UDC 004.451,004.415.5

Verification of Operating System Components

Alexey V. Khoroshilov (Institute for System Programming of RAS)

Victor V. Kuliamin (Institute for System Programming of RAS)

Alexander K. Petrenko (Institute for System Programming of RAS)

The paper concerns recent advances in reaching the goal of industrial operating system (OS) verification. By industrial OS we mean a system actively used in some industrial domain, elaborated and maintained for a significant time, not a proof-of-concept OS developed with mostly research intentions. We consider decomposition of this goal into tasks related with various functional components of OS and various properties under verification, and application of different verification methods to those tasks. This is a trial to explicate and summarize the experience of several projects on various OS components and different OS features verification conducted in ISP RAS.

Keywords: Operating system, verification, testing, monitoring, static analysis, deductive verification

1. Introduction

Modern industrial operating systems, which are used for plenty of real-life applications, are rather complex. They are not just very large pieces of code, they also have a great number of heterogeneous features, should operate on a large variety of hardware from diverse manufacturers, and are to provide for application developers numerous interfaces, which are expected not only to work correctly, but also to use underlying hardware in effective, efficient, and fault-tolerant way. By industrial operating system (OS) we mean in this article an OS actively used in some industrial domain (or a general purpose one), elaborated and maintained for a significant time. We do not discuss here OSes developed with certain research purposes or as a proof-of-concept, they may be much more simple than industrial ones and have different specifics.

An OS is supposed to perform two main tasks.

- It should organize operation of multiple applications on some machine, managing hardware resources, and protect applications from interfering each other.
- It should provide interface for application developers to use those resources in a convenient way, and also to transfer data between applications, if needed.

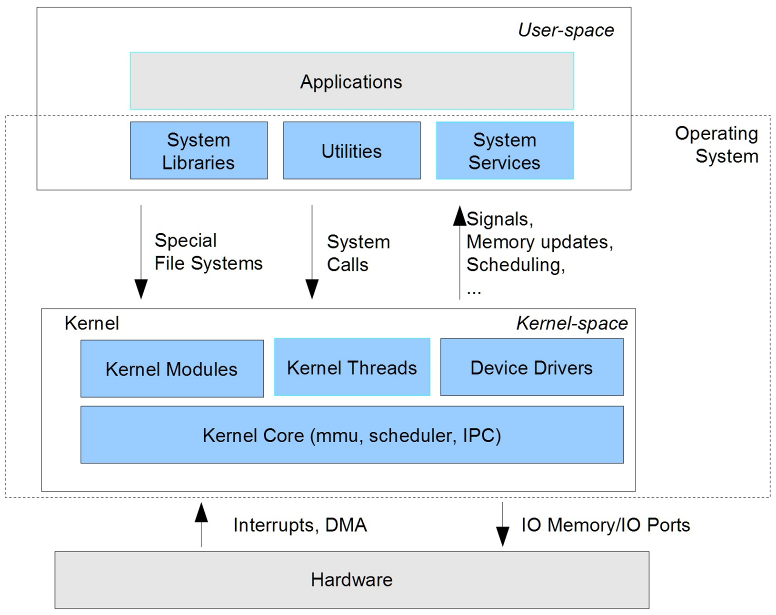

Fig. 1. Main parts of general purpose OS

The main part of an OS is *a kernel,* which works in privileged processor mode (kernel mode) and so has unbounded access to all system resources. Kernel manages application access to hardware resources, sets access policies, and prevents their violation. Some functions that do not require privileged mode are also sometimes included into kernel for efficiency.

Applications can interact with kernel mostly with *system calls,* which are calls of kernel functions with switch into privileged mode. There are additional ways to interact with kernel, like special file systems (procfs, sysfs, debugfs) in Linux. To provide convenient environment for application developers OS usually provides *system libraries and utilities,* implementing frequently used functions that require interaction with kernel. To solve tasks that need activity from the kernel side, *system services* are provided. Such tasks include communication protocols, managing special devices, etc. Corresponding services can work in kernel mode or in user mode. Figures 1 and 2 show the structure of general purpose OS and real-time OS correspondingly.

The above sketchy review of OS structure gives some hints on its complexity. The verification of industrial OS is also rather complex, especially if one takes into consideration the following.

- A plenty of features of modern OS provide various feature interaction cases and corner cases, where much more scrutinized inspection of required behavior is necessary.
- Multitasking support in modern OS makes checking behavior correctness much more intricate.
- Basic OS functionality must be available in spite of some faults in hardware or software

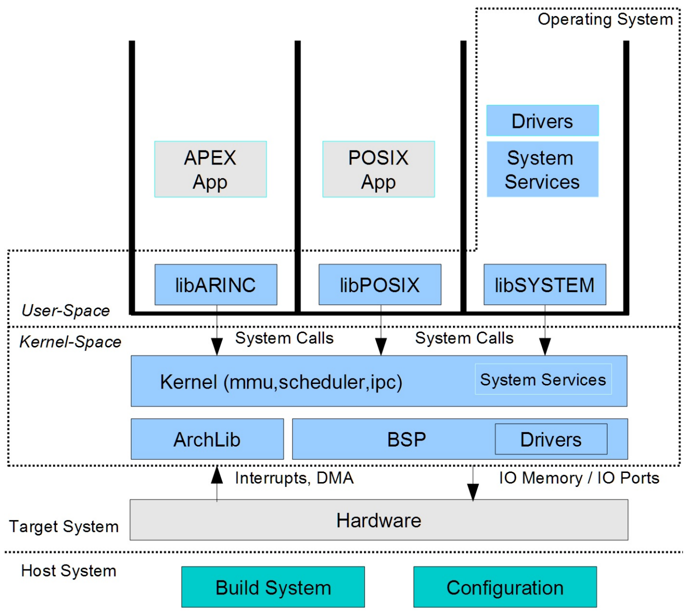

Fig. 2. Main parts of real time OS

components. This fault tolerance should also be verified.

- Modern OS usually supports network communications and provides a workplace for many users. It ensures certain security policies, setting restrictions on data and operations available for different users and processes. Those restrictions should be preserved in case of faults and attacks from malicious users or software components coming from network.
- Support of huge variety of heterogeneous hardware is usually implemented as deep configurability of an OS. So, one needs verification of OS behavior in various possible configurations, the total number of which is usually larger than astronomical numbers.
- The mere code size and number of functions in modern OS are great. The size of Linux kernel version 4.1 is reported [1] to be about 20.3 million lines of code (LOC), while drivers part, which is created and supported by diverse developers and is responsible for most of bugs is about 11.5 million LOC. The size of Windows XP is estimated as 45 million LOC [2]. The number of functions in system libraries of Debian 7.0 is about 720 thousand [3], while the number of system calls is about 350.

The numbers provided make the goal of thorough verification of industrial OS unreachable at this moment. Nevertheless, the developer community needs some methods to assure correctness, efficiency, security, and fault tolerance of modern OSes. The only reasonable way to help is to use various available verification methods to ensure those properties for some parts of OS code or functionality, or to ensure only a few critical properties on OS as a total. Important results of partial verification are faults and errors found, so, although we cannot now guarantee strict correctness or security of an OS, we can come closer to them (if we do not forget about that ultimate goal in routine quality assurance processes). Thus, increasing scalability of used methods and tools step by step, we can enlarge the verified parts of code and functionality, targeting to reach the goal of complete verification somewhen in future.

In this paper we review the activity on verification of different parts and features of OSes performed in the Institute for System Programming of Russian Academy of Sciences in a dozen of projects conducted last 20 years. This activity uses and integrates different methods of verification for different OS properties and components, applying them to Linux OS and several specialized real-time OSes. The main methods used are as follows.

- **Testing and dynamic analysis.** Testing can be performed in different ways. The most lightweight testing focuses on producing test cases for basic behavior of functions, skipping consideration of complex or even non-so-often cases. The main target test completeness criterion for such testing is coverage of functions. The most thorough testing is intended to provide as strict and accurate checks as possible. It uses formal specification of required behavior, tries to formulate presumptions and strict guarantees concerning the correctness of tested system, and is targeted on coverage of all conditions met both in code and requirements, along with some corner cases (buffers overflow, failures of underlying hardware, processing simultaneous events, etc.). Various testing methods between these two extremes are also used. The main properties under test are functional, but testing is also the main method to check efficiency and fault tolerance. Along with testing other dynamic analysis methods, not requiring preparation of test suites, can be used.
- **Static analysis.** Static analysis also presents a wide range of approaches, from simple, quick, and lightweight checkers seeking a bounded number of bug patterns and producing a lot of false positives — reports on bugs, which actually aren't, to rather complex tools using formal specifications and configurable analyses, capable to catch very intricate bugs, requiring large effort during their configuration, usually with not-so-high numbers of false positives. Static analysis is widely used to check various code, but usually more complex and powerful techniques are applied to components with more strict requirements, like OS kernel modules.
- **Deductive verification.** Deductive verification is used to verify most important security or correctness properties. There are well-known examples of OS kernel verification [4–6],

but in all cases the verified code is much smaller than kernels of typical industrial OS. Nevertheless, deductive verification techniques can be applied to industrial OS code and provide valuable results.

Below we explicate and try to summarize the experience obtained by ISP RAS in dozens of projects, where some kind of verification was performed on various components of industrial OSes (Linux and several real-time OSes for specific domains). The paper organized as follows. In Section 2 we provide review of testing techniques used to check different OS components. Section 3 reports on application of various static analysis techniques, usually in conjunction with some dynamic analysis. Section 4 describes our results in deductive verification of OS security. Then the Conclusion sums up the exposition and describes possible further development.

2. Testing and dynamic analysis

ISP RAS develops OS components testing methods since its foundation in 1994. The first such results are related with KVEST [7], a method for test generation based on formal specifications of functional behavior in form of software contracts, used to construct several test suites for real-time OS developed and maintained by Nortel Networks.

2.1. Formal approaches

Later this approach was refined and extended into UniTESK method [8]. The basic ideas of the method are as follows.

- Requirements to library functions behavior are specified as *software contracts* — preconditions, postconditions, and data type invariants (they may be considered as common parts of pre- and postconditions of all functions dealing with those data types). Software contracts are written in extension of C language or with the help of specialized libraries in pure C/C++.
- Test completeness criteria are formulated as coverage of branches in postconditions. If there is a need to add some situations to coverage goals, they are formulated as specific additional branches, not related with behavior restrictions.
- Test scenarios are represented as extended finite state machines, for which execution of all reachable transitions guarantees coverage of all coverage goals (branches) specified in postconditions of functions called (each transition corresponds to a sequence of function calls). The control state of test scenario is a generalization of data structures used in

specifications of functions tested by this scenario.

- Testing is performed by automatic traversal of a state machine defined in test scenario. Each call to a function under test is augmented with call to the oracle function generated from postcondition and evaluating correctness of the results obtained.
- Testing of parallelism is based on interleaving semantics [9]. It is performed by gathering all the observed events (function calls, function returns, and others) and constructing a linear sequence of those events, in which all pre- and postconditions hold. If such a sequence cannot be constructed, a bug is recorded.

UniTESK was used for conformance test suite creation for Core part of Linux Standard Base (LSB), which describes system libraries and almost coincides with POSIX, in OLVER Project [10], where 1532 functions of LSB Core was formally specified and tested. The same method was applied in conformance test suite development for ARINC-653 part 1 standard [11] describing 54 functions.

Another test construction method, not using formal specifications, but based on formal investigation of requirements was used to create conformance tests for mathematical functions working with floating-point numbers in POSIX system libraries [12]. The method uses as test data specific floating point values, including numbers having patterns in mantissa (like 0000FFFFAAAA in hexadecimals), boundaries of domains of specific function behavior (such behaviors include monotonicity, sign preservation, well-known asymptotics), and so-called *worst cases,* numbers, for which correct function calculations requires much more precision than in average. For now test suites for 104 functions was developed.

2.2. Informal approaches

Several other methods used for test construction in ISP RAS are not based on formal specifications, but targeted on strict requirements traceability, so that tests are developed to check certain explicitly formulated requirements and they report on violation of this requirements (providing their ids) when find some bug.

The first method [13] is based on manual test case development with further parameterization making a test case a template. For test execution each template is supplemented with several arrays of arguments that are put in place of corresponding parameters. The method was used to create tests for more than 4000 functions in Linux system libraries, they detected about 40 bugs.

Another approach [14] provides automatic generation of sanity tests (checking only basic functionality) on the base of initialization procedures for data type values and libraries and preconditions of functions specified manually and stored in a database. This method provides rather surface testing, but can be used for massive test generation with little effort. It was applied to Linux libraries containing about 20000 functions.

2.3. Fault tolerance testing and dynamic analysis

For monitoring Linux kernel modules KEDR framework [15] was developed in ISP RAS. It makes possible to intercept calls from single kernel module, and so to observe its behavior in dynamics. On the base of KEDR the following verification techniques are implemented.

- KEDR Leak Check used to detect memory leaks in kernel modules. It is more convenient for leak detection then kmemleak [16] included in Linux distribution, but cannot be used to check kernel core code.
- Kernel Strider [17] used to detect data races, situations when several threads read and write one region of memory in unordered manner. Kernel Strider gathers information on module execution, which is then analyzed by ThreadSanitizer [18], data race detection tool developed by Google.
- KEDR Fault Simulation [19] used for fault tolerance testing. The testing organized in a following way. First, the module under test is executed in ordinary way and KEDR detects all calls to functions (system calls or calls of hardware-specific operations) that can fail, but very rare do this during real work. Second, for each call the test is executed, in which this call is simulated as failed. This approach helped to detect several bugs in mature file system drivers like ext4.

A specific example of monitoring used to detect data races is given by RaceHound tool [20], which implements the same idea as DataCollider [21]. It detects memory regions where a thread can write, sets hardware breakpoint on access to such regions, and inserts additional wait intervals around memory access operations in other threads in runtime. If this leads to an access to the tapped memory from another thread, a data race is reported.

3. Static analysis

To get more efficiency a large part of general purpose OS code is working in kernel mode, where it has many possibilities to damage important OS data structures. Since the code of

Linux drivers, which also works in kernel mode, is usually written by developers having good knowledge of hardware and not-so-good in rules of correct operation within Linux kernel, this naturally leads to the situation where more than a half of bugs detected in kernel is related with drivers code [22]. The similar relation is true for Windows OS [23].

To make development of kernel modules less error-prone, one needs specific tools that can check the rules of correct kernel application program interface (API) usage. Microsoft Research offers Static Driver Verifier tool [24] (called SLAM earlier) capable to solve this task for Windows. The similar solution is suggested by ISP RAS for Linux under the name of Linux Driver Verification (LDV) framework [25, 26]. The method used by LDV is the following.

- The rules of correct kernel API usage are specified as software contracts in specific notation extending C language. They are interpreted as aspect advices that should be inserted at the points where the specified API functions are called in the module under check. Being inserted in the module code, advice code creates error, if the rules specified are violated.
- The usage model is created for the module functions. This is important for driver modules, since their functions are not called explicitly. The usage model defines all possible sequences of function calls.
- The code of the module under check is processed by aspect weaver, which inserts rule checking code, and augmented by the usage model.
- The main check is performed by static verifier tool (most often BLAST [27] and CPAChecker [28] are used). The tool analyzes the code trying to solve reachability task — whether the error creation instruction can be reached in some execution. If it is reachable, then the corresponding execution scenario demonstrates a bug, incorrect use of kernel API, else the code uses the API functions correctly. Reachability task is solved with the help of counterexample guided abstraction refinement technique (CEGAR) [29], which constructs automatically more and more precise models of code execution, until the error-reaching path in model can be re-executed in real code, or becomes unreachable in the refined model.

LDV detects 5-8 bugs in almost each release of Linux kernel, for now the total number of found bugs is about 2500. It is used routinely to check about 4000 kernel modules.

Another example of static analysis usage is provided by a tool CPALocator [30] developed on the base of CPAChecker and used to search race conditions in OS code.

4. Deductive verification

Deductive verification is usually considered as the most strict and accurate verification technique, at the same time it requires a lot of effort and highly qualified staff to perform it in a productive way. A good review of deductive verification use for OS code is provided by [31].

In ISP RAS projects deductive verification was used to verify security properties of a Linux-based OS modified for specific use in government agencies [32, 33]. The OS is intended to implement a complex security model (called MROSL DP) integrating mechanisms of lattice-based mandatory access control, mandatory integrity control, and role-based access control. All the security mechanisms are implemented with the help of Linux Security Module (LSM) [34], which provides interceptor functions for all access operations in Linux.

First, MROSL DP model was formalized in Event-B and its main security properties (that no subject with less access level can get access to an object with higher confidentiality level; no subject can get access to an object, for which the subject has no a role having right to access to, etc.) were proved. Second, main LSM functions were also formally specified in so-called detailed model, for which the corresponding security properties were also proved. On the third step the contracts of LSM functions should be translated in ACSL, an extension of C language used in code verification framework Frama C/Jessie [35], and this framework should verify the behavior of C code on conformance with the contracts.

Althoug the project is not finished yet, a number of faults was found in the security model itself due to formalization, and several bugs were detected in code during its partial verification.

5. Conclusion

In this paper we provide a systemized review of verification activities used to check various components and features of industrial OS in ISP RAS projects. Although the ultimate goal — the thorough verification of an OS widely used in real-life — still remains unreachable, our experience shows that important advances in that direction were made by research and development community in last years.

The methods and tools developed for different purposes and using different basic approaches — testing, monitoring, static analysis, deductive verification — can enrich each other by borrowing specific modeling or reasoning technique, as it can be shown on example of memory modeling in static analysis and deductive verification tools [36].

One also can see during a last decade an impressive progress in verification techniques applicable to real software. In the domain of OS verification such progress can be illustrated by a method for deductive verification of multithreaded C programs working with shared data proposed in ISP RAS [37]. We hope that in one-two years it will be implemented and we can see results of its experimental evaluation.

Another direction of future research concerns possibilities to reuse verification artefacts created by some methods in other ones [38].

References

1. Why is the Linux kernel 15+ million lines of code? https://unix.stackexchange.com/questions/223746/why-is-the-linux-kernel-15-million-lines-of-code/223770. Aug 2015.
2. How Many Lines of Code in Windows XP? https://www.facebook.com/windows/posts/155741344475532. Jan 2011.
3. Gerlits E.A., Kuliamin V.V., Maksimov A.V., Petrenko A.K., Khoroshilov A.V., Tsyvarev A.V. Testing of Operating Systems // Trudy ISP RAN/Proc. ISP RAS, 2014, 26(1):73-108 (in Russian).
4. Bevier W.R. Kit: a Study in Operating System Verification // IEEE Transactions on Software Engineering, 15(11):1382-1396, Nov 1989.
5. Alkassar E., Paul W.J., Starostin A., Tsyban A. Pervasive Verification of an OS Microkernel // In: Leavens G.T., O'Hearn P., Rajamani S.K. (eds) Verified Software: Theories, Tools, Experiments. VSTTE 2010. Springer, LNCS 6217:71-85.
6. Klein G., Andronick J., Elphinstone K., Murray T., Sewell T., Kolanski R., Heiser G. Comprehensive Formal Verification of an OS Microkernel // ACM Transactions on Computer Systems, ACM, 2014. 32(1), art. 2.
7. Burdonov I., Kossatchev A., Petrenko A., Galter D. KVEST: Automated Generation of Test Suites from Formal Specifications // In: Wing J.M., Woodcock J., Davies J. (eds) FM'99 – Formal Methods. FM 1999. Springer, LNCS 1708:608-621.
8. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. UniTesK Test Suite Architecture // In: Eriksson L.H., Lindsay P.A. (eds) FME 2002:Formal Methods – Getting IT Right. FME 2002. Springer, LNCS, 2391:77-88.
9. Kuliamin V.V., Petrenko A.K., Pakoulin N.V., Kossatchev A.S., Bourdonov I.B. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems // In: Broy M., Zamulin A.V. (eds) Perspectives of System Informatics. PSI 2003. Springer, LNCS 2890:450-461.
10. Grinevich A., Khoroshilov A., Kuliamin V., Markovtsev D., Petrenko A., Rubanov V. Formal Methods in Industrial Software Standards Enforcement // In: Virbitskaite I., Voronkov A. (eds) Perspectives of Systems Informatics. PSI 2006. Springer, LNCS 4378:456-466.
11. Maksimov A. Requirements-based conformance testing of ARINC 653 real-time operating systems // Proc. of Data Systems In Aerospace (DASIA 2010), ESA SP-682.

12. Kuliamin V. Standardization and Testing of Mathematical Functions. // Proc. of Perspectives of System Informatics, PSI 2009. Springer, LNCS 5947:257-268.
13. Khoroshilov A., Rubanov V., Shatokhin E. Automated Formal Testing of C API Using T2C Framework // Proc. of International Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008), pp.56-70.
14. Zybin R.S., Kuliamin V.V., Ponomarenko A.V., Rubanov V.V., Chernov E.S. Automation of broad sanity test generation // Programming and Computer Software, 34(6):351-363, 2008.
15. Shatokhin E. Using Dynamic Analysis To Hunt Down Problems in Kernel Modules // Presentation at LinuxCon Europe 2011, Czech Republic, Prague, 26-28 October 2011.
16. kmemleak description. https://www.kernel.org/doc/Documentation/kmemleak.txt.
17. Kernel Strider. https://code.google.com/p/kernel-strider/.
18. Serebryany K., Iskhodzhanov T. ThreadSanitizer: data race detection in practice // In Proc. of Workshop on Binary Instrumentation and Applications (WBIA 2009). ACM, 2009, pp.62-71.
19. Tsyvaerv A., Khoroshilov A. Using Fault Injection for Testing Linux Kernel Components // Trudy ISP RAN/Proc. ISP RAS, 2015, 27(5):157-174 (in Russian).
20. Race Hound tool. http://forge.ispras.ru/projects/race-hound.
21. Erickson J., Musuvathi M., Burckhardt S., Olynyk K. Effective data-race detection for the kernel // Proc. of USENIX conference on Operating systems design and implementation, OSDI 2010, pp. 151-162.
22. Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analysis of typical faults in Linux operating system drivers // Trudy ISP RAN/Proc. ISP RAS, 2012, 22:349-374 (in Russian).
23. Ball T., Levin V., Rajamani S.K. A decade of software model checking with SLAM // Communications of the ACM, vol. 54, issue 7, pp. 68-76, 2011.
24. Ball T., Bounimova E., Cook B., Levin V., Lichtenberg J., McGarvey C., Ondrusek B., Rajamani S.K., Ustuner A. Thorough static analysis of device drivers // Proc. of ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), pp. 73-85, 2006.
25. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Architectire of Linux Driver Verification // Trudy ISP RAN/Proc. ISP RAS, 2011, 20:163-187 (in Russian).
26. Zakharov I.S., Mandrykin M.U., Mutilin V.S., Novikov E.M., Petrenko A.K., Khoroshilov A.V. Configurable Toolset for Static Verification of Operating Systems Kernel Modules // Trudy ISP RAN/Proc. ISP RAS, 2014, 26(2):5-42 (in Russian).
27. Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST: Applications to software engineering // International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007.
28. Beyer D., Keremoglu M.E. CPAchecker: A tool for configurable software verification // Proc. of International Conference on Computer Aided Verification (CAV 2011), Springer, LNCS 6806:184–190.
29. Clarke E., Grumberg O, Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement // In: Emerson E.A., Sistla A.P. (eds) Computer Aided Verification. CAV 2000. Springer, LNCS 1855:154-169.

30. Andrianov P.S., Mutilin V.S., Khoroshilov A.V. Adjustable method with predicate abstraction for detection of race conditions in operating systems // Trudy ISP RAN/Proc. ISP RAS, 2016, 28(6):65-86 (in Russian).
31. Klein G. Operating system verification – An overview // Sadhana, 2009, 34(1):27-69.
32. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Formal Verification of OS Security Model with Alloy and Event-B // Proc. of Int. Conf. on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2014), pp. 309–313.
33. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Comparison of specification decomposition methods in Event-B // Programming and Computer Software, 2016, 42(4):198-205.
34. Wright C., Cowan C., Morris J., Smalley S., Kroah-Hartman G. Linux Security Module Framework // In: Ottawa Linux Symposium, vol. 8032, 2002.
35. Marhé C., Moy Y. The Jessie Plugin for Deductive Verification in Frama-C // INRIA Saclay Île-de-France and LRI, CNRS UMR, 2012.
36. Mandrykin M.U., Mutilin V.S. Survey of memory modeling methods in static verification tools // Trudy ISP RAN/Proc. ISP RAS, 2017, 29(1):195-230 (in Russian).
37. Mandrykin M.U., Khoroshilov A.V. Towards deductive verification of C programs with shared data // Programming and Computer Software, 2016, 42(5):324-332.
38. Petrenko A.K., Kuliamin V.V., Khoroshilov A.V. Integration Points of Operating System Verification Techniques // Trudy ISP RAN/Proc. ISP RAS, 2015, 27(5):175-190 (in Russian).