Uncertainty problem in high-level model based trace analysis as part of runtime verification

Aleksei Karnov
Software Engineering Department
Ivannikov Institute for System Programming of the RAS
Moscow, Russia
karnov@ispras.ru

***Abstract*—The article discusses the problem of applying runtime verification to large and complex systems such as general-purpose operating systems. When verifying the security mechanisms of operating systems, modern practices and standards require a formal security policy model (SPM). The SPM must be verified using formal model methods, and it must also be used to verify the completeness and consistency of the operating system's security mechanisms by confirming compliance with the formal requirements of the SPM. In this case, it is convenient to have a single model suitable for both formal verification and implementation testing. For practical application, it is necessary, on the one hand, to select a subset of model language constructs suitable for both acts, and on the other hand, to develop special techniques for analyzing execution traces that allow to effectively perform thousands of test cases. The article addresses both of these issues. We present an analysis of language constructs that allow us to use the model for both verification and execution trace analysis. We also offer techniques that have been developed to optimize the runtime verification of Linux-based systems.**

***Index Terms*—runtime verification, trace analysis, Event-B**

I. Introduction

Runtime verification [1] is a computing system analysis and execution approach based on information extracting from a running system and using it to detect observed behaviors satisfying or violating certain properties. By observed behavior we mean a sequence of events in order of their occurrence in the system. Each event is supplied with data which may contain the parameters of the event and its result. This sequence of events with data is called an execution trace [2].

Runtime verification can be also defined as a collection of formal methods for studying the dynamic evaluation of execution traces against formal specifications. We consider a special case when a formal model of the system under tests acts as a formal specification. The complexity of the model in terms of size and number of internal details is usually less than the complexity of the system. This makes the model a more convenient object to apply formal verification techniques. This approach can significantly improve the reliability of runtime verification results.

Leaving aside the process of collecting the execution trace, we are dealing with the analysis of the execution trace and the formal model. From this point of view, the implementation of the system does not matter and is only a source of execution traces. In this paper we use the term runtime verification only in the meaning of trace analysis and do not consider testing and system instrumentation techniques, skipping the stages of tests derivation and test runs on the system.

Our research aims to effectively carry out this analysis and to overcome the problems that we inevitably encounter. Section II explains our motivation for tools and approaches we are using and enumerates goals of the research. Section III contains a brief description of the problem with the example. Section IV provides a brief overview of existing approaches to trace analysis and model execution. The current results of the study are presented in Section V. Section VI lists future directions of work.

II. Motivation

When it comes to mission-critical systems such as information security, modern standards and practices require [3] a formal security policy model (SPM). The SPM must be verified [4] using formal methods. One of the most important advantages of runtime verification in our areas of applications is the ability to use the same formal model for both formal specification and testing. With this approach, we can create a closed-loop verification system, where the correctness of the observable behavior of the system under test will be compared with the verified SPM.

This is the reason we need to present a formal model of the target system in a form that is both convenient for formal verification methods and runtime verification. It turns out that in practice such a representation is difficult to offer. One of the possible solutions is proposed in this paper.

The main practical application of the proposed approach is verification of information security measures of operating systems. There is a lot of work in this area [5], [6], [7]. The Event-B language and tools for working with it were chosen as the main means of describing formal access control models. Event-B has the advantages of simple and understandable language constructs and Rodin [8] – a convenient IDE for model development including deductive verification tools and many other useful plugins. These plugins also include tools for model animation, which demonstrates the fundamental ability to use the language for trace analysis.

So there are no serious reasons to abandon Event-B but there are some non-trivial problems that complicate and sometimes prevent the use of these models for runtime verification. The goals of this research are:

1) To conduct an analysis and classification of language constructs and techniques for writing specifications in Event-B, which complicate runtime verification;
2) To select a subset of the language and templates for writing fragments of specifications that either eliminate the problems of trace analysis or allow one to transform models and bring them into a form convenient for runtime verification;
3) To develop a set of model transformation techniques;
4) To develop analysis methods for models reduced to a form convenient for runtime verification.

III. Uncertainty problem

Let us consider a simple traffic light model. The model state contains two boolean variables:

$$cars\_go \in BOOL \tag{1}$$

$$peds\_go \in BOOL \tag{2}$$

To ensure traffic safety, the model contains an invariant that prohibits cars and pedestrians from moving at the same time:

$$\neg(cars\_go = TRUE \wedge peds\_go = TRUE) \tag{3}$$

As the initial state we can take any state that does not violate invariant 3. The events $cars$ and $peds$ have a parameter go and can change state variables to the value of the parameter. To ensure that invariant 3 is not violated, events have enabling conditions which are called guards. Guard 4 refers to $cars$ and guard 5 refers to $peds$.

$$go = TRUE \Rightarrow peds\_go = FALSE \tag{4}$$

$$go = TRUE \Rightarrow cars\_go = FALSE \tag{5}$$

The state space of the model is shown in Fig. 1.

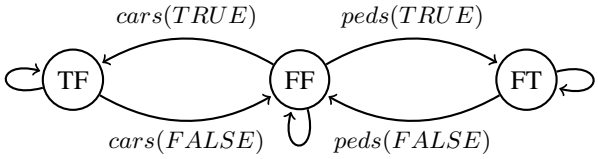

Fig. 1. Traffic light state space.

But we can't track traffic directly, so we need to add a context to the model that contains the colors of the traffic lights:

$$red \in COLORS \tag{6}$$

$$green \in COLORS \tag{7}$$

$$red \neq green \tag{8}$$

The observed events will have a parameter $colors$ that corresponds to the set of traffic signals, and the parameter go will not be present in the collected trace. For the new parameter, we must add guards 9–11 to both events.

$$colors \subseteq COLORS \tag{9}$$

$$green \in COLORS \Rightarrow go = TRUE \tag{10}$$

$$green \notin COLORS \Rightarrow go = FALSE \tag{11}$$

This model is complete and consistent if we assume that the presence of a green light at a traffic light allows movement regardless of the combination of other colors. But when trying to execute the model, we are faced with numerous uncertainties.

Let us consider the trace $\{cars(\{green\}),\ cars(\{red\}),\ peds(\{green\})\}$. The trace satisfies the model only if the initial state of the variable $peds\_go$ is $FALSE$: FF and TF from Fig. 1. But we also allowed the initial state to be FT. We may encounter this situation in Event-B when we use uncertain assignment actions. There are three types of assignment actions: simple assignment, arbitrary choice from set of values and a choice from a predicate. We can't fully ignore them because a choice from a predicate is the only way to model if-then-else construction but to be certain it needs to be written in the specific way.

If we analyze the trace from a real traffic light, we can encounter events like $cars(\{yellow\})$, $cars(\{red, yellow\})$ and $cars(\emptyset)$. Value of the set $COLORS$ follows from a set of first order logic predicates 6–8. The predicates allow for the presence of other colors in the traffic light so the value is uncertain. The issue is important because predicates are the only way to define constants in Event-B and there are some other cases when it is necessary. Sometimes there is only one solution that satisfies the predicates, but in the example we face an infinite set of many possible solutions.

From the point of view of the model we are also not sure that $yellow$ does not mean $green$ or red. Since it is impossible to specify absolutely all objects as constants, we need a way to specify objects in the trace that are different from existing constants.

As already mentioned, the parameter go is missing in the trace. This problem arises from the expressive ability of the Event-B language. The only way to compute some intermediate value is to add an additional parameter to the event and restrict its value with a new guard. The obvious solution is to omit computable parameters but it significantly impairs the readability of the model. The other side of the problem is that this case is syntactically indistinguishable from missing a parameter due to an error in the trace collection process.

The main advantage of the modeling approach is the ability to abstract from specific values in case we want to check only part of the properties. For example, in this model we abstracted from the presence of additional colors. During the analysis we need to ensure that at least one possible value exists and it satisfies specified properties. Like the previous case it is also hard to distinguish from the situation of imprecise specification.

IV. Related works

ProB [9] toolset includes an animator tool which can execute Event-B model events with predicates which restrict or determine values of event parameters. ProB core is written in SICStus Prolog language, so ProB toolset uses the possibilities of logical programming to find suitable values for any

identifier and avoids any predicate restrictions. In case if value is uncertain the ProB animator can give a nondeterministic choice. Traces are non-linear and can contain branches to all possible transitions to different states by given event and predicates over its parameters. Unfortunately it is problematic to use the animator for big test cases while it struggles with big sets of values, big traces and big model states. This clearly follows from the priorities of ProB developers and chosen approach: full language support is more important than time efficiency.

There are some works to combine SMT-solvers [10], [11] with Event-B models to execute them. SMT-solvers can be partially used in a deductive verification process, but they have significant constraints when it comes to execution of the whole model. Only part of SMT-solvers supports set theory and still SMT-solvers cannot support any predicates over power sets, since they go beyond first-order logic [12]. Thus, SMT-solvers can only be used on certain parts of Event-B models and, like logic programs, have problems with time efficiency.

The most time-efficient approach is code generation [13]. Event-B models are translated [14], [15], [16] to programs which can be executed over some input data. Making it possible to use collected traces as input is a quite simple task. But existing tools with a code generation approach are not considering the uncertainty problem at all and all values including constants must be given in configuration files.

So there is no effective solution which covers all needs of testing based on formal Event-B models.

V. Current results

A. User-given types

Event-B objects can have boolean type (as element of basic boolean set), integer type (as element of basic integer set), user-given type (as element of a user-defined set called the carrier set in Event-B), ordered pair type (as element of a Cartesian product) or set type (as element of a power set). The Event-B notation of all types is given in Tab. I. The example of a user-given type is $COLORS$ in Section III. Objects of a given type do not contain any value and can be related to other objects with the same type only with equality and inequality predicates such as predicate 8. These objects are uncertain by construction.

We solve the problem of uncertain value by creating a relation between an object and a pair of its carrier sets and an integer. Tab. II shows the interpretation example of a simple set of Event-B predicates. The method for selecting integer values is considered in the next subsection.

TABLE I
Event-B object types

Type	Event-B notation
boolean	$x \in BOOL$
integer	$x \in \mathbb{Z}$
user-given type	$x \in A$
ordered pair	$x \in A \times B$
set	$x \in \mathbb{P}(A)$

TABLE II
Interpretation of user-given type

Event-B	Our interpetation	Solution
$x \in A$	$GIVEN(A,i) \in A$	$i = 1$
$y \in A$	$GIVEN(A,j) \in A$	$j = 2$
$z \in A$	$GIVEN(A,k) \in A$	$k = 1$
$x \neq y$	$GIVEN(A,i) \neq GIVEN(A,j)$	
$x = z$	$GIVEN(A,i) = GIVEN(A,k)$	
	$x = GIVEN(A,i)$	
	$y = GIVEN(A,j)$	
	$z = GIVEN(A,k)$	

This approach is quite similar to that used in ProB. Sets (including carrier sets) in ProB are represented as lists without repetition and the value of a given object is a concatenation of set's name and index of element in this corresponding list. In the example from Tab. II symbolic values for x and y are $A1$ and $A2$. We also use this notation in formulas to have trace compatibility with ProB.

Proposed solution A.1. *For all objects of user-given type that differ from the existing constants, the trace uses identifiers consisting of the type name and an integer number.*

In SMT-solvers values are some generated strings. This method is more general than ours. By creating additional generation rules we can both get the same $A1$ and $A2$ strings as values and get string representation of real data to ease the testing process without additional name mapping. For example, we have a carrier set that corresponds to user abstractions, and all its elements have value of user name in the system.

An alternative approach is used in code generation, where the carrier set is not a set but a data type. In this case, the carrier sets representation must be different from the representation of usual sets and may be more convenient for large and infinite sets, but also may complicate the interpretation of predicates.

B. Computation from predicate

In Event-B we have objects designated by identifiers. In case of Event-B variables we have assignments to compute their values but in other cases we need to compute values directly from predicates. This is the only way to find values of Event-B constants and computable parameters of events.

Objects implied by identifiers can have certain values if they are described by a simple equality predicate. From predicates 12–14 we can easily extract certain values.

$$NUMBER = 20 \tag{12}$$

$$2 \mapsto TRUE = PAIR \tag{13}$$

$$SET = \{A, B\} \tag{14}$$

But in some cases predicates can lead to uncertain values. In predicate 15 $NUMBER$ is any number less than or equal to 20. In predicate 16 $PAIR$ is any ordered pair from set REL, which represents some relation. In predicates 17–18

SET contains the elements A, B and can contain every other element of the same type as A and B.

$$NUMBER \leq 20 \tag{15}$$
$$PAIR \in REL \tag{16}$$
$$A \in SET \tag{17}$$
$$B \in SET \tag{18}$$

We are referring to such predicates as the defining predicates. If a defining predicate is an equality predicate and one of its operands is an identifier and the other is a computable expression, we are referring to the expression as a defining expression. For system specifications suitable for testing we must restrict types of defining predicates to avoid the uncertainty.

Proposed solution B.1. *For all boolean, integer and ordered pair type objects whose value follows from predicates, there must be a defining predicate in the form of an equality between the identifier and the defining expression*. So for NUMBER and PAIR we allow defining predicates 12–13 and restrict defining predicates 15–16. Defining expressions can contains other identifiers as long as all identifiers have defining predicates and there are no loops in definitions:

$$NUMBER \mapsto TRUE = PAIR \tag{19}$$
$$NUMBER = 2 \tag{20}$$

Proposed solution B.2. *An equality or inequality relationship must be defined between all objects of the same user-given type*. As long as given objects do not have any value in the model, we require predicates that determine equality or inequality relation between all objects of the same carrier set like in the example from Tab. II. If a set of relations between objects is specified, we can number the objects in a way that the relation between numbers corresponds to the relation between objects. In this way, a solution is constructed, which is also reflected in Tab. II.

In addition to equality and inequality, there is a partition predicate that also allows a more compact definition of inequality relationship:

$$partition(S, S1, S2) \Leftrightarrow (S1 \cup S2 = S) \wedge (S1 \cap S2 = \emptyset) \tag{21}$$

If the carrier set S contains four constants A, B, C and D it is more comfortable to use one partition predicate than write six inequalities between A, B, C and D:

$$partition(S, \{A\}, \{B\}, \{C\}, \{D\}) \tag{22}$$

Sets can also be defined by equality predicates but there are some problems. Size of sets can be large and it doesn't make sense to define all objects as constants. As long as enumeration of big sets is either inconvenient or just impossible, an alternative way to define sets is needed.

Proposed solution B.3. *Defining predicates for sets must be equality predicates from Solution B.1 or predicates from Tab. III*. Defining predicates can be considered as a set theory problem. There is a general solution to the problem, but the value is uncertain. We can initially assume that SET is an empty set and build a minimal particular solution. We can use all other predicates over SET to ensure that the particular solution is correct. This logic can be also applied to all kinds of relations that can be considered as sets of ordered pairs.

TABLE III
DEFINING PREDICATES FOR SETS

Predicate	Solution
$x \in SET$	$\{x\} \subseteq SET \subseteq \Omega^{a}$
$s \subseteq SET$	$s \subseteq SET \subseteq \Omega$
$partition(SET, s1, s2)$	$SET = s1 \cup s2$
$partition(s1, SET, s2)$	$SET = s1 \setminus s2$

[a] Ω denotes the universe set for elements of SET.

The particular solution may be incorrect due to the predicates that are not among the defining ones. This case is illustrated by predicates 23–24. We need to complete sets S and REM with new elements for predicate 24 to be satisfied.

$$partition(S, \{A\}, \{B\}, REM) \tag{23}$$
$$REM \neq \emptyset \tag{24}$$

We also need to complete the sets that appear in the assignments, since the absence of elements leads to an incorrect state of the model. To complete a set, we need to know that it is finite and its exact cardinality.

Proposed solution B.4. *All sets that appear in assignments or require completion to satisfy the axioms must be finite and their cardinality must be specified*. The example of required definition is predicate 25.

$$finite(SET) \wedge card(SET) = 20 \tag{25}$$

In all other cases, it is possible to have a minimal solution, dynamically expanding it with new elements as they appear in the trace.

Currently we divide the cases by the finite property of sets, but we do not consider the common case where a set is finite but large and can be expanded dynamically for effective performance. One of the future goals of this research is to implement an algorithm which can identify such sets for optimization.

C. Uncertain assignments

There are three types of variable assignment actions in Event-B: simple assignment which can be also expressed in its multiple form, arbitrary choice from a set of possible values and assignment from predicate. Only simple assignments lead to certain results. Assignment actions 26–31 are complete analogues of defining predicates 12–18.

$$NUMBER := 20 \tag{26}$$
$$PAIR := 2 \mapsto TRUE \tag{27}$$
$$SET := \{A, B\} \tag{28}$$
$$NUMBER :| NUMBER' \leq 20 \tag{29}$$
$$PAIR :\in REL \tag{30}$$
$$SET :| A \in SET' \wedge B \in SET' \tag{31}$$

Arbitrary choice is certain only if the set of variants contains one and only one element. Trace analysis in case of uncertain arbitrary choice is simple but exhaustive: we must execute the rest of the trace several times, each time choosing every value possible. If there are many such actions in trace, time consumption increases exponentially. So such type of assignment action is not allowed in any form.

Proposed solution C.1. *Arbitrary choice assignment action is not allowed.*

Assignment from predicate is uncertain in most cases but the only possible way to express if-then-else construction in Event-B is a predicate. So we allow assignment from predicate in only possible form. If we have a predicate p which represents a condition and alternative values A and B which represents assignments results and are condition depended, the assignment can be written as a disjunction of conjunctions of conditional predicates and defining predicates:

$$var :| p \wedge var' = A \vee \neg p \wedge var' = B \tag{32}$$

This representation can be extended for any number of condition branches but a disjunction of all the conditional predicates must be a logical tautology. This is the completeness condition. Also all possible conjunctions between two conditional predicates must be unsatisfiable. This is the certainty condition. In the example above the completeness condition is $p \vee \neg p \Leftrightarrow \top$ and the certainty condition is $p \wedge \neg p \Leftrightarrow \bot$ and both conditions are satisfied.

If the assignment is contained in event with a single guard g, the guard can filter some condition branches. So we can also extend the assignment representation by another case where condition predicates are not covering all alternatives possible. In this case the completeness condition takes form:

$$var :| p \wedge var' = A \vee q \wedge var' = B \tag{33}$$
$$(g \Rightarrow p \vee q) \Leftrightarrow \top \tag{34}$$

Continuing these arguments, we can formulate the general case for n branches and k guards.

Proposed solution C.2. *Assignment from predicate is allowed to use only if it has form 35 and satisfies completeness condition 36 and certainty condition 37. Any other forms are prohibited.*

$$var :| \bigvee_{i=1}^{n} (p_i \wedge var' = Val_i) \tag{35}$$
$$((\bigwedge_{j=1}^{k} g_j) \Rightarrow (\bigvee_{i=1}^{n} p_i)) \Leftrightarrow \top \tag{36}$$
$$\forall i, j \cdot 0 < i < j \leq n : p_i \wedge p_j \Leftrightarrow \bot \tag{37}$$

D. Missing parameters

As already mentioned, the values of some parameters are not provided by system under test and can be computed from the corresponding guards. From the point of view of trace analysis, these parameters are simply missing. As in previous cases, we require that the values be certain. But at the same time we do not require formulas of a specific type.

Proposed solution D.1. *If the value of a computable parameter follows from the guards, it must be certain.*

Currently we solve the problem by creating a mediator – a component that translates a trace collected from a target system to trace where data is represented as model entities. This component is model-specific and is required to perform trace analysis abstracted from the system. This component can also compute and add missing parameters to the trace.

Proposed solution D.2. *The values of all computable parameters must be found before analyzing the correctness of the trace.*

Let's return to the traffic light example from Section III. The real execution trace is supplied to the input of the mediator. The mediator must substitute the identifiers red and $green$ if the corresponding traffic light colors light up. If the yellow signal lights up, the identifier $COLOR3$ is substituted according to Solution A.1. Also, if the green signal lights up, the value $TRUE$ is substituted for parameter go according guard 10, otherwise the value $FALSE$ is substituted according guard 11.

It must be taken into account that we supplement the trace with properties that are not present in the observed behavior. If implemented imprecisely, they may incorrectly influence the test verdict. This approach complicates the development of the mediator and can raise new errors in the testing process. On the other hand, it improves time effectiveness and takes into account the nature of computable parameters which helps distinguish computable parameters from parameters missing due to an error in the trace collection process.

Proposed solution D.3. *When analyzing the correctness of the trace, if a guard cannot be checked due to a missing parameter, the test fails*. The solution narrows the class of valid traces, which may be controversial for some purposes.

The automated resolution of missing parameters is a future goal of the research.

E. Optimizations

To test our approach we developed a trace analysis tool prototype. We chose Java as a programming language for the

prototype because there is a set of libraries to work with Event-B models, so we don't need to implement an Event-B environment and a parser. The implementation which uses the proposed techniques and makes direct calculations of all objects and predicates showed only a slight gain in execution time compared to ProB tool. The good sign was that the prototype with all its inefficiency still executed slightly faster. But to achieve good performance of trace analysis it was necessary to make some optimizations.

Proposed solution E.1. *Power sets, Cartesian products, sets of relations and functions must have representation other than enumerated sets*. To optimize memory, we do not store all elements in memory and interpret some formulas differently. For example, predicate 38 can be rewritten as predicate 39. Rewriting and interpretation of formulas is done automatically.

$$SUBSET \in \mathbb{P}(SET) \tag{38}$$

$$SUBSET \subseteq SET \tag{39}$$

Proposed solution E.2. *When processing quantifiers, it is necessary to automatically analyze the formula to limit the iteration over objects*. Time measurements showed that processing quantified predicates is the most time consuming task. Analyzing a formula to reduce the range of possible values greatly speeds up the execution of formulas.

Proposed solution E.3. *Checking invariants should be optional*. Model invariants are a large set of predicates that need to be checked after the execution of every event from the trace. These predicates often contain quantifiers so even with optimizations they are still time-consuming.

If we consider trace analysis in the context where the model is already verified, the fact that a state of the system satisfies the invariants follows from the fact that parameters of the event satisfies the guards. So there is no need to check invariants at all. There is no such option when using the ProB animator.

If model is not verified, the possible optimization is to check only invariants that contain variables changed by action of the last event.

VI. Future work

At the moment, all model transformations are carried out automatically at the level of individual formulas. No techniques have been proposed to transform the entire model manually for a more efficient testing process. It is necessary to conduct a more extensive analysis of existing practices for constructing formal models. At the moment, the main reference point was the access and information flow control model of OS Linux [5], [17].

The restrictions proposed in subsection V-B are strong but reasonable for models which are used for testing. In case it is not possible to comply with the restrictions, usage of SMT-solvers on the specific part of the model may help to select constant values as long as resolution of constants is performed only once and does not affect the efficiency of following trace analysis. It is necessary to conduct experiments and evaluate the effectiveness of this approach.

The current version of the tool does not provide deferred computation for objects other than sets that can be expanded dynamically. In the example from Section III, it is possible to determine the uncertain initial state of the model from the first events. This approach can be used to support some level of abstraction in the model.

The desired result of the research is to bring test execution times closer to those using code generation. Testing the trace analysis tool prototype on a Linux OS security system model and 39000 test traces gives us acceleration from more than 10 hours analysis using ProB down to 8 minutes analysis using the prototype. Manually converting the model into Python code allows us to run the same set of tests in 1 minute.

VII. Conclusion

This paper is devoted to the research of problems that arise during testing of formal systems such as security systems of general purpose operating systems such as Linux. It is necessary to build a formal model of security policies to define the requirements and be able to prove their consistency and completeness. The verified security policy model is used in a testing of system implementation where we perform a model analysis to ensure that observable system behavior satisfies the model. But the analysis requires large resources and in some cases faces insurmountable difficulties.

The research revealed the particular problems connected with value uncertainty. We identified the subset of Event-B language to avoid unsolvable uncertainty and proposed the approach to resolve certain values and effectively perform trace analysis. These results can be applied to other formal model languages which use first order logic.

We implemented the proposed methods in the trace analysis tool prototype. The prototype was tested on a large set of test traces with a security system model and showed a significant gain in analysis time compared to the ProB animator tool. This analysis can be performed with a single entity of a model state space which shows support of long traces.

References

[1] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to runtime verification," in *Lectures on Runtime Verification*, ser. Lecture Notes in Computer Science, E. Bartocci and Y. Falcone, Eds. Springer, 2018, vol. 10457, pp. 1–33.

[2] G. Reger and K. Havelund, "What is a trace? A runtime verification perspective," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Springer, 2016, vol. 9953.

[3] *Information protection. Formal access control model. Part 1. General principles*, State Std. GOST R 59 453.1, 2021, (in Russian).

[4] *Information protection. Formal access control model. Part 2. Recommendations on verification of formal access control model*, State Std. GOST R 59 453.2, 2021, (in Russian).

[5] P. Devyanin, D. Efremov, V. Kulyamin, A. Petrenko, A. Khoroshilov, and I. Shchepetkov, *Modeling and verification of access control security policies in operating systems*. Moscow, Russia: Hotline-Telecom, 2019, (in Russian).

[6] D. Efremov, V. Kopach, E. Kornykhin, V. Kulyamin, A. Petrenko, A. Khoroshilov, and I. Shchepetkov, "Runtime verification of operating systems based on abstract models," in *Proceedings of the Institute for System Programming of the RAS*. Proceedings of ISP RAS, 2021, vol. 33, no. 6, pp. 15–26, (in Russian).

[7] P. Devyanin and M. Leonova, “The techniques of formalization of OS Astra Linux Special Edition access control model using Event-B formal method for verification using Rodin and ProB,” in *Prikladnaya Diskretnaya Matematica*, 2021, no. 52, pp. 25–40, (in Russian).
[8] J.-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, and L. Voisin, “Rodin: An open toolset for modelling and reasoning in Event-B,” in *The International Journal on Software Tools for Technology Transfer*. Springer, 2010, vol. 12, pp. 447–466.
[9] M. Leuschel and M. Butler, “ProB: an automated analysis toolset for the B method,” in *The International Journal on Software Tools for Technology Transfer*. Springer, 2008, vol. 10, pp. 185–203.
[10] D. Déharbe, “Integration of SMT-solvers in B and Event-B development environments,” *Science of Computer Programming*, vol. 78, pp. 310–326, 03 2013.
[11] J. Schmidt and M. Leuschel, “SMT solving for the validation of B and Event-B models,” in *The International Journal on Software Tools for Technology Transfer*. Springer, 2022, vol. 24, pp. 1043–1077.
[12] E. Brauer, “Second-order logic and the power set,” *Journal of Philosophical Logic*, vol. 47, pp. 123–142, 2018.
[13] A. Fürst, T. Hoang, D. Basin, K. Desai, N. Sato, and K. Miyazaki, “Code generation for Event-B,” presented at the Integrated Formal Methods 2014, Bertinoro, Italy, 09 2014.
[14] S. Wright, “Automatic generation of C from Event-B,” presented at the Workshop on Integration of Model-based Formal Methods and Tools, Bangkok, Thailand, 02 2009.
[15] F. Yang, J.-P. Jacquot, and J. Souquières, “JeB: Safe simulation of Event-B models,” presented at the The 20th Asia-Pacific Software Engineering Conference, Bangkok, Thailand, 12 2013.
[16] N. Cataño and V. Rivera, “EventB2Java: A code generator for Event-B,” in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, O. T. S. Rayadurgam, Ed. Springer, 2016, vol. 9690.
[17] P. Devyanin, *Security models of computer systems. Control for access and information flows*. Moscow, Russia: Hotline-Telecom, 2013, (in Russian).