой порядок, вообще говоря, разный для разных вершин, и поэтому они могут строить разные MIS'ы. Для того, чтобы вершины строили один и тот же MIS, в вершинах должен использоваться один и тот же линейный порядок вершин, порождающий, так называемый, лексикографический MIS (LFMIS). Однако для этого, скорее всего, потребуется больше памяти, скорее всего, порядка размера графа $\boldsymbol{O}(nf)$.

Если вершина собрала описание всего графа, она может определить все возможные MIS`ы. Если каждый MIS упорядочить, т.е. преобразовать в последовательность, например, по возрастанию номеров вершин, то можно выбрать MIS являющийся наибольшим в лексикографическом порядке.

Заметим, что такой MIS является наибольшим по числу вершин в нём. Тем самым, решается более сложная задача поиска MaxIS (maximim independent set). Эта задача может решаться в алгоритмах всех типов **1**, **2**, **2a**, **3**.

Для хранения *IS* в виде шкалы вершин достаточно памяти размером $\boldsymbol{O}(n) = \boldsymbol{o}(nf)$ при $n \to \infty$. Информация о разметке содержит MIS также в виде шкалы вершин размером $\boldsymbol{O}(n) = \boldsymbol{o}(nf)$. Получаются варианты алгоритмов с оценками из 0 для $R = P = \boldsymbol{O}(n)$.

5. Нижние оценки сложности для FSB, MST и MIS

Если решение задачи зависит от периферийных рёбер, то, очевидно, $T \le 2d_0+1$. Задачи FSB, MST и MIS зависят от периферийных рёбер, что показывается примерами, в которых два графа отличаются друг от друга только периферийными рёбрами, а задачи имеют на этих графах разные решения.

FSB: 1) дерево, в котором есть, по крайней мере, две периферийные вершины, и 2) такое же дерево с добавлением периферийного ребра. В случае 1 все рёбра дерева являются мостами, а во втором случае все рёбра цикла, образуемого этим периферийным ребром и деревом, мостами не являются.

MST: 1) дерево, в котором есть, по крайней мере, две периферийные вершины, и 2) такое же дерево с добавлением периферийного ребра, имеющего вес меньше, чем какое-нибудь ребро дерева из цикла, образуемого этим периферийным ребром и деревом. В случае 1 все рёбра дерева принадлежат MST, а во втором случае периферийное ребро принадлежит MST, а некоторое ребро дерева не принадлежит MST.

MIS: пример на Рис. 2. Одна из вершин *a* или *b* должна не принадлежать MIS, так как они соединены ребром. Пусть, для определённости, это будет вершина *a*. Посмотрим, что можно сказать о вершинах, обозначенных знаком «?». Если они не соединены ребром, то обе должны принадлежать MIS, а иначе только одна из них. Но это ребро периферийное.

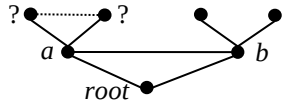


Рис. 2. Зависимость MIS от периферийных рёбер.

Fig. 2. Dependence of MIS on peripheral edges.

6. Алгоритмы типа 4 в $\mathcal{RSM}$ для задач с однозначным решением, зависящим только от простых циклов

Будем говорить, что задача на графе имеет однозначное решение, зависящее только от простых циклов графа, если локальная разметка в вершине зависит только от простых циклов, в которые входит эта вершина. К таким задачам относятся, например, FSB, MST и задача о гамильтоновом цикле.

Для FSB ребро, инцидентное вершине, является мостом тогда и только тогда, когда оно не входит ни в какой простой цикл, проходящий через вершину. Для MST ребро, инцидентное вершине, принадлежит MST тогда и только тогда, когда в каждом простом цикле, проходящем через вершину, это ребро не имеет наибольшего веса.

Гамильтонов цикл существует не во всяком графе, но в некоторых графах (например, в полном графе при $n > 3$) может быть несколько разных гамильтоновых циклов. В нумерованном графе всегда есть возможность однозначно выбрать один из гамильтоновых циклов. Для этого достаточно каждый гамильтонов цикл представить в виде последовательности номеров вершин, начиная с вершины с наименьшим номером 1 и выбирая направление обхода в сторону той из двух вершин, смежной по циклу с вершиной с номером 1, которая имеет наименьший номер. После этого однозначно выбирается гамильтонов цикл как наименьший в лексикографическом порядке таких последовательностей.

Для подобного рода задач можно предложить в $\mathcal{RSM}$ алгоритмы типа **4** как модификацию алгоритмов типа **3** с временем $T = 2d_0+1$ или $T = 2d_0$, если нет периферийных рёбер. Идея алгоритмов основана на том факте, что расстояние между вершинами в одном цикле ограничено сверху не $d \le 2d_0$, а примерно в два раза меньшей величиной. Покажем формально, что каждая вершина узнает о всех простых циклах, в которые она входит, не позже, чем корень узнает весь граф. Как показано в п.0, корень узнаёт весь граф через время $2d_0+1$ или $2d_0$, если нет периферийных рёбер.

Действительно, пусть ребро ab и вершина c входят в простой цикл (Рис. 3). Нам достаточно показать, что время t, через которое вершина c узнает о ребре ab, не больше $2d_0+1$ или $2d_0$, если нет периферийных рёбер. Обозначим: d_{ac} – расстояние по циклу от a до c, d_{bc} – расстояние по циклу от b до c, d_a – расстояние от a до корня, d_b – расстояние от b до корня. Тогда $t \le min\{d_a+1+d_{bc},\ d_b+1+d_{ac}\}$. Но $d_a \le d_0$ и $d_b \le d_0$. Поэтому $min\{d_a+1+d_{bc}, d_b+1+d_{ac}\} \le d_0 + 1 + min\{d_{bc}, d_{ac}\}$. Так как цикл простой, его длина $d_{ac} + d_{bc} + 1 \le d+1 \le 2d_0+1$, поэтому $min\{d_{bc}, d_{ac}\} \le d_0$. Тем самым, $t \le 2d_0+1$. Но если периферийных рёбер нет, то либо $d_a < d_0$, либо $d_b < d_0$. Пусть для определённости $d_a < d_0$. Если $d_{bc} \le d_0$, то $d_a+1+d_{bc} < 2d_0 + 1$. Если $d_{bc} > d_0$, то $d_{ac} < d_0$ (так как $d_{ac} + d_{bc} \le 2d_0$), поэтому $d_b+1+d_{ac} < 2d_0 + 1$. Тем самым, если периферийных рёбер нет, то $t \le 2d_0$.

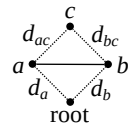


Рис. 3. Вершина c и ребро ab входят в один цикл.

Fig. 3. The vertex c and the edge ab enter into one cycle.

Алгоритмы типа **4** используют сообщения ***Старт*** класса ***P*** с опцией *O* и ***ИнфоВсем*** класса $\boldsymbol{M_e}$, сообщение ***Финиш*** не используется. Поведение каждой вершины зависит от решаемой задачи.

Для FSB каждая вершина помечает все инцидентные ей рёбра как «мосты», а потом снимает эту пометку для ребра, который оказывается в некотором цикле. Для MST каждая вершина помечает все инцидентные ей рёбра как «принадлежащие MST», а потом снимает эту пометку для ребра, который оказывается ребром с наибольшим весом в некотором цикле. Для гамильтонова цикла каждая вершина на каждом раунде, узнав о части графа, выбирает в ней простой цикл, в который она входит, наибольшей длины и в соответствии с лексикографическим порядком соответствующей циклу последовательности номеров вершин. Если такой цикл есть, вершина помечает два инцидентных ей ребра, входящих в этот цикл, как принадлежащие гамильтонову циклу (если на предыдущем раунде были помечены какие-то другие два ребра, с них пометка снимается).

Когда корень узнаёт весь граф, ему не нужно дожидаться извещения от всех вершин о выполнении локальной разметки: он может сразу послать вовне завершающее сообщение о том, что задача решена. Для FSB или MST корень посылает вовне сообщение «мосты размечены» или «MST размечено», соответственно. При поиске гамильтонова цикла корень сначала проверяет существование гамильтонова цикла по собранной информации о графе. Если его нет, корень посылает вовне сообщение «гамильтонова цикла нет», а если есть, корень посылает вовне сообщение «гамильтонов цикл есть и размечен».

Для алгоритмов типа **4** по сравнению с алгоритмами типа **3** в $\mathcal{RSM}$ меняется только время $T \le 2d_0+1$, а остальные оценки из 0 сохраняются.

Заметим, что в $\mathcal{RAM}$ аналогичный алгоритм не работает, так как из-за различных и переменных временах перемещения сообщений по рёбрам корень может узнать о всём графе до того, как все вершины узнали о простых циклах, в которые они входят, т.е. до того, как в них завершена локальная разметка.

7. Алгоритмы типа 5 – разметка во время сбора информации для FSB и MST

Для задач FSB и MST можно предложить алгоритмы типа **5** как модификацию алгоритмов типа **1** (без «экономной модификации»), которые размечают граф за одну пару *broadcast* + *convergecast*. Сообщение ***Старт*** имеет класс ***P*** с опцией *O* сохранения обратного ребра, а сообщение ***ИнфоКорню*** – в $\mathcal{RSM}$ класс ***C***, а в $\mathcal{RAM}$ – класс $\boldsymbol{M_k}$. Отличие от алгоритмов типа **1** в том, что там сообщение ***Старт*** в $\mathcal{RAM}$ имел класс ***P*** без опций. Как и для алгоритмов типа **1** сообщение ***Старт*** имеет параметр *номер вершины*, а сообщение ***ИнфоКорню*** – параметры *номер вершины* и *множество соседей*. Также эти сообщения имеют дополнительные параметры, описываемые ниже.

Конец работы корень определяет так, как описано выше в п.0, с помощью двух битовых шкал: ***шкалы всех вершин*** и ***шкала всех соседей***.

Идея алгоритма основана на использовании *векторов*, где *вектором маршрута* называется последовательность номеров рёбер маршрута, который проходится сообщением. Если ребро *ab* проходится в направлении $a \to b$, то в вектор помещается номер этого ребра в вершине *a*. Вектор пути от корня имеет размер $\boldsymbol{O}(d_0 log \Delta)$ (в $\mathcal{RAM}$ $\boldsymbol{O}(D_0 log \Delta)$, но так как $D_0 \le 2d_0$, то $\boldsymbol{O}(D_0 log \Delta) = \boldsymbol{O}(d_0 log \Delta)$). Для взвешенного графа вектор, кроме номера ребра, содержит его вес, будем называть такой вектор с весами рёбер *взвешенным вектором*, он имеет размер $\boldsymbol{O}(d_0 log \Delta w)$. Сообщение ***Старт*** содержит дополнительный параметр: *вектор маршрута*, который сообщение проходит. *Вектор вершины a* – это вектор маршрута из первого полученного вершиной *a* сообщения ***Старт***, он хранится в переменной ***вектор вершины*** и далее обозначается как v_a, а его длина – как r_a. Заметим, что (взвешенный) вектор вершины – это последовательность номеров (и весов) прямых рёбер от корня до вершины.

Рассмотрим ситуацию, когда вершина *a* получает по ребру *ab* повторное сообщение ***Старт*** с вектором маршрута *v*. Если $v = v_a \cdot i \cdot j$, то ребро *ab* – прямое в *a* с номером *i* и обратное в *b* с номером *j*. В противном случае ребро *ab* – хорда. Каждая хорда *ab* образует цикл, состоящий из этой хорды и двух путей по остову от вершин *a* и *b* до *развилки c*, где развилка *c* – это такая вершина, что путь по остову от корня до *c* – это наибольший общий префикс путей по остову от корня до вершин *a* и *b* (Рис. 4). Вектор развилки v_c есть наибольший общий префикс векторов v_a и v_b.

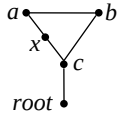


Рис. 4. Хорда ab и развилка c.

Fig. 4. Chord ab and fork c.

Поведение вершины *a* при обнаружении хорды *ab* с развилкой *c* зависит от решаемой задачи. Для FSB каждое ребро цикла нужно пометить как «не мост» в обоих его концах. При этом вершина *a* должна обеспечить пометку хорды *ab* в вершине *a* и каждого ребра на пути от *a* до *c* в обоих его концах. Для MST нужно пометить как «не принадлежащее MST» ребро цикла с наибольшим весом. Вершина *a* должна обеспечить пометку этого ребра при условии, что это хорда (помечается в вершине *a*) или ребро принадлежит пути от *a* до *c* (помечается в обоих его концах). Для обеих задач при обнаружении хорды *ab* вершиной *b*, вершина *b* обеспечивает выполнение аналогичной работы.

Вершина *a* помечает хорду *ab* (для MST если её нужно пометить) сразу, как только обнаруживает эту хорду. А *информацию о разметке остова*, т.е. о том, какие рёбра нужно пометить на пути от *a* до *c*, вершина *a* помещает в

параметры создаваемого ею сообщения ***ИнфоКорню***. Однако это сообщение создаётся вершиной a после обнаружения не одной хорды ab, а всех хорд, инцидентных a, т.е. после получения вершиной a сообщения ***Старт*** по всем инцидентным вершине a рёбрам. Поэтому информация о разметке остова для всех хорд $ab_1,ab_2,...ab_k$ с развилками $c_1,c_2,...c_k$ объединяется. Заметим, что все эти развилки $c_1,c_2,...c_k$ располагаются на одном пути от корня до a. Пусть c – ближайшая к корню развилка среди развилок $c_1,c_2,...c_k$ т.е. имеющая вектор наименьшей длины $r = min\{r_{c_1},r_{c_2},...r_k\}$.

Для FSB результат объединения информации о разметке остова – это все рёбра на пути от a до c. Достаточно указать путь по остову от корня до a, т.е. вектор v_a, и число r как индекс по вектору v_a: пометить нужно все рёбра, соединяющие вершины с векторами $v_a[1..j]$ и $v_a[1..j+1]$, где $r_a > j \geq r$.

Для MST результатом объединения информации о разметке остова является множество рёбер на пути от a до c. Достаточно указать вектор v_a и множество индексов $\{r_1,r_2,...r_k\}$ по вектору v_a: пометить нужно ребро, соединяющие вершины с векторами $v_a[1..\ r_j]$ и $v_a[1..r_j+1]$, где $j = 1..k$. Множество $\{r_1,r_2,...r_k\}$ можно задавать битовой шкалой размером $\boldsymbol{O}(d_0)$.

Информация о разметке остова зависит также от модели.

В $\mathcal{RSM}$ сообщение ***ИнфоКорню*** имеет класс $\boldsymbol{C}$ и посылается по обратному ребру после того, как получены сообщения ***ИнфоКорню*** по всем прямым рёбрам. Поэтому в вершине происходит объединение информации о разметке остова из всех принимаемых сообщений ***ИнфоКорню***, результат такого объединения всё равно имеет вид (v_a,r) для FSB или $(v_a,\{r_1,r_2,...r_k\})$ для MST. Кроме того, сообщение ***ИнфоКорню*** двигается из вершины a к корню по остову, поэтому вектор v_a можно не указывать. Таким образом, в $\mathcal{RSM}$ у сообщения ***ИнфоКорню*** есть параметр *информация о разметке остова* вида r размером $\boldsymbol{O}(log d_0)$ для FSB или $\{r_1,r_2,...r_k\}$ размером $\boldsymbol{O}(d_0)$ для MST. Вершина x, получая по ребру i сообщение ***ИнфоКорню***, помечает как «не мост» обратное ребро, если для FSB $r < r_x$ или для MST $r_j+1 = r_x$ для некоторого j, и ребро i, если для FSB $r \leq r_x$ или для MST $r_j = r_x$ для некоторого j.

В $\mathcal{RAM}$ сообщение ***ИнфоКорню*** распространяется множественной рассылкой, поэтому вектор v_a нужно указывать. Таким образом, в $\mathcal{RAM}$ у сообщения ***ИнфоКорню*** есть параметр *информация о разметке остова* вида (v_a,r) размером $\boldsymbol{O}(d_0 log\Delta+log d_0) = \boldsymbol{O}(d_0 log\Delta)$ для FSB или $(v_a,\{r_1,r_2,...r_k\})$ размером $\boldsymbol{O}(d_0 log\Delta+d_0) = \boldsymbol{O}(d_0 log\Delta)$ для MST. Вершина x, получая по ребру i сообщение ***ИнфоКорню***, делает такие же пометки как в $\mathcal{RSM}$, но только при условии, что она лежит на пути от корня до a, т.е. $v_x \leq v_a$.

Для алгоритмов типа **5** оценки приведены в 0.

Табл. 3. Оценки алгоритмов типа 5.
Table 3. Estimations of algorithms of type 5.

	FSB	***MST***	***FSB***	***MST***
MODEL	*RSM*		*RAM*	
Classes, P(*options*)	***P***(*O*)+***C***		***P***(*O*)+$\boldsymbol{M_к}$	
A	$\boldsymbol{O}(d_0 log\Delta)$ = $\boldsymbol{O}(nlogn)$	$\boldsymbol{O}(d_0 log\Delta w)$ = $\boldsymbol{O}(nlogn)$	$\boldsymbol{O}(nlogd_0)$ = $\boldsymbol{O}(nlogn)$	
M_1 and S_1				
M_{all}	$\boldsymbol{O}(n\Delta d_0 log\Delta)$ = $\boldsymbol{O}(n^3 logn)$	$\boldsymbol{O}(n\Delta d_0 log\Delta w)$ = $\boldsymbol{O}(n^3 logn)$	$\boldsymbol{O}(n^2\Delta logd_0)$ = $\boldsymbol{O}(n^3 logn)$	
S_{all}				
T	$\leq 2d_0+1$			

8. Заключение

В статье рассматривается модель распределённой системы с выделенным корнем, с которого начинается и в котором заканчивается работа алгоритмов. Предлагается 8 типов алгоритмов (по 4 для синхронного и асинхронного вариантов системы) для решения любой задачи на нумерованном неориентированном графе, основанные на сборе информации о всём графе в корне или в каждой вершине. Для трёх задач (MIS, FBS и MST) приводятся нижние оценки сложности, а алгоритмы оптимизируются по памяти. Кроме того, для задач, имеющих однозначное решение, зависящее только от простых циклов (в частности, FBS, MST, поиск гамильтонова цикла), предлагаются алгоритмы с минимальным временем работы в синхронном варианте модели. Для двух задач (FBS и MST) предложены алгоритмы, которые оптимальны по времени как в синхронном, так и в асинхронном вариантах модели.

Для синхронной некорневой модели, в частности, модели LOCAL существуют эффективные алгоритмы решения различных задач на графах, в том числе рандомизированные. Поэтому одним из возможных направлений дальнейших исследований предполагается изучение возможности симуляции этих алгоритмов в корневой модели, как синхронной, так и асинхронной.

Список литературы

[1]. Joseph JaJa. An Introduction to Parallel Algorithms, Addison-Wesley, 1992, ISBN 0-201-54856-9.

[2]. Stephen A.Cook. An overview of computational complexity. Communications of the ACM. 1983, June, Vol.26, No.6.

[3]. Luis Barba. LITERATURE REVIEW: Parallel algorithms for the maximal independent set problem in graphs. October 2012. web-site: http://cglab.ca/~lfbarba/parallel_algorithms/Literature_Review.pdf. (accessed: December 2016)

[4]. R.M. Karp, A. Wigderson. A fast parallel algorithm for the maximal independent set problem. Proc. 16th Annual ACM Symposium on Theory of Computing. ACM, New York, 1984, pp. 266–272.

[5]. M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. SIAM Journal on Computing. 1986, Vol.15, No 4., pp. 1036-1053. DOI:10.1137/0215074.

[6]. Noga Alon, Laszlo Babai, Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. Journal of Algorithms. 1986, December, Vol. 7, Issue 4, pp. 567-583.
[7]. David Peleg. Distributed computing — A Locality-sensitive approach. SIAM Monographs on Discrete Mathematics and Applications. 2000, 359 pp.
[8]. N.A. Lynch. Distributed Algorithms. The Morgan Kaufmann Series in Data Management Systems.1996, 904 pp.
[9]. Thomas Moscibroda. Locality, Scheduling, and Selfishness: Algorithmic Foundations of Highly Decentralized Networks. PhD thesis, ETH Zurich, 2006.
[10]. Y. Métivier, J. M. Robson, N. Saheb-Djahromi, A. Zemmar. An optimal bit complexity randomized distributed MIS algorithm. Distributed Computing. April 2011, Vol. 23, Issue 5, pp. 331–340.
[11]. Mohsen Ghaffari. An Improved Distributed Algorithm for Maximal Independent Set. Proc. of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). 2016, pp.270-277.
[12]. Mohsen Ghaffari. An Improved Distributed Algorithm for Maximal Independent Set. Cornell University Library: https://arxiv.org/pdf/1506.05093v2.pdf (accessed: December 2016)
[13]. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In Foundations of Computer Science (FOCS) 2012, IEEE, 2012, pp. 321–330. Also coRR abs/1202.1983v3.
[14]. Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC). ACM, 2004, pp 300–309.
[15]. И. Бурдонов, А. Косачев. Общий подход к решению задач на графах коллективом автоматов. Труды ИСП РАН, том 29, вып. 2, 2017, стр. 27-76.
[16]. И. Бурдонов, А. Косачев. Размер памяти для хранения упорядоченного корневого графа. Труды ИСП РАН, том 29, вып. 2, 2017, стр. 7-26.

Distributed algorithms on rooted undirected graphs

Igor Burdonov <igor@ispras.ru>
Alexander Kossatchev <kos@ispras.ru>
Alexander Sortov <sortov@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Distributed algorithms of solving problems on undirected graphs are considered. In section 2, a model is defined featuring a root as a starting and ending point of the algorithm execution. Synchronous and asynchronous versions of the model are described. In section 3, algorithms of solving any problems are suggested based on collecting information on the whole graph in the root or in any vertex, as well as, on the graph labeling (its vertices and/or edges), if required. Emphasis is made on the time of the algorithm execution or on saving memory in vertices and total size of transferred messages, if this time is minimal. The rest of the paper considers optimizations for particular problems: creation of Maximal Independent

Set (MIS), Finding Set of Bridges (FSB), creation of Minimum Spanning Tree (MST) in a edge-weighted graph. In section 4, a modification of general algorithms for these problems is suggested decreasing the estimate of memory size of vertices and messages. Section 5 includes lower-bound estimates of solution complexity for these problems. In section 6, for synchronous model, the time of algorithms execution with graph labeling is decreased to the lower bound for problems with single-valued solution depending on only simple cycles of the graph, in particular, FSB, MST and the problem of Hamiltonian cycle search. In section 7, time-optimal algorithms for FSB and MST are considered for both synchronous and asynchronous models. Conclusion summarizes the results and outlines the directions for further research.

Keywords: rooted undirected graph; distributed algorithms; graph problems; maximal independent set; minimum spanning tree; finding bridges.

DOI: 10.15514/ISPRAS-2017-29(5)-14

For citation: Burdonov I., Kossatchev A., Sortov A. Distributed algorithms on rooted undirected graphs. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 283-310 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-14

References

[1]. Joseph JaJa. An Introduction to Parallel Algorithms, Addison-Wesley, 1992, ISBN 0-201-54856-9.

[2]. Stephen A.Cook. An overview of computational complexity. Communications of the ACM. 1983, June, Vol.26, No.6.

[3]. Luis Barba. LITERATURE REVIEW: Parallel algorithms for the maximal independent set problem in graphs. October 2012. web-site: http://cglab.ca/~lfbarba/parallel_algorithms/Literature_Review.pdf. (accessed: December 2016)

[4]. R.M. Karp, A. Wigderson. A fast parallel algorithm for the maximal independent set problem. Proc. 16th Annual ACM Symposium on Theory of Computing. ACM, New York, 1984, pp. 266–272.

[5]. M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. SIAM Journal on Computing. 1986, Vol.15, No 4., pp. 1036-1053. DOI:10.1137/0215074.

[6]. Noga Alon, Laszlo Babai, Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. Journal of Algorithms. 1986, December, Vol. 7, Issue 4, pp. 567-583.

[7]. David Peleg. Distributed computing — A Locality-sensitive approach. SIAM Monographs on Discrete Mathematics and Applications. 2000, 359 pp.

[8]. N.A. Lynch. Distributed Algorithms. The Morgan Kaufmann Series in Data Management Systems.1996, 904 pp.

[9]. Thomas Moscibroda. Locality, Scheduling, and Selfishness: Algorithmic Foundations of Highly Decentralized Networks. PhD thesis, ETH Zurich, 2006.

[10]. Y. Métivier, J. M. Robson, N. Saheb-Djahromi, A. Zemmar. An optimal bit complexity randomized distributed MIS algorithm. Distributed Computing. April 2011, Vol. 23, Issue 5, pp. 331–340.

[11]. Mohsen Ghaffari. An Improved Distributed Algorithm for Maximal Independent Set. Proc. of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). 2016, pp.270-277.

[12]. Mohsen Ghaffari. An Improved Distributed Algorithm for Maximal Independent Set. Cornell University Library: https://arxiv.org/pdf/1506.05093v2.pdf (accessed: December 2016)

[13]. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In Foundations of Computer Science (FOCS) 2012, IEEE, 2012, pp. 321–330. Also coRR abs/1202.1983v3.

[14]. Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC). ACM, 2004, pp 300–309.

[15]. I. Burdonov, A. Kossatchev. A general approach to solving problems on graphs by collective automata. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 2, 2017, pp. 27-76 (in Russian).

[16]. I. Burdonov, A. Kosachev. Size of the memory for storage of ordered rooted graph. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 2, 2017, pp. 7-26 (in Russian).

Программное обеспечение для создания адаптивных сеток

А.Н. Семакин <arte-semaki@yandex.ru>
Московский Государственный Технический Университет им. Н. Э. Баумана
105005, Россия, Москва, 2-я Бауманская ул., д. 5, стр. 1

Аннотация. В этой статье мы представляем программный пакет для создания адаптивных конечно-разностных сеток через представление гидромеханических переменных разреженным рядом вейвлетов. Приводятся технические детали реализации. В частности, описаны используемые структуры данных и способ распараллеливания вычислений. Также представлены результаты решения некоторых физических задач с привлечением данного пакета.

Ключевые слова: программное обеспечение; вейвлеты; адаптивные сетки.

DOI: 10.15514/ISPRAS-2017-29(5)-15

Для цитирования: Семакин А.Н. Программное обеспечение для создания адаптивных сеток. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 311-328. DOI: 10.15514/ISPRAS-2017-29(5)-15

1. Введение

Практически все физические процессы одновременно протекают в широком диапазоне пространственных масштабов. Например, перенос примесей в атмосфере зависит как от крупномасштабной циркуляции (~10 тыс. километров), так и мелкомасштабной конвекции (~100 метров или меньше). Характер многофазных течений определяется не только факторами, действующими в масштабе все области (геометрия границ, гравитация и т.д.), но и в значительной степени зависит от межфазных взаимодействий на границе раздела фаз, где происходит резкое изменение материальных свойств среды.

Для достижения высокой точности в ходе моделирования численные методы должны разрешать все вовлеченные пространственные масштабы, что требует наличия огромного объема вычислительных ресурсов. В результате подобные расчеты достаточно сложно и дорого проводить на регулярной основе. Однако, довольно часто не требуется рассчитывать всю рассматриваемую область с одинаково высоким уровнем точности. Обычно в каждый отдельный момент времени малые пространственные масштабы проявляются на

нескольких относительно маленьких участках исходной расчетной области. В остальных частях исходной области физические переменные изменяются плавно и могут быть рассчитаны на грубой сетке без потери точности. Эта особенность была положена в основу целого ряда методов расчета на адаптивных сетках.

В этой статье мы представляем программный пакет создания адаптивных сеток с помощью одного из существующих подходов — теории вейвлетов. Эта адаптивная техника позволяет нам концентрировать точки сетки в областях, где гидродинамические переменные (скорость, плотность, давление, концентрация химических компонентов и т.д.) претерпевают резкие изменения, и разрежать сетку в областях, где эти переменные изменяются в незначительной степени. Сама теория вейвлетов предоставляет математический инструмент для выделения таких областей. Среди характерных особенностей нашего программного пакета можно выделить: эффективный алгоритм разложения гидродинамических переменных в ряд по вейвлетам с быстрым вычислением вейвлет-коэффициентов, быстрый алгоритм адаптации сетки на основе полученного ряда вейвлетов, возможность применения конечно-разностных схем высокого порядка точности (3 и выше), способность конструировать одно-, двух- и трехмерные сетки.

Статья организована следующим образом. В разделе 2 описаны однородные сетки и три вида адаптивных сеток, показаны преимущества адаптивного измельчения сеток. В разделе 3 перечислены существующие программные пакеты по построению адаптивных сеток. В разделе 4 приведены некоторые теоретические положения метода построения адаптивных сеток с помощью вейвлетов. В разделах 5 и 6 представлены использующиеся в нашем программном пакеты структуры данных и способ распараллеливания вычислений. В разделе 7 приведены некоторые примеры численных расчетов.

2. Однородные и адаптивные сетки

Любой конечно-разностный метод преобразует дифференциальное уравнение, описывающее тот или иной физический процесс, в конечно-разностное уравнение, определенное на некоторой разностной сетке. Подавляющее большинство численных методов использует однородные сетки с постоянным интервалом между узлами. Этот интервал обычно выбирается так, чтобы решение разностного уравнения было как можно ближе к решению исходного дифференциального уравнения.

Однородная сетка обеспечивает равномерное разрешение всей расчетной области. Если гидромеханические переменные имеют резкие градиенты в какой-то одной небольшой части расчетной области, и для их расчета требуется достаточно мелкая сетка, то мы должны использовать эту мелкую сетку во всей области, несмотря на то, что для получения решения с приемлемой точностью в остальной части области можно задействовать более

грубую сетку. Таким образом, использование однородной сетки приводит к неэффективному расходованию имеющихся вычислительных ресурсов.

Одним из возможных путей решения этой проблемы является переход от однородных сеток к адаптивным. Адаптивные сетки увеличивают плотность узлов в областях с резкими изменениями гидродинамических переменных и уменьшают там, где значения переменных меняются слабо. Кроме того, адаптивные сетки перестраиваются с течением времени вслед за конфигурацией потока. Эти свойства минимизируют требования к вычислительным системам, так как численные алгоритмы занимают не больше вычислительных ресурсов, чем им нужно для достижения заданной точности.

Существует несколько подходов к построению адаптивных сеток: криволинейные сетки, вложенные сетки и адаптивное измельчение сеток (АИС).

Криволинейные сетки строятся с помощью криволинейных систем координат, которые сгущают сетку вдоль одной или нескольких координатных направлений. В результате образуются хорошо разрешенные пространственные полосы, которые пересекают всю расчетную область в направлении, перпендикулярном соответствующей координате. Лишь небольшая часть каждой такой полосы (~20% или меньше) приходится на область резкого изменения гидромеханических переменных. Остальная часть (~80%) приходится на участки, подробное разрешение которых не требуется. Таким образом, хотя криволинейные сетки имеют существенно меньший размер по сравнению с однородными сетками, значительная доля узлов таких сеток используется крайне неэффективно.

Вложенные сетки обычно состоят из основной разреженной сетки, покрывающей всю область, и нескольких дополнительных сеток меньшего размера и с большей плотностью точек, которые встроены в основную сетку на участках, требующих более подробного разрешения, чем основная часть области. Использование дополнительных встраиваемых сеток конечного размера убирает избыточное разрешение значительных участков исходной области, возникающее при использовании криволинейных систем координат, и, следовательно, еще больше уменьшает размер сетки. Обычно на практике используют набор из основной крупной сетки и одной или двух последовательно вложенных друг в друга малых сеток.

Как криволинейные, так и вложенные сетки имеют фиксированный размер, который задается вручную в начальный момент времени и не меняется во время расчетов. Это означает, что данные сетки не способны реагировать на динамические изменения пространственных масштабов и требуют их предварительной оценки. Также для построения этих сеток требуется предварительная информация о возможном распределении значений гидромеханических переменных в расчетной области.

В отличие от описанных выше подходов к построению адаптивных сеток методы АИС способны динамически адаптировать сетку к любым особенностям течения без какой-либо предварительной информации. Методы АИС автоматически измельчают или огрубляют сетку в зависимости от текущего распределения значений гидромеханических переменных в расчетной области согласно заданному адаптационному критерию. Возможность оперативно изменять размер сетки позволяет методам АИС мгновенно реагировать на любые изменения в пространственных масштабах, поддерживая точность расчетов на заданном уровне.

3. Программные пакеты для АИС

В отличие от методов, работающих с сетками, структура и размеры которых известен до начала расчетов, АИС работает с динамически изменяющимися сетками, структура которых заранее неизвестна, что сильно усложняет задачу эффективного представления адаптивной сетки в памяти компьютера, а также последующей работы с различными ее частями.

Большая трудоемкость разработки программного кода под АИС привела к появлению и распространению ряда готовых программных продуктов для построения и управления адаптивными сетками. Такие пакеты обычно содержат систему переменных, функций и классов, написанных на одном из языков программирования высокого уровня. Их можно включить в состав собственного программного продукта и использовать, не разбирая подробно внутренние детали реализации, что существенно сокращает время написания собственных программ.

На данный момент в программных пакетах реализованы две стратегии измельчения сетки. Первая стратегия использует иерархию сеток, которые принадлежат разным уровням разрешения и покрывают последовательно уменьшающуюся часть исходной области (см. рис. 1а). В этом подходе измельчаются относительно большие блоки ячеек сетки, которые используются в качестве основных программных единиц при построении структур данных. Из-за регулярной формы сеток, входящих в иерархию, достаточно сложно эффективным образом охватить различные нерегулярные течения с теми или иными особенностями. В этом случае существенное количество точек будет приходиться на участки, не требующие подробного разрешения. Вследствие относительной простоты реализации подавляющая часть программных пакетов реализует именно этот подход построения адаптивных сеток: PARAMESH [1], AMRCLAW [2], AMROC [3], Enzo [4], HAMR [5], DAGH [6], AMR++ [7], SAMRAI [8], AMRCART [9], IAMR [10].

Вторая стратегия использует для измельчения отдельные ячейки сетки вместо их блоков. Каждая ячейка может быть поделена на несколько дочерних ячеек меньшего размера независимо от остальных (см. рис. 1б). В результате данная стратегия обладает большей гибкостью, но производит сетки нерегулярной

формы, что усложняет их программную реализацию. Из-за этого лишь очень небольшое число пакетов реализует данную стратегию (например, Gerris [11]).

На рис. 2 показаны примеры адаптивных сеток, построенных по обеим стратегиям. Адаптивные сетки имеют два уровня разрешения и покрывают некое облако химического компонента, занимающее лишь часть расчетной области. Сетка, основанная на блоках, состоит из двух блоков разного размера (рис. 2а). Большой блок разрешает всю расчетную область на первом уровне, а второй блок используется для разрешения облака на втором уровне. Хорошо видно, что часть ячеек второго блока лежит за пределами облака и, следовательно, используется нерационально. В свою очередь второй уровень адаптивной сетки, построенной путем измельчения ячеек, имеет нерегулярную структуру и содержит только те ячейки, которые разрешают само облако. Таким образом, данная сетка подробно разрешает только ту часть исходной области, которая нуждается в этом. В результате размер данной сетки существенно уменьшается по сравнению с блочной.

Наш программный пакет использует третью стратегию измельчения сетки — измельчение сетки на основе узлов (рис. 1в). Когда возникает необходимость, любой узел сетки на данном уровне разрешения может породить до 8 дочерних узлов в двухмерном случае и до 26 дочерних узлов в трехмерном случае на более высоком уровне разрешения. При измельчении ячеек мы получаем только 4 или 8 дочерних ячеек в двухмерном и трехмерном случаях, соответственно (сравните рис. 1б и 1в). В результате для обеспечения той же мелкости наша сетка требует меньшего числа уровней разрешения, что влечет за собой меньшее время доступа к каждому отдельному узлу сетки.

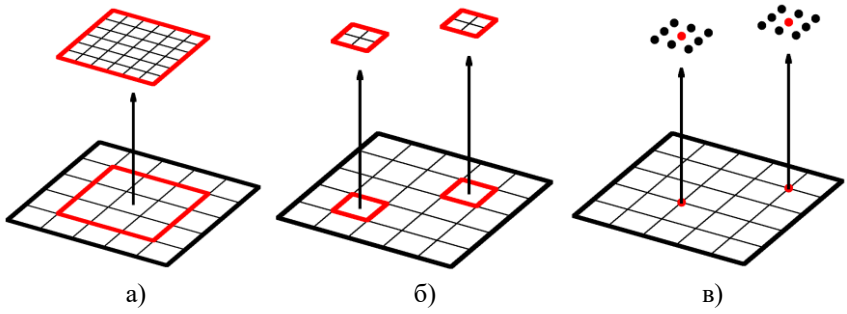
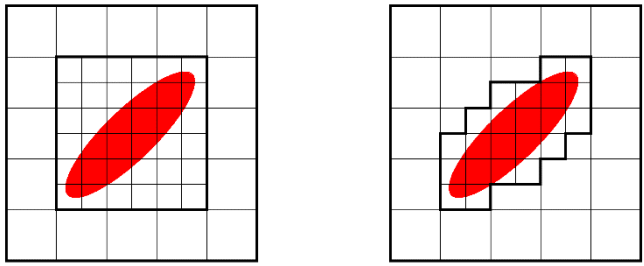


Рис. 1.Способы измельчения сетки: (а) на основе блоков, (б) на основе ячеек, (в) на основе узлов.

Fig. 1. Approaches to the grid refinement: (a) block-based grid, (b) cell-based grid, and (c) point-based grid.



а) б)

Рис. 2 Примеры адаптивных сеток с двумя уровнями разрешения: (а) на основе блоков, (б) на основе ячеек.

Fig. 2. Examples of the adaptive grids with two levels of resolution: (a) block-based grid and (b) cell-based grid.

4. Адаптация сетки с помощью вейвлетов

Рассмотрим реализованный в нашем пакете способ построения адаптивных сеток на основе теории вейвлетов.

В любой момент времени t_0 достаточно гладкая функция $f(\mathbf{x})$ может быть разложена в ряд по вейвлетам [12]:

$$f(\mathbf{x}) = \sum_{\mathbf{k}} b_{\mathbf{k}} \varphi_{\mathbf{k}}(\mathbf{x}) + \sum_{l=1}^{\infty} \sum_{\mathbf{k}} \mathbf{D}_{l,\mathbf{k}} \boldsymbol{\Psi}_{l,\mathbf{k}}(\mathbf{x}), \quad (1)$$

где l — уровень разрешения, $\mathbf{k}$ — положение в пространстве, $\varphi_{\mathbf{k}}(\mathbf{x})$ — масштабирующая функция, $\boldsymbol{\Psi}_{l,\mathbf{k}}(\mathbf{x}) = \{\psi^n_{l,\mathbf{k}}(\mathbf{x}), n = 1,...,\alpha\}$ — вектор вейвлетов и $\mathbf{D}_{l,\mathbf{k}} = \{d^n_{l,\mathbf{k}}, n = 1,...,\alpha\}$ — вектор коэффициентов вейвлетов. Значения α, $\mathbf{k}$ и $\mathbf{x}$ зависят от размерности пространства: для трехмерного пространства $\alpha = 7$, $\mathbf{k} = (i, j, k)$, $\mathbf{x} = (x, y, z)$, для двухмерного пространства $\alpha = 3$, $\mathbf{k} = (i, j)$, $\mathbf{x} = (x, y)$ и для одномерного пространства $\alpha = 1$, $\mathbf{k} = (i)$, $\mathbf{x} = (x)$. Для ряда (1) разработано множество различные семейств вейвлетов и масштабирующих функций. Мы используем вейвлеты семейства Deslauriers-Dubuc [13]. Выбор данного семейства обусловлен возможностью как построения самих вейвлетов, так и вычисления их коэффициентов в разложении (1) с помощью специальной достаточно простой интерполяционной схемы [14].

В ряде (1) масштабирующие функции $\varphi_{\mathbf{k}}(\mathbf{x})$ используются для представления основных крупномасштабных особенностей поведения функции $f(\mathbf{x})$. Вейвлеты $\psi^n_{l,\mathbf{k}}$ описывают поведение функции на мелких масштабах. Чем выше уровень разрешения l, тем меньше вклад вейвлетов в функцию $f(\mathbf{x})$.

Поскольку вклад вейвлета $\psi^n_{l,\mathbf{k}}$ в ряд (1) определяется величиной его коэффициента $d^n_{l,\mathbf{k}}$, мы можем сжать представление функции путем отсечения вейвлетов, чьи коэффициенты меньше некоторого порога ε. Порог ε выбирается достаточно малым, чтобы удалить из ряда (1) только те вейвлеты, которые несут лишь несущественную информацию о поведении функции. В результате получаем разреженный ряд вейвлетов:

$$f^{\varepsilon}(\mathbf{x}) = \sum_{\mathbf{k}} b_{\mathbf{k}} \varphi_{\mathbf{k}}(\mathbf{x}) + \sum_{l=1}^{J} \sum_{|d|>\varepsilon} \mathbf{D}_{l,\mathbf{k}} \boldsymbol{\Psi}_{l,\mathbf{k}}(\mathbf{x}), \quad (2)$$

где $f^{\varepsilon}(\mathbf{x})$ — это аппроксимация функции $f(\mathbf{x})$ с ошибкой, задаваемой порогом ε, J — конечный максимальный уровень разрешения, который зависит от величины ε. Разреженный ряд (2) содержит лишь небольшое число значимых вейвлетов.

Из способа построения вейвлетов семейства Deslauriers-Dubuc вытекает взаимно-однозначное соответствие между вейвлетами и точками физического пространства, т.е. каждый вейвлет ассоциируется с определенной точкой. Поэтому разреженный ряд (2) задает множество точек пространства, которые образуют сетку со сгущениями в областях быстрого изменения значений функции.

Множество значимых вейвлетов описывает только текущее поведение функции в данный момент времени. Однако физические процессы развиваются со временем. Это проявляется в постоянном появлении новых деталей в пространственном распределении функции f, которые отсутствуют в данный момент времени и, следовательно, не могут быть описаны значимыми вейвлетами. Чтобы учесть возможные изменения в поведении функции, в разреженный ряд (2) необходимо добавить дополнительные вейвлеты, чьи коэффициенты в данный момент времени меньше порога ε, но могут пересечь его в ближайшем будущем. Вместе значимые и дополнительные вейвлеты образуют множество активных вейвлетов, которые используются при построении адаптивной сетки.

5. Структура данных

Важнейшей деталью любого программного пакета, производящего адаптивные сетки, является структура данных, используемая для представления этой сетки. Выбор структуры данных задает порядок работы с данными, способ их хранения в памяти компьютера и алгоритм доступа. Фактически правильный выбор структуры данных определяет степень эффективности программного пакета.

Структура данных — это совокупность некоторых однотипных элементов, хранящихся в памяти компьютера, и набор операций для работы с этими

элементами. Выделяют два подхода к созданию структур данных — статический и динамический.

Статическая структура данных остается неизменной все время работы с ее элементами. Она имеет фиксированный размер и размещается в памяти компьютера в момент запуска программы. В результате данная структура способна предоставить быстрый доступ к любому своему элементу. Однако такие операции как добавление и удаление новых элементов требуют переформирования сразу всей структуры данных, что занимает достаточно большой объем времени работы CPU. Примером статических структур данных служит массив.

Динамическая структура данных — это структура, которая может изменять свой размер во время работы программы. Каждый ее элемент размещается в памяти только тогда, когда появляется новая порция данных. Все неактивные элементы своевременно удаляются из памяти. Поэтому динамическая структура данных содержит только актуальную информацию и занимает минимально возможный объем памяти. В отличие от статической структуры здесь операции добавления и удаления элементов выполняются предельно быстро, но динамическое размещение в памяти очень сильно замедляет доступ к каждому отдельному элементу. Примерами динамических структур данных являются деревья и списки.

Однородная сетка — это неизменное множество узлов, занимающих одни и те же позиции в пространстве все время расчетов. Размер и положение однородной сетки задаются заранее и не могут быть изменены. Единственная операция, которая выполняется с сеткой — это доступ к ее узлам. Поэтому для однородной сетки наиболее подходящей является статическая структура данных.

Адаптивная сетка — это множество узлов переменного размера. Такая сетка постоянно добавляет и удаляет узлы, следуя за эволюцией моделируемого физического процесса. Любой численный алгоритм, основанный на адаптивной сетке, использует все основные операции (прямой доступ, вставка и удаление) при работе с ее узлами. Также адаптивная сетка имеет меняющуюся со временем нерегулярную форму.

Если адаптивную сетку представлять статической структурой данных регулярной формы (например, массивом), то эта структура должна содержать избыточное число элементов, чтобы учесть нерегулярную форму адаптивной сетки и минимизировать число операций вставки/удаления. Необходимое число элементов намного больше (примерно на 3-4 порядка), чем предельное число узлов сетки, которые реально используются в расчетах. Поэтому статические структуры данных крайне неэффективны для описания адаптивных сеток, поскольку программа забирает существенно больший объем памяти, чем реально использует.

Более эффективная форма представления адаптивных сеток — это динамическая структура данных. Ее свойства полностью отвечают всем

требованиям эффективного представления адаптивной сетки. Динамическая структура данных может иметь нерегулярную форму, и каждый ее элемент может быть быстро добавлен или удален при необходимости. Эти свойства позволяют нам размещать в памяти столько элементов данных, сколько мы имеем узлов в сетке, и легко модифицировать саму сетку во время расчетов. Единственный недостаток такого представления — это медленный доступ к узлам.

Эффективная реализация адаптивных алгоритмов подразумевает следующие требования к динамической структуре данных: 1) быстрый доступ к любому элементу структуры, 2) быстрое удаление старых и добавление новых элементов в структуре, 3) линейная зависимость размера памяти, занимаемой структурой данных, от размера адаптивной сетки, 4) возможность эффективного распараллеливания вычислительного алгоритма.

Как было упомянуто выше, наиболее подходящим выбором для представления адаптивной сетки является динамическая структура данных. Любую адаптивную сетку можно представить в виде дерева или связного списка. Каждая из этих структур имеет свои преимущества и недостатки. Дерево предоставляет быстрый доступ к своим элементам, а связный список более подходит для организации параллельных вычислений.

В нашей работе мы представили адаптивную сетку в виде комбинации дерева и связного списка (см. рис. 3), что позволяет воспользоваться их преимуществами и взаимно нейтрализовать их недостатки. Так, мы используем дерево для быстрого доступа к необходимым узлам сетки и связный список для параллельных вычислений. В последнем случае связный список делится на несколько дочерних списков, которые распределяются между параллельными процессорами.

Дерево — это структура данных, составленная из элементов, содержащих указатели на другие элементы. В дереве, использующемся для представления одномерной сетки, каждый элемент имеет не более 2 указателей на элементы, представляющие узлы сетки на следующем уровне разрешения. В двухмерном и трехмерном случаях каждый элемент имеет до 8 или 26 указателей на дочерние элементы с более высокого уровня разрешения, соответственно. Положение дочерних элементов в трехмерном случае показано на рис. 4.

Связный список представляет собой группу элементов, выстроенных в некоторой последовательности. Каждый элемент содержит данные и ссылку на следующий элемент списка.

Для представления дерева и связного списка в одномерном случае мы определили производный тип данных type Point ... end type Point. Каждый элемент содержит индексы (i,l), левый и правый указатели на дочерние элементы в дереве и указатель на следующий элемент в связном списке. Индексы элемента определяют его место в дереве (l — уровень, на котором находится элемент в дереве, считая от корня, i — индекс позиции элемента на

уровне l) и положение в пространстве соответствующего узла сетки как $x = i / 2^l$. Левый указатель ссылается на ближайший узел сетки с уровня разрешения $l+1$, чья пространственная координата меньше, чем координата данного элемента. Правый указатель ссылается на ближайший узел сетки с уровня разрешения $l+1$, чья пространственная координата больше, чем координата данного элемента. Каждый элемент типа Point одновременно имеет указатели, необходимые для его включения как в дерево, так и в связный список, т.е. мы не дублируем элементы — обе структуры данных состоят из одних и те же элементов. На рис. 3 каждый узел сетки однозначно идентифицируется парой (i,l), которая определяет его положение как в дереве, так и в связном списке.

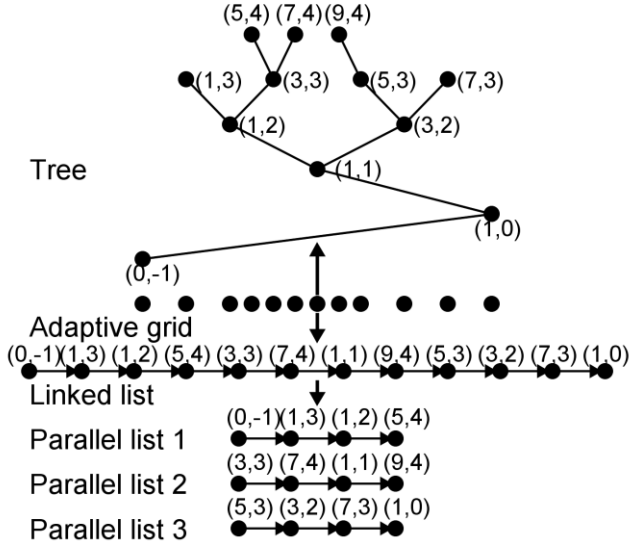


Рис. 3 Представление адаптивной сетки в виде дерева и связного списка.

Fig. 3. A representation of the adaptive grid in the form of a tree and a linked list.

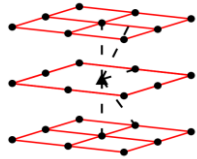


Рис. 4 Некоторый элемент (треугольник) со своими дочерними элементами (круги) в трехмерном дереве.

Fig. 4. A node (triangle) and its child nodes (circles) in the three-dimensional tree data structure

Двумерные и трехмерные сетки отличаются от одномерной одной и двумя дополнительными размерностями, соответственно. Здесь элементы характеризуются тремя или четырьмя индексами — (i, j, l) в двухмерном случае и (i, j, k, l) в трехмерном случае. Данные индексы определяются из пространственных координат соответствующих узлов сетки $x = i / 2^l$, $y = j / 2^l$ и $z = k / 2^l$. Также каждый элемент содержит 8 или 26 указателей на дочерние элементы на уровне $l+1$ в дереве и один указатель на следующий элемент в связном списке.

Для дискретизации пространственных производных наш пакет формирует конечно-разностные шаблоны до 8 порядка точности включительно. Для этого специальная процедура ищет элемент дерева, ближайший к данному, и принимает расстояние до него в пространстве за шаг шаблона. Далее процедура добавляет в шаблон все необходимые соседние элементы, расстояние до которых кратно шагу шаблона. В случае двух и трех измерений шаблоны формируются вдоль каждого направления. Заметим, что шаблоны в разных направлениях могут иметь разные шаги. В типе Point шаблон представлен массивом указателей на соответствующие элементы.

Для работы со структурами данных нам также нужны внешние ссылки на корень дерева и на первый элемент связного списка. Листинг программы для одномерного и трехмерного случаев приведен ниже:

1. Program listing for the 1D case in FORTRAN

```
type Child
        type(Point), pointer :: ref
end type Child
type Point
        integer :: i, l
        type(Child), allocatable, dimension(:) :: d_dx
        type(Child), dimension(1:2) :: links_ to_the_children_in_the_tree
        type(Point), pointer :: link_to_the_next_node_in_the_list
end type Point
type(Point), pointer :: head_of_the_linked_list
type(Point), pointer :: root_of_the tree
```

2. Program listing for the 3D case in FORTRAN

```
type Child
        type(Point), pointer :: ref
end type Child
type Point
        integer :: i, j, k, l
        type(Child), allocatable, dimension(:) :: d_dx,d_dy,d_dz
```

```
        type(Child), dimension(1:26) :: links_ to_the_children_in_the_tree
        type(Point), pointer :: link_to_the_next_node_in_the_list
    end type Point
    type(Point), pointer :: head_of_the_linked_list
    type(Point), pointer :: root_of_the tree
```

6. Параллельная реализация

Любой численный метод, использующий адаптивные сетки, выполняется в два шага: построение адаптивной сетки и численное решение модельных уравнений. На обоих шагах выполняется последовательность некоторых операций в каждой точке адаптивной сетки. Эти операции берут текущие значения гидромеханических переменных в ближайших точках сетки и дают новое значение для данной точки:

$$c_k^{new} = f(c_1^{old}, c_2^{old}, ..., c_n^{old}), k = 1, 2, ..., n, \quad (3)$$

где c_k^{new} и c_k^{old} — новое и текущее значения гидромеханической переменной в точке k. Так как результаты вычисления c_k^{new} в данной точке не зависят от аналогичных результатов в других точках, каждая операция может выполняться в параллельном режиме.

Адаптивная сетка как множество точек служит базисом для параллельной реализации численных методов. С помощью отдельной процедуры множество узлов сетки делится на подмножества одинакового размера, которые в дальнейшем могут быть переданы параллельным процессорам для обработки. Тогда последовательность операций, составляющая численный алгоритм, будет выполняться каждым параллельным процессором на выделенном ему подмножестве точек сетки. Например, сетка на рис. 5 состоит из 12 узлов. Эти узлы распределяются между 3 процессорами так, что каждый процессор имеет 4 узла. Далее процессоры выполняют последовательно операции 1 и 2 только на принадлежащих им точках, т.е. одновременно обрабатываются 3 узла. Для каждого узла время выполнения отдельной операции одинаково. Также узлы могут обрабатываться в любой последовательности. Поэтому распределение узлов между процессорами произвольно. Например, на рис. 5 узлы распределены по процессорам последовательно слева направо: первый процессор обрабатывает узлы 1–4, второму принадлежат узлы 5–8, а третьему достаются последние узлы 9–12. Если бы мы распределили узлы в другой последовательности, общее время работы не изменилось бы.

7. Примеры расчетов

Мы применили разработанный программный пакет к решению нескольких физических задач: движение в последовательно сужающемся и расширяющемся потоке (рис. 6), перенос облака примеси через Тихий океан из Китая в США (рис. 7) и подъем капли толуола в воде (рис. 8).

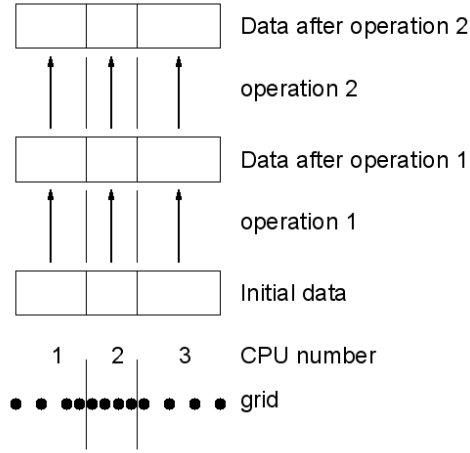


Рис. 5 Распределение точек сетки между процессорами.
для проведения параллельных вычислений

Fig. 5. Distribution of grid nodes among CPUs for the parallel implementation.

Движение в сужающемся-расширяющемся потоке и в атмосфере описывается одним уравнением адвекции массовой доли некоторого модельного вещества, которая использовалась в качестве входной переменной программного пакета для построения адаптивной сетки. Движение капли толуола в воде описывается полной системой уравнений гидромеханики (уравнение неразрывности, уравнение импульса и уравнение адвекции характеристической функции). Адаптивная сетка в этом случае строилась по двум переменным — характеристической функции и кинетической энергии.

Рис. 6–8 показывают, что адаптивная сетка, построенная с помощью разработанного программного пакета, подробно разрешает участки, где гидромеханические переменные претерпевают резкие изменения (внутри облака примеси и капли), и использует крупный шаг между узлами в остальной области. Адаптивная сетка имеет минимально возможный размер (на 3–5 порядков меньший, чем эквивалентная ей однородная сетка), при этом она достаточно велика, чтобы учитывать все возможные динамические изменения в потоке и не вносить каких-либо искажений. Например, численные эксперименты показали, что постоянная перестройка сетки в ходе решения приводит к крайне незначительной ошибке в сохранении массы, не превосходящей по ходу расчетов 0.02%.

Детальный анализ двухмерного движения облака примесей над Тихим океаном может быть найден в [15]. Анализ трехмерного движения облака примесей в атмосфере и движения капли толуола в воде будет опубликован в последующих статьях.

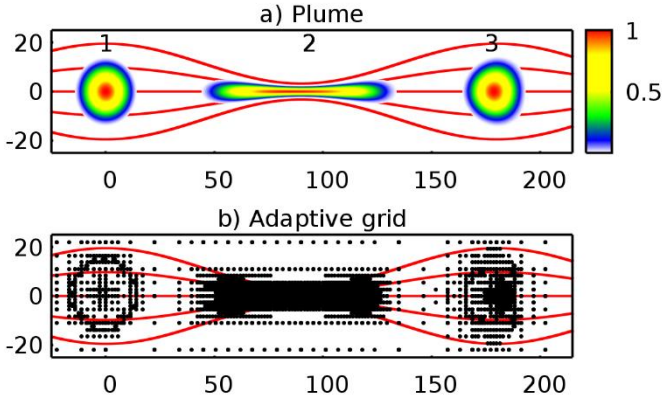


Рис. 6 Движение облака примесей (a) и эволюция соответствующей адаптивной сетки (b) в сужающемся-расширяющемся потоке для различных моментов времени (1), (2), (3).

Fig. 6. Evolution of (a) a plume and (b) the corresponding adaptive grid in the convergent-divergent flow at different time moments (1), (2), (3).

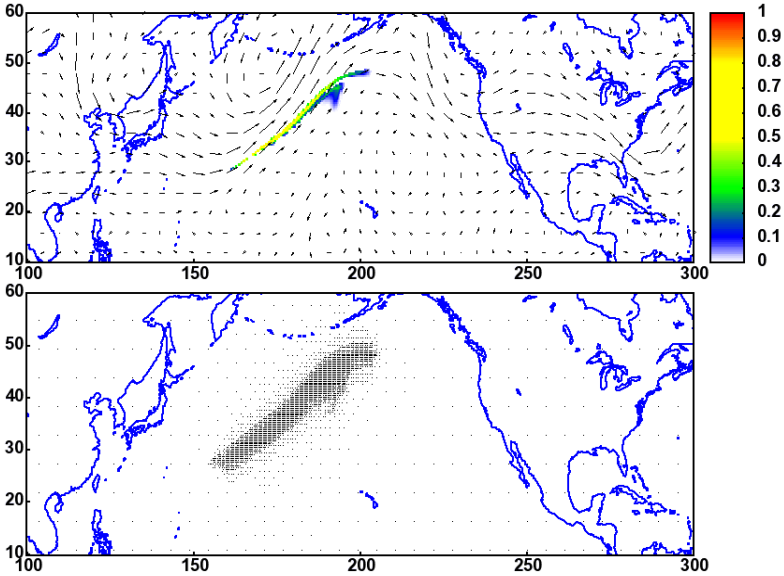


Рис. 7 Перенос облака примесей над Тихим океаном 1-ого мая 2001 года (верхняя панель) и соответствующая адаптивная сетка (нижняя панель).

Fig. 7. Pollution plume transport over the Pacific Ocean in the free troposphere on May 1, 2001 (the upper panel) and the corresponding adaptive grid (the lower panel).

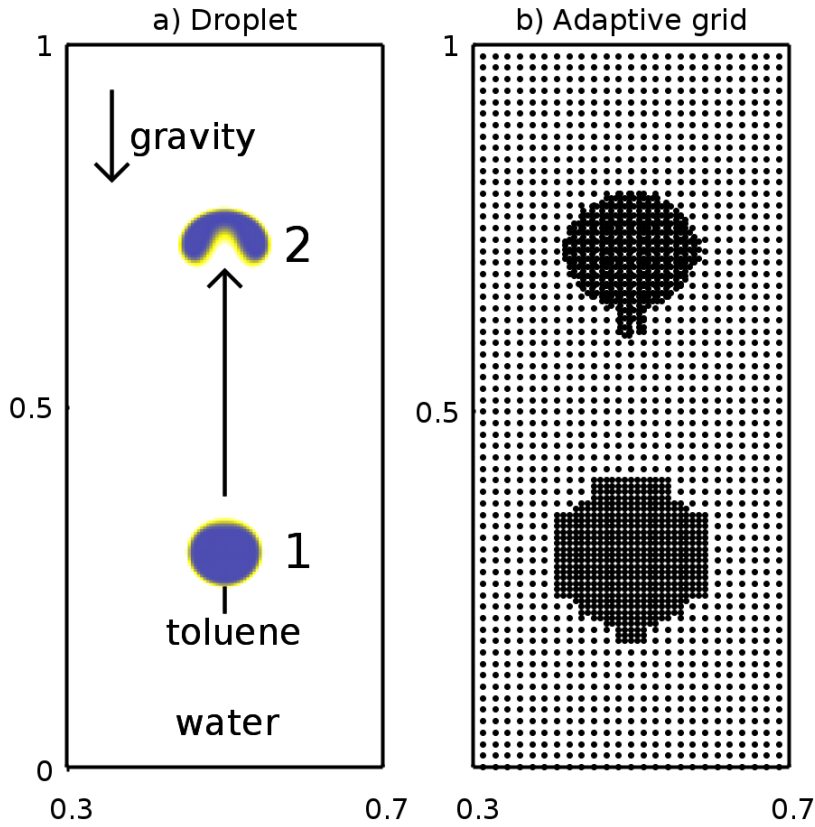


Рис. 8 Подъем капли толуола в воде. Здесь капля и соответствующая адаптивная сетка показаны для начального (1) и некоторого промежуточного (2) моментов времени.

Fig. 8. Evolution of a toluene droplet rising in water. Here the droplet and the corresponding adaptive grid are shown for the initial time moment (1) and some intermediate time moment (2) in the left and right panels, respectively.

8. Заключение

Мы разработали программный пакет для создания одно-, двух- и трехмерных адаптивных сеток. Адаптация основана на теории вейвлетов и осуществляется через разложение одной или более гидромеханических переменных в ряд вейвлетов. В компьютерной памяти после построения адаптивная сетка представляется в виде комбинации двух разных динамических структур

данных — дерева и связного списка. Совместное использование двух структур данных позволяет поддерживать соответствие между пространственной структурой сетки и ее программным представлением, при необходимости получать быстрый доступ к любой точке сетки и эффективно распараллеливать вычисления.

Программный пакет был применен к моделированию нескольких физических проблем. Было показано, что адаптивная сетка, построенная нашим пакетом, позволяет получать численное решение хорошего качества и при этом поддерживать размер сетки на минимально приемлемом уровне.

Список литературы

[1]. MacNeice P., Olson K. M., Mobarry C., deFainchtein R., Packer C. PARAMESH: A parallel adaptive mesh refinement community toolkit. Comput. Phys. Commun., 126, 2000, pp. 330–354.

[2]. Berger M. J., LeVeque R. J. Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems. SIAM J. Numer. Anal., 35, 1998, pp. 2298–2316.

[3]. Deiterding R., Radovitzky R., Mauch S. P., Noels L., Cummings J. C., Meiron D. I. A virtual test facility for the efficient simulation of solid material response under strong shock and detonation wave loading. Eng. Comput., 22, 2006, pp. 325–347.

[4]. Bryan G. L. et al. Enzo: An adaptive mesh refinement code for astrophysics. 2013, arXiv:1307.2265.

[5]. Neeman H. Autonomous hierarchical adaptive mesh refinement for multiscale simulations. Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1996.

[6]. Parashar M., Browne J. C. On partitioning dynamic adaptive grid hierarchies. Proceedings of the 29th Annual Hawaii International Conference on System Sciences, 1996.

[7]. Quinlan D. Adaptive mesh refinement for distributed parallel architectures. Ph.D. dissertation, University of Colorado at Denver., 1993.

[8]. Hornung R. D., Kohn S. R. Managing application complexity in the SAMRAI object-oriented framework. Concurr. Comp. Pract. E., 14, 2002, pp. 347–368.

[9]. Walder R., Folini D. A-MAZE: A code package to compute 3D magnetic flows, 3D NLTE radiative transfer, and synthetic spectra. Thermal and Ionization Aspects of Flows from Hot Stars: Observations and Theory, ASP Conference Series, 204, 2000, pp. 281–284.

[10]. Almgren A. S., Bell J. B., Colella P., Howell L. H., Welcome M. L. A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations. J. Comp. Phys., 142, 1998, pp. 1–46.

[11]. Popinet S., Rickard G. A tree-based solver for adaptive ocean modeling. Ocean Model., 16, 2007, pp. 224–249.

[12]. Holmstrom M. Solving hyperbolic PDEs using interpolating wavelets. SIAM J. Sci. Comput., 21, 1999, pp. 405–420.

[13]. Donoho D. L. Interpolating wavelet transforms. Department of Statistics. Stanford University, Tech. Rep., 1992.

[14]. Deslauriers G., Dubuc S. Symmetric iterative interpolation processes. Constr. Approx., 5, 1989, pp. 49–68.

[15]. Semakin A. N., Rastigejev Y. Numerical simulation of global-scale atmospheric chemical transport with high-order wavelet-based adaptive mesh refinement algorithm. Mon. Wea. Rev., 144, 2016, pp. 1469–1486.

Software for adaptive grid construction

A.N. Semakin <arte-semaki@yandex.ru>
Bauman Moscow State Technical University,
5, Second Bauman st., Moscow, 105005, Russia.

Abstract. In this paper, we present a software package for the construction of an adaptive finite-difference grid by expanding physical variables into a sparse wavelet series. Some technical details of the program implementation are given. In particular, we describe data structures used for the grid representation in computer memory and tools for organizing parallel calculations on the adaptive grid. Also the results of several numerical simulations involving the developed package are shown.

Keywords: software; wavelets; adaptive grids.

DOI: 10.15514/ISPRAS-2017-29(5)-15

For citation: Semakin A.N. Software for adaptive grid construction. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 311-328 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-15

References

[1]. MacNeice P., Olson K. M., Mobarry C., deFainchtein R., Packer C. PARAMESH: A parallel adaptive mesh refinement community toolkit. Comput. Phys. Commun., 126, 2000, pp. 330–354.
[2]. Berger M. J., LeVeque R. J. Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems. SIAM J. Numer. Anal., 35, 1998, pp. 2298–2316.
[3]. Deiterding R., Radovitzky R., Mauch S. P., Noels L., Cummings J. C., Meiron D. I. A virtual test facility for the efficient simulation of solid material response under strong shock and detonation wave loading. Eng. Comput., 22, 2006, pp. 325–347.
[4]. Bryan G. L. et al. Enzo: An adaptive mesh refinement code for astrophysics. 2013, arXiv:1307.2265.
[5]. Neeman H. Autonomous hierarchical adaptive mesh refinement for multiscale simulations. Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1996.
[6]. Parashar M., Browne J. C. On partitioning dynamic adaptive grid hierarchies. Proceedings of the 29th Annual Hawaii International Conference on System Sciences, 1996.
[7]. Quinlan D. Adaptive mesh refinement for distributed parallel architectures. Ph.D. dissertation, University of Colorado at Denver., 1993.

[8]. Hornung R. D., Kohn S. R. Managing application complexity in the SAMRAI object-oriented framework. Concurr. Comp. Pract. E., 14, 2002, pp. 347–368.
[9]. Walder R., Folini D. A-MAZE: A code package to compute 3D magnetic flows, 3D NLTE radiative transfer, and synthetic spectra. Thermal and Ionization Aspects of Flows from Hot Stars: Observations and Theory, ASP Conference Series, 204, 2000, pp. 281–284.
[10]. Almgren A. S., Bell J. B., Colella P., Howell L. H., Welcome M. L. A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations. J. Comp. Phys., 142, 1998, pp. 1–46.
[11]. Popinet S., Rickard G. A tree-based solver for adaptive ocean modeling. Ocean Model., 16, 2007, pp. 224–249.
[12]. Holmstrom M. Solving hyperbolic PDEs using interpolating wavelets. SIAM J. Sci. Comput., 21, 1999, pp. 405–420.
[13]. Donoho D. L. Interpolating wavelet transforms. Department of Statistics. Stanford University, Tech. Rep., 1992.
[14]. Deslauriers G., Dubuc S. Symmetric iterative interpolation processes. Constr. Approx., 5, 1989, pp. 49–68.
[15]. Semakin A. N., Rastigejev Y. Numerical simulation of global-scale atmospheric chemical transport with high-order wavelet-based adaptive mesh refinement algorithm. Mon. Wea. Rev., 144, 2016, pp. 1469–1486.

Численное исследование теплоотдачи в каналах с неглубокими подковообразными лунками

[1] А.А. Цынаева <a.tsinaeva@rambler.ru>
[2] С.Е. Разоренов <razserg@list.ru>
[1] В.В. Белая <bonyparkery@gmail.com>
[1] ФГБОУ ВО СамГТУ,
443010, Россия, г. Самара, ул. Молодогвардейская, дом 244
[2] АО Транснефть-Приволга, Самарское РНУ,
443020, Россия, г. Самара, ул. Ленинская, дом 100

Аннотация. Работа посвящена численному исследованию теплоотдачи в прямоугольных каналах с односторонним нанесением неглубоких лунок. Математическое моделирование выполнено на базе уравнения Навье-Стокса, уравнения неразрывности, уравнения энергии. Для замыкания системы уравнений использована k-w-sst модель турбулентности. Численное решение получено с помощью программного пакета Code Saturne, распространяемого на основе свободной лицензии. Для построения сетки использовался программный пакет Salome. В работе рассмотрены вопросы верификации получаемого численного решения. На базе выполненного численного исследования проведена оценка теплогидравлической эффективности неглубоких лунок различной конфигурации для прямоугольных каналов с односторонним нанесением неглубоких лунок.

Ключевые слова: свободное программное обеспечение, численное моделирование, аэродинамика, теплообмен, лунки.

DOI: 10.15514/ISPRAS-2017-29(5)-16

Для цитирования: Цынаева А.А., Разоренов С.Е., Белая В.В. Численное исследование теплоотдачи в каналах с неглубокими подковообразными лунками. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 329-344. DOI: 10.15514/ISPRAS-2017-29(5)-16

1. Введение

При разработке теплообменных аппаратов на основе перспективных методов интенсификации теплообмена [1] численное моделирование является способом снижения временных и материальных затрат на проектирование. В настоящее время для численного решения таких научно-технических задач

широко применяются программные комплексы OpenFoam и Code_Saturne, распространяемые на базе свободной лицензии [2,3].

Следует отметить, что программные комплексы со свободной лицензией [2,3] имеют открытый код, и могут быть как с графическим интерфейсом, так и без него. При этом каждый пользователь подбирает такой программный комплекс, который соответствует его запросам и возможностям.

Основной задачей создания перспективных теплообменных аппаратов является разработка таких методов интенсификации теплообмена, для которых рост гидравлического сопротивления будет не столь значительным [1,4,5]. К таким методам следует отнести использование выемок различной конфигурации: сферические, траншейные, двухполостные, гантелеобразные лунки и т.д. [4-7]. Для сравнения параметров выемок обычно используется соотношение коэффициента теплоотдачи к коэффициенту гидравлического трения, определяющее теплогидравлическую эффективность поверхности теплообмена. Использование математического моделирования позволяет снизить число экспериментальных испытаний и оценить эффективность предлагаемых решений на стадии проектирования. При этом разработка новых решений может осуществляться не только с привлечением программных комплексов на базе свободной лицензии [2,3,8], но и с использованием облачных сервисов [9].

Программный комплекс Code_Saturne [2], выбранный авторами в качестве инструмента исследования, изначально разрабатывался с целью решения задач гидродинамики и теплообмена для энергетического оборудования атомных станций [2]. При этом, имеющийся в нем математический аппарат, возможно использовать и при исследовании систем обеспечения микроклимата и их элементов [10], при исследовании естественной конвекции [11], при разработке перспективных теплообменных аппаратов [12,13]. Кроме того, согласно документации, этот программный продукт может использоваться для численного исследования атмосферных течений, электромагнитных явлений и двухфазных потоков [2].

2. Формулировка задачи, расчетные сетки

Работа посвящена численному исследованию течения и теплообмена в прямоугольном канале с двумерными подковообразными лунками относительной глубиной $h/D<0.15$, где h – глубина лунки, D – диаметр лунки. Такие лунки считаются неглубокими [4]. В качестве рабочего тела задан воздух.

2.1 Математическая модель

Математическое моделирование течения и теплообмена выполнено в программном комплексе Code_Saturne с помощью RANS подхода. Математическая модель включала в себя уравнение сохранения массы

(уравнение неразрывности), уравнение сохранения количества движения (Навье-Стокса), уравнение энергии, уравнение состояния. Проекции на координатные оси *x*, *y*, *z* уравнения движения в форме Навье-Стокса:

$$\begin{cases} \rho\dfrac{du}{dt} = g_x - \dfrac{\partial P}{\partial x} + \dfrac{\partial}{\partial x}\left[\mu\left(2\dfrac{\partial u}{\partial x} - \dfrac{2}{3}\operatorname{div}\vec{V}\right)\right] + \dfrac{\partial}{\partial y}\left[\mu\left(\dfrac{\partial u}{\partial y} + \dfrac{\partial v}{\partial x}\right)\right] + \dfrac{\partial}{\partial z}\left[\mu\left(\dfrac{\partial u}{\partial z} + \dfrac{\partial w}{\partial x}\right)\right]; \\ \rho\dfrac{dv}{dt} = g_y - \dfrac{\partial P}{\partial y} + \dfrac{\partial}{\partial x}\left[\mu\left(\dfrac{\partial u}{\partial y} + \dfrac{\partial v}{\partial x}\right)\right] + \dfrac{\partial}{\partial y}\left[\mu\left(2\dfrac{\partial v}{\partial y} - \dfrac{2}{3}\operatorname{div}\vec{V}\right)\right] + \dfrac{\partial}{\partial z}\left[\mu\left(\dfrac{\partial v}{\partial z} + \dfrac{\partial w}{\partial y}\right)\right]; \\ \rho\dfrac{dw}{dt} = g_z - \dfrac{\partial P}{\partial x} + \dfrac{\partial}{\partial x}\left[\mu\left(\dfrac{\partial u}{\partial z} + \dfrac{\partial w}{\partial x}\right)\right] + \dfrac{\partial}{\partial y}\left[\mu\left(\dfrac{\partial v}{\partial z} + \dfrac{\partial w}{\partial y}\right)\right] + \dfrac{\partial}{\partial z}\left[\mu\left(2\dfrac{\partial w}{\partial z} - \dfrac{2}{3}\operatorname{div}\vec{V}\right)\right]. \end{cases} \quad (1)$$

где ρ – плотность; u, v, w – проекции вектора скорости потока на оси *x*, *y*, *z*; *t* – время; g_x, g_y, g_z – проекции вектора внешней массовой силы на координатные оси; P – давление; μ – коэффициент динамической вязкости;

Уравнение неразрывности в виду отсутствия при решении поставленной задачи источников массы принимает вид:

$$\frac{d\rho}{dt} + \operatorname{div}\left(\rho\vec{V}\right) = 0 \quad , \quad (2)$$

Уравнение энергии:

$$\rho\frac{de}{dt} = -\operatorname{div}\left(\underline{q''}\right) + q''' - P\operatorname{div}\left(\vec{V}\right) + S, \quad , \quad (3)$$

где e – внутренняя энергия; $\underline{q''}$ – вектор потока теплоты, равный $\underline{q''} = -\lambda\underline{\nabla T}$ [2,14]; $q''' = 0$ – тепловая мощность внутренних источников теплоты; λ – коэффициент теплопроводности; T – температура.

Диссипативная функция S определяется выражением [2,14]:

$$S = \mu\left\{2\left[\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial z}\right)^2\right] + \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right)^2 + \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)^2 - \right.$$
$$\left. - \frac{2}{3}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)^2\right\}, \quad (4)$$

Так, как в качестве рабочего тела рассматривался воздух, то уравнение состояния принимало вид уравнения Менделеева-Клайперона [14]:

$$\frac{P}{\rho} = \frac{R_m}{m_g} \cdot T, \quad (5)$$

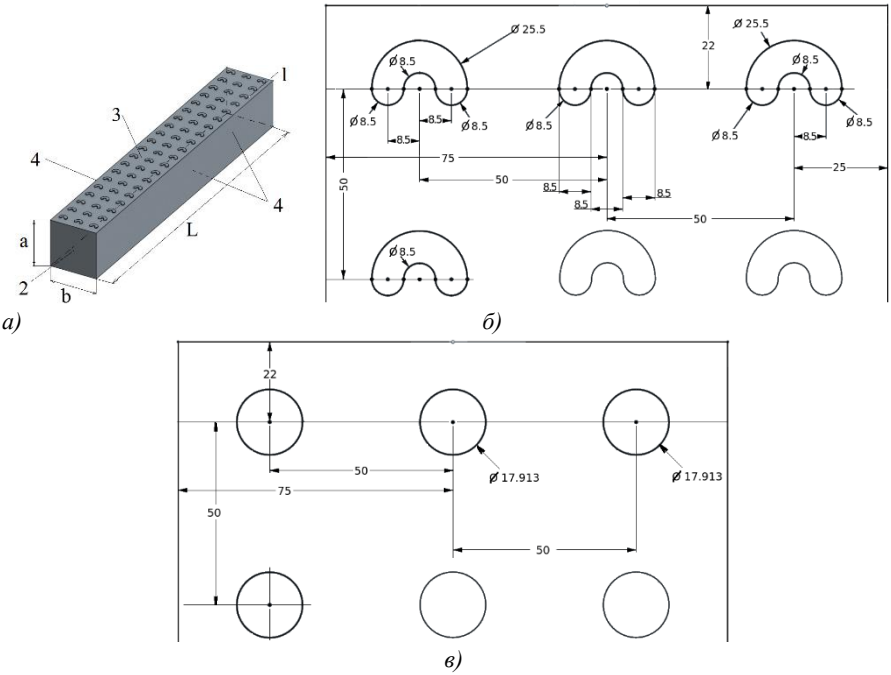
где R_m – универсальная газовая постоянная; m_g – молекулярная масса газа.

В RANS подходе для замыкания дифференциальных уравнений была использована *k-ω sst* модель турбулентности [15], выбор которой обусловлен опытом численных исследований авторов, а также по результатам анализа работ [6,10,11,12,13,16].

Численное решение получено методом конечных объемов, в рамках которого уравнения интегрируются по каждому контрольному объему Ω_i сетки [2].

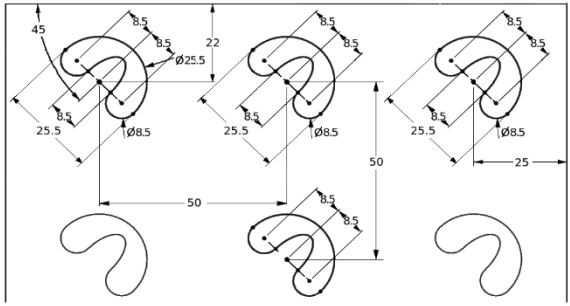
2.2 Условия моделирования и характерные масштабы задачи

Численное исследование теплообмена и гидравлического сопротивления в канале подковообразными лунками выполнено для прямоугольного канала (рис. 1) сечением a×b=150×150 мм и длиной L=1000 мм. Одна из стенок канала имеет подковообразные лунки с относительной глубиной h/D=2/17,913=0,111<0.15, площадь пятна лунки составляет 252.02 мм^2. Расположение лунок на стенке – коридорное, в три ряда, продольный и поперечный шаг между лунками составляет 50 мм. Количество лунок в одном ряду по длине канала составляет 20 шт, на стенке лунки выполнены в три ряда. На рис. 1 также представлены геометрические характеристики лунок и их размещение на поверхности канала относительно друг друга.



а) *б)*

в)



г)

Рис. 1. Конфигурация канала и выемок на его поверхности канала: (а) общий вид расчетной области: 1 – вход; 2 – выход; 3 – обогреваемая стенка канала с неглубокими лунками; 4 – необогреваемые стенки канала без лунок; (б) простые подковообразные 2d лунки; (в) цилиндрические лунки; (г) подковообразные наклоненные к потоку на 45 градусов 2d лунки

Fig. 1. Configuration of the channel and notches on its channel surface: (a) general view of the calculation area: 1 - input; 2 - output; 3 - heated channel wall with shallow wells; 4 - unheated canal walls without holes; (b) simple horseshoe-shaped 2d wells; (c) cylindrical wells; (d) horseshoe-shaped inclined to the flow at 45 degrees 2d wells

Для анализа эффективности разработанной конструкции подковообразной лунки проводилось сравнение параметров с показателями теплообмена и трения для канала с цилиндрическими лунками той же площади пятна и глубины.

Описание условий моделирования, в том числе граничных условий: на входе в канал задавалась скорость потока ($\underline{\bar{u}}_{00} = 1; 5; 10$ м/с), направленная по нормали, на выходе из канала заданы условия свободного истечения в окружающую среду, стенки канала полагались гладкими, на обогреваемой стенке с лунками задавались условия второго рода, тепловой поток к обогреваемой стенке (поз. 3, рис. 1, *а*) $q_w = 500$ Вт/м^2, на необогреваемых стенках (поз. 4, рис. 1, *а*) $q_w = 0$. Расчет проведен при критерии Рейнольдса $\mathrm{Re}_l = 11278$, $\mathrm{Re}_l = 56390$, $\mathrm{Re}_l = 112782$, l – гидравлический диаметр прямоугольного канала $l = 2ab/(a+b)$, В качестве рабочего тела принят воздух, коэффициент кинематической вязкости $\upsilon = 13{,}28 \cdot 10^{-6}$ м^2/с; $c_P = 1005$ Дж/кг, ускорение свободного падения $\underline{g} = 9{,}81$ м/с^2. Так как по предварительной оценке произведение критериев $Gr \cdot \mathrm{Pr} > 10^6$, то влияние внешних массовых сил необходимо учитывать [17]. Эта оценка была в дальнейшем подтверждена в результате численного исследования. Плотность потока определяется температурой в соответствии с уравнением (5). Задание граничных условий осуществлялось в программном комплексе Code_Saturne.

2.3 Дискретизация расчетной области

Построение сетки было выполнено с помощью открытой интегрируемой платформы Salome [8]. Для создания 3-d элементов использовался алгоритм Tetrahedron (3D), параметры определялись по Netgen 3D Parameters: Netgen 1D-2D с минимальным размером 0,002, максимальным размером 0,005. Кроме того, вблизи стенки с неглубокими лунками выполнялось дополнительное сгущение сетки, на поверхности стенки сетка выполнялась из треугольных элементов [6]. На рис. 2 представлен анализ зависимости времени дискретизации t_{gen} расчетной области и времени получения решения t_{num} от количества ячеек N_{el}.

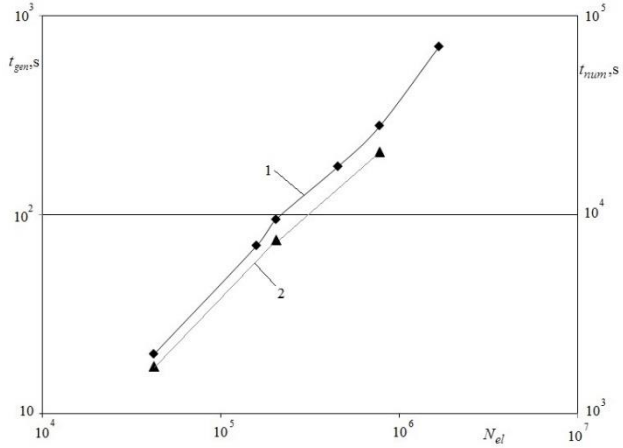


Рис. 2. Характеристика дискретизации расчетной области: 1 – затраты времени на дискретизацию расчетной области; 2 – затраты времени на получение численного решения

Fig. 2. Characterization of the discretization of the computational domain: 1 - the time spent on the sampling area; 2 - the time required to obtain a numerical solution

Размеры ячеек определялись из соотношения времени дискретизации и времени получения численного решения при сохранении приемлемой точности расчета [6], что основано на имеющемся у авторов опыте численного моделирования задач подобного типа (см. работу [6]). Исследование сеточной сходимости при моделировании течения в прямоугольном канале с неглубокими лунками по одной из стенок выполнено одним из авторов (Цынаевой А.А.) в работе [6]. В результате исследования сеточной сходимости в работе [6] выявлено, что при $N_{el} = 700000$ дальнейшее увеличение количества расчетных ячеек сетки не влияет на качество получаемого решения. При количестве ячеек $N_{el} < 500000$ наблюдаются

флуктуации скорости потока в пристеночной области [6]. Результаты этого исследования докладывались на конференции OpenClouds 2015, организованной ИСП РАН (г. Москва, дек. 2015 г.). В этой связи, в данной работе представлены результаты моделирования для прямоугольного канала с неглубокими лунками для сетки с $N_{el} = 776904$ объемных элементов. Удельное количество ячеек на длину расчетной области составило $n_{el} = \mathrm{N}_{el}/\mathrm{L} = 776904/1 = 776904$.

Граничные условия задавались с помощью инструментария программного пакета Code Saturne [2,6] и описаны ранее. В работе использовался графический интерфейс программного пакета Code Saturne [2,6], вычисления производились на компьютере со следующими характеристиками: Intel Core i5, четыре ядра, 4 Гб, 2,6 ГГц. Для разрешения поля давления использовалась схема Multigrid, также, как в работе [6], для поля температуры и для скорости, для кинетической энергии турбулентных пульсаций и для ω, применены автоматические настройки, максимальное количество итераций по каждому циклу задано равным 10000, точность решателя определялась величиной равной 10^{-8}. Схема Multigrid базируется на применении последовательности уменьшающихся сеток, а также операторов, позволяющих осуществлять переход от одной сетки к другой. Численное решение получено итерационным методом, шаг по времени определялся в зависимости от скорости течения, продолжительность численного эксперимента определялась временем трехкратного прохождения рабочего тела в канале, максимальное значение числа Куранта принималось равным пяти. Максимальный шаг по времени был определен из следующего выражения:

$$\Delta\tau_{\max} = \frac{\Delta l \cdot Cr}{\underline{\bar{u}}_{00}}, \qquad (11)$$

где Δl – линейный размер ячейки расчетной сетки; $\underline{\bar{u}}_{00}$ – скорость потока на входе в канал; Cr – число Куранта. Максимальное значение числа Куранта ограничено величиной $Cr = 5$. Это определяется с тем, что для связанного решения уравнений баланса импульса и неразрывности был применен алгоритм SIMPLEC [2].

3. Проверка адекватности

Проверка адекватности численного решения выполнялась путем сравнения результатов математического моделирования с экспериментальными исследованиями.

Во-первых, проводилось сравнение рассчитанных и экспериментально определенных аэродинамических характеристик для течения в канале с неглубокими гантелеобразными лунками. Эксперименты проводились на базе аэродинамической трубы кафедры «Теплогазоснабжения» ФГБОУ ВО СамГТУ. Результаты сравнения экспериментальных и модельных

аэродинамических характеристик опубликованы авторами в работах [18,19]. При сравнении аэродинамических характеристик потока расхождение расчетных и экспериментальных значений находилось в пределах погрешности эксперимента [18,19].

Для проверки качества решения при наличии теплообмена проведено сравнение рассчитанных и экспериментально определенных характеристик теплообмена в узком прямоугольном канале с неглубокими цилиндрическими лунками при различных параметрах течения. В этом случае, были использованы эксперименты КНИТУ им. Туполева-КАИ (г. Казань) [20].

Эксперимент в КНИТУ им. Туполева-КАИ (г. Казань) [20] был выполнен для щелевого канала с сечением a×b=2×96 мм и длиной L=196 мм, лунки цилиндрические с острой кромкой с h=1,6 мм, D=16 мм, h/D=0.1, расположены выемки в шахматном порядке с шагом 8 мм, рабочее тело – воздух, $\mathrm{Re}_l = 200...20000$. В работе [20] также представлен результат обобщения по критериальной зависимости экспериментальных данных КНИТУ им. Туполева-КАИ по теплоотдаче для каналов с цилиндрическими выемками при $\mathrm{Re}_l = 2500...20000$:

$$Nu = 0.0164 \cdot \mathrm{Re}_l^{0.85} \cdot (h/D)^{0.07}, \quad (12)$$

Nu – критерий Нуссельта, $Nu = \alpha' \cdot l/\lambda$; α' – коэффициент теплоотдачи; l – характерный размер, м; λ – коэффициент теплопроводности рабочего тела.

Так как эти результаты не были авторами опубликованы ранее, то сравнение расчетных и экспериментальных характеристик теплообмена представлено на рис. 3. Анализируя, представленные на рис. 3 результаты, следует отметить, что наблюдается достаточно качественное соотношение результатов расчета с экспериментальными данными, но имеются и некоторые расхождения.

При $\mathrm{Re}_l = 3000$ расхождение не превышает 10 процентов, далее расхождение увеличивается, но при $\mathrm{Re}_l = 12000$ вновь снижается. Такое рассогласование данных связано, по-видимому с тем, что при $2300 < \mathrm{Re}_l < 10000$ режим течения является переходным. Однако расхождение не превышает 25 процентов. В этой связи, численное исследование теплообмена в канале с неглубокими подковообразными лунками ограничено результатами для развитого турбулентного режима течения при $\mathrm{Re}_l >> 10000$, как наиболее распространенного, по мнению авторов, для теплообменных аппаратов.

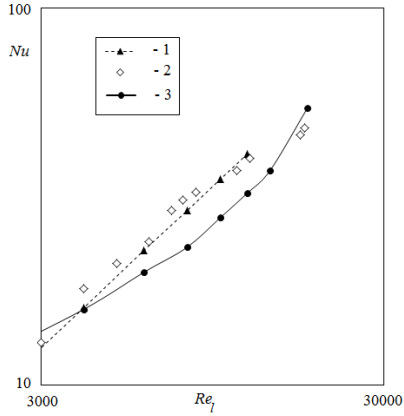


Рис. 3. Данные по теплоотдаче с учетом развития поверхности при наличии выемок: 1 – результаты расчета (12) по критериальной зависимости [20]; 2 – эксперимент КНИТУ им. Туполева-КАИ [20]; 3 – численный расчет авторов в Code Saturne
Fig. 3. Heat transfer data taking into account surface development in the presence of depressions: 1 - calculation results (12) according to the criterial dependence [20]; 2 - experiment Knitov. Tupolev-KAI [20]; 3 - numerical calculation of authors in Code Saturne

4. Результаты численного моделирования и их обсуждение

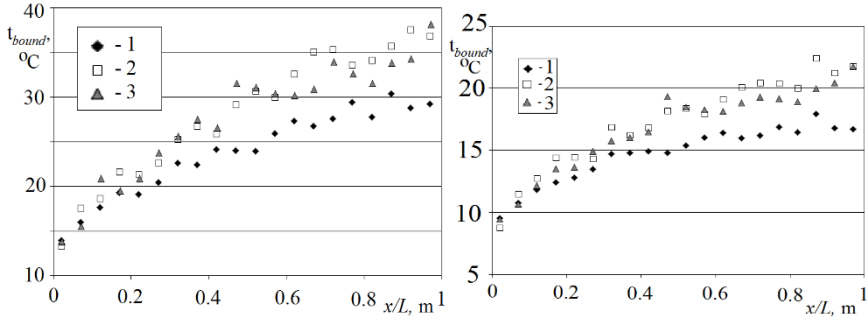
Численное моделирование выполнено для трех типов расчетной области: прямоугольного канала с цилиндрическими лунками, канала с подковообразными лунками [19], канала с подковообразными лунками, повернутыми к потоку на 45 градусов (см. рис. 1).

Целью численного моделирования являлся анализ течения и теплообмена в прямоугольном канале с неглубокими подковообразными 2d лунками для определения теплогидравлической эффективности каналов с такими лунками. Для корректного сравнения теплогидравлической эффективности поверхностей с различными лунками были выполнены следующие условия: одинаковая глубина (h/D=0.11, h=2 мм) лунок, равные площади «пятна» исследуемых лунок (S_p=252.02 мм^2), одинаковый тип расположения и число рядов лунок (n=3).

Далее показаны результаты численного исследования течения и теплообмена в канале с неглубокими лунками (рис. 1). На рис. 4 представлены значения температуры стенки канала для центрального ряда лунок, осредненные по площади лунки:

$$t_{bound} = \frac{\sum t_i \cdot F_i}{\sum F_i}, \qquad (13)$$

где t_{bound} – средняя температура стенки для лунки; F_i – площадь поверхности ячейки; t_i – температура стенки в i-той ячейке сетки; $\sum F_i = S_p$ – площадь пятна лунки.



а) *б)*

Рис. 4. Температура дна лунки (а) для Re_l=56391, (б) для Re_l=112782:
1 – подковообразная лунка; 2 – цилиндрическая лунка; 3 – подковообразная лунка, повернутая относительно потока на 45 градусов
Fig. 4. The bottom of the well temperature (a) for Rel = 56391, (b) for Rel = 112782:
1 - horseshoe hole; 2 - a cylindrical hole; 3 - horseshoe hole, rotated relative to the flow by 45 degrees

Как видно из рис. 4, конструкция лунки определяет температуру поверхности лунки.

Величина локального коэффициента теплоотдачи определялась по выражению:

$$\alpha' = q/(t_{bound} - t_f), \quad (13)$$

где q – удельный тепловой поток от стенки, Вт/м^2; t_f – температура потока, определяемая на оси потока над лункой.

Значения локального коэффициента теплоотдачи для центрального ряда лунок показаны на рис. 5. Анализируя рис. 5, следует отметить, что наибольшие значения коэффициента теплоотдачи характерны для канала с подковообразными лунками (рис. 1, б). Для повернутых на 45 градусов относительно потока подковообразных лунок и для цилиндрических лунок коэффициенты теплоотдачи по величине оказываются близкими.

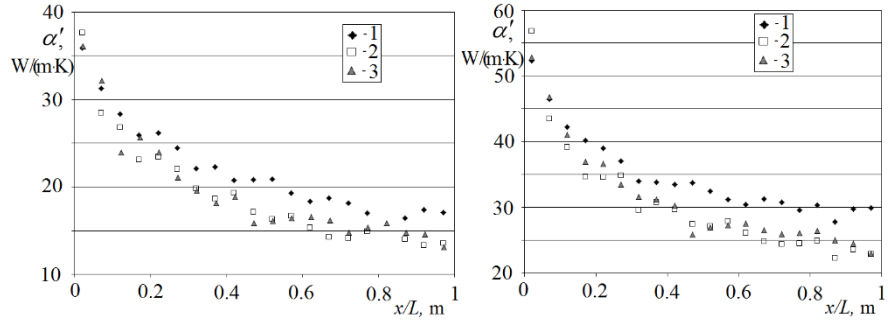


Рис. 5. Коэффициент теплоотдачи (а) для Re_l=56391, (б) для Re_l=112782 : 1 – подковообразная лунка; 2 – цилиндрическая лунка; 3 – подковообразная лунка, повернутая относительно потока на 45 градусов

Fig. 5. Heat transfer coefficient (a) for Rel = 56391, (b) for Rel = 112782: 1 - horseshoe hole; 2 - a cylindrical hole; 3 - horseshoe hole, rotated relative to the flow by 45 degrees

На рис. 6 представлены результаты численного исследования эффективной теплоотдачи и гидросопротивления для прямоугольных каналов с односторонним нанесением неглубоких лунок различной геометрии (рис. 1).

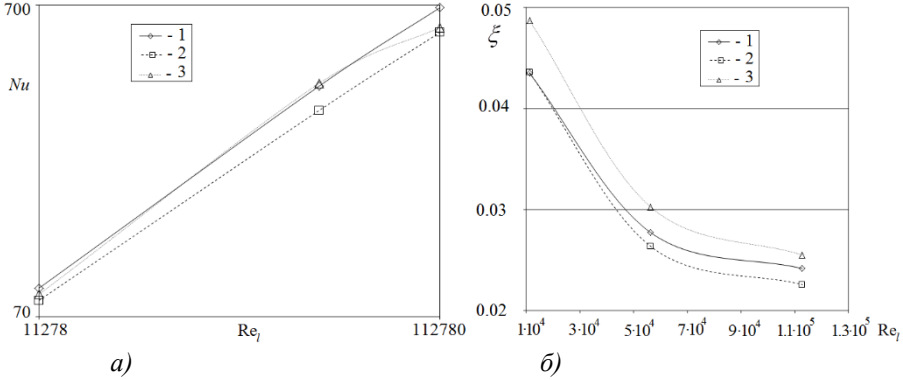


Рис. 6. Результаты численного моделирования (а) данные по теплоотдаче, (б) данные по гидравлическому сопротивлению: 1 – подковообразная лунка; 2 – цилиндрическая лунка; 3 – подковообразная лунка, повернутая относительно потока на 45 градусов

Полученные результаты показали, что коэффициенты теплоотдачи в рассмотренной области изменения скорости потока ($\underline{\bar{u}}_{00} = 1...10$ м/с) Re=11278…112780 для канала с подковообразными лунками (рис. 1, б) оказались выше, чем для канала с цилиндрическими лунками (рис. 1, а) или подковообразными лунками, повернутыми к потоку на 45 градусов (рис. 1, в).

Коэффициент гидравлического сопротивления канала с односторонним нанесением неглубоких подковообразных повернутых к потоку на 45 градусов

лунок (рис. 1, в) от 11 до 15 процентов выше, чем для канала с цилиндрическими лунками (рис. 1, а). Гидросопротивление канала с подковообразными лунками (рис. 1, б) только до 5 процентов выше, чем сопротивление канала с цилиндрическими лунками. При этом коэффициент теплоотдачи для канала с подковообразными лунками до 15,6 процента выше, чем для канала с цилиндрическими лунками. Анализ теплогидравлической эффективности каналов с односторонним нанесением неглубоких лунок различной конфигурации по результатам численных исследований представлен на рис. 7.

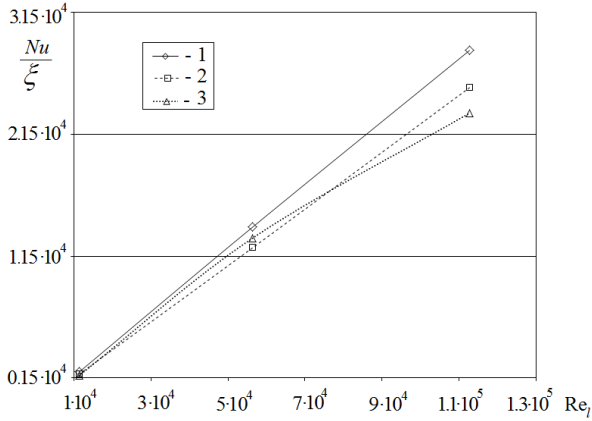


Рис. 7. Анализ теплогидравлической эффективности каналов с односторонним нанесением неглубоких лунок: 1 – подковообразная лунка; 2 – цилиндрическая лунка; 3 – подковообразная лунка, повернутая относительно потока на 45 градусов
Fig. 7. Analysis of thermal and hydraulic efficiency of channels with one-sided application of shallow holes: 1 - horseshoe hole; 2 - a cylindrical hole;3 - horseshoe hole, rotated relative to the flow by 45 degrees

Как видно из рис. 7, теплогидравлическая эффективность канала с подковообразными лунками выше, чем канала с цилиндрическими лунками или с подковообразными лунками, повернутыми к потоку на 45 градусов при развитом турбулентном течении потока.

5. Заключение

На базе открытого пакета Code Saturne проведено численное моделирование течения и теплообмена в прямоугольном канале односторонним нанесением неглубоких лунок. Проведена оценка построения качественной расчетной сетки применительно к течениям в каналах с неглубокими лунками. По средству сравнения результатов численного и физического эксперимента выполнена проверка адекватности разработанной модели, показавшая адекватность выбранных методов и инструментов численного исследования.

В результате исследования выявлено, что в каналах с неглубокими подковообразными лунками эффективность теплоотдачи до 15,6 процентов выше, чем в каналах с цилиндрическими лунками. При этом гидравлическое сопротивление канала с подковообразными неглубокими лунками, выполненными на одной из его сторон, оказывается до 5 процентов выше, сопротивления канала с цилиндрическими лунками. Так, при Re=11278 коэффициент гидравлического трения равен $\xi = 0.0487$, при Re=56390 только $\xi = 0.03$, при Re=112780 составляет $\xi = 0.0255$. Так что теплогидравлическая эффективность канала с подковообразными лунками выше эффективности канала с цилиндрическими лунками. Кроме того, полученные данные показали, что наклон подковообразных лунок по направлению к потоку значительно сказывается на величине гидравлического сопротивления и интенсивности теплоотдачи. Оказалось, что теплогидравлическая эффективность канала существенно зависит не только от конфигурации лунок, но и их расположения относительно потока.

Список литературы

[1]. А. Леонтьев, Н. Пилюгин, Ю. Полежаев, В. Поляев. Научные основы технологий XXI века. Москва, Энергомаш, 2000. 136 с.

[2]. Страница инструмента Code Saturne — http://code-saturne.org/cms/

[3]. Страница инструмента Open Foam — http://www.openfoam.com/

[4]. Ю. Гортышов, И. Попов, В. Олимпиев, А. Щелчков, С. Каськов. Теплогидравлическая эффективность перспективных способов интенсификации теплоотдачи в каналах теплообменного оборудования. Интенсификация теплообмена: монография. Казань: Центр инновационных технологий, 2009.

[5]. С. Исаев, А. Леонтьев, Н. Корнев, Э. Хассель, Я. Чудновский. Интенсификация теплообмена при ламинарном и турбулентном течении в узком канале с однорядными овальными лунками, Теплофизика высоких температур, том 53, №3, 2015 г., стр. 390-402. doi:10.7868/S0040364415030060

[6]. А. Цынаева, М. Никитин. Численное моделирование течения в канале с неглубокими лунками с использованием Code Saturne. Труды института системного программирования РАН, том 28 (выпуск 1), 2016 г., стр. 185-196. DOI: 10.15514/ISPRAS-2016-28(1)-10

[7]. А. Ильинков, А. Щукин, В. Такмовцев, И. Хабибуллин, В. Ильинкова. Особенности обтекания двухполостных выемок. Вестник КГТУ им. А.Н. Туполева, 2014 г., №2, стр. 53-57.

[8]. Страница инструмента Salome — http://www.salome-platform.org/

[9]. Страница инструмента On Shape — https://www.onshape.com/

[10]. Е. Менялкина, К. Кулясова, М. Никитин. Исследование работы воздушно-тепловой завесы с помощью численного моделирования. В сборнике: Традиции и инновации в строительстве и архитектуре. Строительные технологии сборник статей. Самарский государственный архитектурно-строительный университет; под ред. М.И. Бальзанникова, К.С. Галицкова, А.К. Стрелкова. Самара, 2016. С. 336-340.

[11]. M. Nikitin. Modeling of natural convection. Proceedings 7911583: 2nd International Conference on Industrial Engineering, Applications and Manufacturing, ICIEAM 2016.

Date of Conference: 19-20 May 2016. Date Added to IEEE Xplore: 27 April 2017. DOI: 10.1109/ICIEAM.2016.7911583

[12]. К. Подлипнова, М. Никитин. Численное исследование теплообмена с оребренной поверхностью. Труды Академэнерго. 2016. № 4. С. 42-49. https://elibrary.ru/item.asp?id=29217306

[13]. A. Tsynaeva, M. Nikitin. The technology of heat transfer enhancement in channels by means of flow pulsations. Proceding: MATEC Web of Conferences 5. Series "5th International Scientific Conference "Integration, Partnership and Innovation in Construction Science and Education", IPICSE 2016" 2016. P. 04018. DOI: 10.1051/matecconf/20168604018

[14]. Н. Ковальногов. Прикладная механика жидкости и газа. Ульяновск: Изд. УлГТУ, 2010.

[15]. F. Menter. Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications. AIAA Journal, 1994, vol. 32, № 8, pp. 1598-1605.

[16]. S. Isaev, P. Baranov, N. Kudryavtsev, D. Lysenko, A. Usachov. complex analysis of turbulence models, algorithms, and grid structures at the computation of recirculating flow in a cavity by means of vp2/3 and fluent packages. part 2. Estimation of models adequacy. Thermophysics and Aeromechanics, 2006. vol. 13, issue 1, pp. 55–65. DOI: 10.1134/S1531869906010060

[17]. В. Исаченко, В. Осипова, А. Сукомел. Теплопередача. Москва, Энергия. 1981. 440 с.

[18]. А. Цынаева, С. Разоренов Исследование потерь давления в канале с гантелеобразными лунками. Сборник. Традиции и инновации в строительстве и архитектуре. Строительные технологии сборник статей, 2016 г., стр. 345-349. — https://elibrary.ru/item.asp?id=25929084

[19]. А. Цынаева, С. Разоренов. Исследование течения в канале с подковообразными лунками. Материалы X Школы-Семинара молодых ученых и специалистов академика РАН В.Е. Алемасова. секц.4 №005. — http://x-school.knc.ru/files/prog2.pdf

[20]. И. Габдрахманов, А. Щелчков, И. Попов, С. Исаев. Применение пластинчатых теплообменных аппаратов с поверхностными интенсификаторами теплоотдачи в системах «EGR» для улучшения экологических характеристик ДВС. Вестник Казанского технологического университета, 2015 г., Т. 18. № 5. стр. 205-208.

Numerical modeling of heat transfer of channel with shallow curly dimples

[1] A. Tsynaeva <a.tsinaeva@rambler.ru>
[2] S. Razorenov <razserg@list.ru >
[1] V. Belaya <bonyparkery@gmail.com>
[1] Samara State Technical University,
244 Molodogvardeiskaya Str., Samara, 443001, Russian Federation.
[2] Tatneft-Privolga-Samara RNU,
100, Leninskaya Str., Samara, 443020, Russian Federation

Abstract. Numerical investigation is focused to heat transfer of rectangular channel with dimples. Developed model was tested for adequacy by simulating of experiment, conducted by Kazan National Research Technical University named after A. N. Tupolev – KAI.

Numerical model was verified too for adequacy by reproducing of experiment by A. Tsynaeva, S. Razorenov. The test has been held with Reynolds number Re=3000...30000. The results was found in enough agreement. The heat transfer in rectangular channel with shallow curly dimples was modeled with source Code_Sarurne. Numerical modeling are based by RANS approach with k-w SST model. The study was conducted to air (ν=13.28 10^{-6} m^2/s, $c_P = 1005$ J/(kg·K)). The 3D computation domain was meshed with source Salome by version 7.6.0. The SIMPLEC algorithm are used for U-p calculation. Generation time of mesh and calculation was estimated. The study of heat transfer was demonstrated by efficiency of curly dimples. Developed model shows heat transfer advantage up to 15.6 % of curly dimples over cylindrical ones of the same depth (*h/D*=0.11, *h*=2 mm) and contact patch area (S = 252.02 mm^2).

Keywords: free software, numerical simulation, heat transfer, flow, dimples.

DOI: 10.15514/ISPRAS-2017-29(5)-16

For citation: Tsynaeva A., Razorenov S., Belaya V. Numerical modeling of heat transfer of channel with shallow curly dimples. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 329-344 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-16

References

[1]. A. Leont'ev, N. Pilyugin, Yu. Polezhaev, V. Poljaev. Nauchnye osnovy tekhnologiy XXI veka [Scientific basis of technologies in XXI century]. Moscow, Energomash, 2000. 136 s. (in Russian)

[2]. Page of software Code Saturne — http://code-saturne.org/cms/

[3]. Page of software Open Foam — http://www.openfoam.com/

[4]. Y. Gortyshov, I. Popov, V. Olimpiev, A. Schelchkov, S. Kas'kov. Teplogidravlicheskaya ehffektivnost' perspektivnyh sposobov intensifikacii teplootdachi v kanalah teploobmennogo oborudovaniya. Intensifikaciya teploobmena: monografiya. [Thermohydraulic efficiency of the perspective ways of intensification of heat transfer in channels heat transfer equipment. Intensification of heat transfer: monograph]. Kazan. [Kazan], Centr innovacionnyh tekhnologij [Center for innovative technology], 2009. (In Russian)

[5]. S. Isaev, A. Leontiev, N. Kornev, E. Hassel, Y. Chudnovskii. Heat transfer intensification for laminar and turbulent flows in a narrow channel with one-row oval dimples in High Temperature, vol 53, Issue 3, 2015. pp. 375-386. doi: 10.7868/S0040364415030060

[6]. A. Tsynaeva, M. Nikitin. Chislennoe modelirovanie techenija v kanale s neglubokimi lunkami s ispol'zovaniem Code Saturne [Numerical modeling of rectangular channel with shallow dumbbell dimples based Code Saturne]. Trudy ISP RAN [Proceedings of ISP RAS], Volume 28 (Issue 1), 2016, pp. 185-196 (in Russian). DOI: 10.15514/ISPRAS-2016-28(1)-10

[7]. A. Linkov, A. Schukin, V. Tokmovtsev, I. Habibullin, V. Ilenkova Osobennosti obtekaniya dvuhpolostnyh vyemok. [Features of the flow past of bicameral dimples]. Vestnik KGTU im. A.N. Tupoleva. [Bulletin of KSTU by Tupolev], Issue 2, 2014, pp. 53-57 (in Russian).

[8]. Page of software Salome — http://www.salome-platform.org/

[9]. Page of software On-Shape — https://www.onshape.com/

[10]. E. Menyalkina, K. Kulyasova, M. Nikitin. Issledovanie raboty vozdushno-teplovoj zavesy s pomoshchyu chislennogo modelirovaniya. [Study of air curtain with the help of numerical simulation] V sbornike: Tradicii i innovacii v stroitelstve i arhitekture stroitelnye tekhnologii sbornik statej, Samarskij gosudarstvennyj arhitekturno-stroitelnyj universitet pod red. M. I. Balzannikova, K. S. Galickova, A. K. Strelkova. [Procedings: Traditions and innovations in construction and architecture. Construction techniques a collection of articles, Samara State University of Architecture and Construction; Editors are M. I. Balzannikov, K. S. Galitskov, A. K. Strelkov]. Samara, 2016. pp. 336-340 (in Russian).

[11]. M. Nikitin. Modeling of natural convection. Proceedings 7911583: 2nd International Conference on Industrial Engineering, Applications and Manufacturing, ICIEAM 2016. Date of Conference: 19-20 May 2016. Date Added to IEEE Xplore: 27 April 2017. DOI: 10.1109/ICIEAM.2016.7911583

[12]. K. Podlipnova, M. Nikitin. Chislennoe issledovanie teploobmena s orebrennoj poverhnostyu [Numerical study of heat transfer from finned surface]. Trudy Akademehnergo. [Proceding of Academenergo], Issue 4, 2016, pp. 42-49 (in Russian). https://elibrary.ru/item.asp?id=29217306

[13]. A. Tsynaeva, M. Nikitin. The technology of heat transfer enhancement in channels by means of flow pulsations. Proceding: MATEC Web of Conferences 5. Series "5th International Scientific Conference "Integration, Partnership and Innovation in Construction Science and Education", IPICSE 2016" 2016. P. 04018. DOI: 10.1051/matecconf/20168604018

[14]. N. Kovalnogov. Prikladnaya mekhanika zhidkosti i gaza [Applied mechanics of liquid and gas]. Ul'yanovsk [Ulyanovsk]: UlGTU [UlSTU], 2010. (in Russian).

[15]. F. Menter. Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications. AIAA Journal, 1994, vol. 32, issue 8, pp. 1598-1605.

[16]. S. Isaev, P. Baranov, N. Kudryavtsev, D. Lysenko, A. Usachov. complex analysis of turbulence models, algorithms, and grid structures at the computation of recirculating flow in a cavity by means of vp2/3 and fluent packages. part 2. Estimation of models adequacy. Thermophysics and Aeromechanics, 2006. vol. 13, issue 1, pp. 55–65. DOI: 10.1134/S1531869906010060

[17]. V. Isachenko, V. Osipova, A. Sukomel. Teploperedacha. [Heat transfer]. Moscow, 1981. 440 pp. (in Russian)

[18]. A. Tsynaeva, S. Razorenov Issledovanie poter' davleniya v kanale s ganteleobraznymi lunkami [Investigation of pressure losses in a channel with a dumbbell dimples]. Sbornik. Tradicii i innovacii v stroitel'stve i arhitekture. Stroitel'nye tekhnologii sbornik statej [Proceedings of Tradition and innovation in construction and architecture. Construction techniques.], 2016. pp. 345-349 (in Russian). https://elibrary.ru/item.asp?id=25929084

[19]. A. Tsynaeva, S. Razorenov Issledovanie techeniya v kanale s podkovoobraznymi lunkami. [Investigation flow by channel with a curly dimples] Materialy X SHkoly-Seminara molodyh uchenyh i specialistov akademika RAN V.E. Alemasova. sekc.4 No.005. [In: proceedings of X School-Seminar of young scientists and specialists of academician V. E. Alemasov. Section No. 005.]. (in Russian) — http://x-school.knc.ru/files/prog2.pdf

[20]. I. Gabdrakhmanov, A. Shchelchkov , I. Popov, S. Isaev. Primenenie plastinchatyh teploobmennyh apparatov s poverhnostnymi intensifikatorami teplootdachi v sistemah «egr» dlya uluchsheniya ehkologicheskih harakteristik DVS [The application of plate heat exchangers with surface intensifiers of heat transfer in the systems "egr" to improve the environmental performance of the engine]. Vestnik Kazanskogo tekhnologicheskogo universiteta [Proceedings of Kazan technological University], 2015, Vol. 18. Issue 5. pp. 205-208.