Analysis of the Verification Approaches for the Cyber-Physical Systems

Sergiy Korotunov $^{1[0000-0001-5184-4455]}$, Galyna Tabunshchyk $^{2[000-0003-1429-5180]}$, Karsten Henke $^{3[0000-0001-5424-9053]}$, Dieter Wuttke $^{4[0000-0001-8030-647X]}$

^{12}Zaporizhzhia National Technical University, Zhukovsky str., 64, Zaporizhzhia, 69063, Ukraine

skorotunov@yahoo.com, galina.tabunshchik@gmail.com

^{34}Ilmenau University of Technology, Max-Planck-Ring str., 14, Ilmenau, 98693, Germany

karsten.henke@tu-ilmenau.de, dieter.wuttke@tu-ilmenau.de

Abstract. In the paper the possibility of utilization of the Kripke structures for applying linear-time temporal logic in problems of verification of reactive systems is considered. The definition and main characteristics of the cyber-physical systems, finite state machines, Kripke structures and temporal logics are considered by the authors. Example of modeling on the base of GOLDi is provided.

Keywords. Verification, parallel program, reactive system, cyber-physical systems, Kripke structure, finite-state machine, temporal logic.

1 Introduction

With the high complexity of modern computer systems (millions of lines of code and billions of transistors), it is impossible to completely avoid errors. It is not only related to the application software developed in a short time in a competitive market environment, but also to critical components, such as hardware (I/O devices, microprocessors) and software (compilers, operating systems). It is obvious that the systems, in the design and implementation of which mistakes are made, can in one or another situation behave in unpredictable ways and lead to serious consequences. The literature describes many cases of this kind. As vivid examples following can be represented.

In 1996, the launch vehicle Ariane 5, developed by the European Space Agency, exploded at the 39th second after launch. As it turned out later, the onboard system partially used the software of the previous version of the rocket, the Ariane 4, in particular, the inertial navigation system control module [1]. When converting from a 64-bit floating-point number representing the inclination of the rocket to a 16-bit integer, an overflow occurred that was not handled by the module. When developing the module, it was assumed that overflow was impossible due to the physical limitations of the rocket, but new engines which were used in the Ariane 5 surpassed those limitations.

Another notable example is the case of the medical device Therac-25 [2], which was launched in 1982 by the Canadian firm Atomic Energy of Canada Limited. The device was intended for radiation therapy – irradiation of a cancer. In some situations, the irradiation intensity increased by 2 orders or more due to a number of mistakes made in the design and implementation of the built-in control system. At least two patients died, and several people were disabled during the operation of the device (the period from June 1985 to January 1987).

It is important to understand that errors in computer systems are not exceptional. According to statistics, the average number of errors per thousand lines of un-debugged code ranges in 10-50[5] [3]. Moreover, there is a tendency to the degradation of design quality (apparently, this is a consequence of the increasing complexity of systems and optimization of the costs of their creation) [4]. At the same time, our life is increasingly dependent on computers. Therefore, ensuring the correctness and reliability of computer systems (in other words, verification) is becoming increasingly important.

2 Program verification

2.1 Verification approaches

Verification is the process of checking of the compliance of a program (its model) with the requirements placed on it. If the program meets the requirements, it is called correct. Otherwise, the program is called incorrect, and the fact of non-compliance of the program with the requirements is an error. Thus, verification is the analysis of the program for the presence or absence of errors in it. An indefinite verdict is also possible when errors are not found, but their absence has not been proven.

It is necessary to take into account many nuances. First, the requirements, as a rule, are formulated informally, in natural language, so it is not always possible to unambiguously determine whether the program corresponds to them or not. Secondly, proving the absence of errors in the program is extremely difficult from a mathematical point of view (because of the fact that this task is not fully amenable to automation).

Verification methods can be divided into three main groups [5]:

- formal methods, which utilize mathematically rigorous analysis of the program model and requirements model;
- testing methods that verify the actual behavior of the program on a certain set of scenarios;
- expertise performed by people based on their knowledge and experience directly on the design results (for example, code inspection).

Each of the specified groups of methods has its advantages and disadvantages, each has its own area of applicability. Full verification of complex systems of responsible purpose is impossible without the joint use of different approaches. This work focuses on mostly formal methods of verification programs.

2.2 Formal verification

Formal verification is based on mathematical (logical) modeling programs and requirements for them. The idea is exactly the same as when using models in other areas of knowledge:

- a model is created – an idealized description of the object or phenomenon under study;
- the model is investigated using mathematical methods;
- research results are transferred to a real object or phenomenon.

Of course, the applicability of this approach is determined by the models used – it is necessary to clearly understand the conditions for their adequacy.

The general process of formal verification [6] is shown in Fig. 1:

- a formal program model is created;
- a formal requirements model is created;
- the compliance of the program model with the requirements model is formally verified;
- based on the results of the test, it is concluded that the real program does not meet the real requirements (in other words, that there are no or no errors in the program).

For the presentation of program models and requirements models, the languages of the formal specification of programs (modeling languages) and the languages of the formal specification of requirements are used respectively.

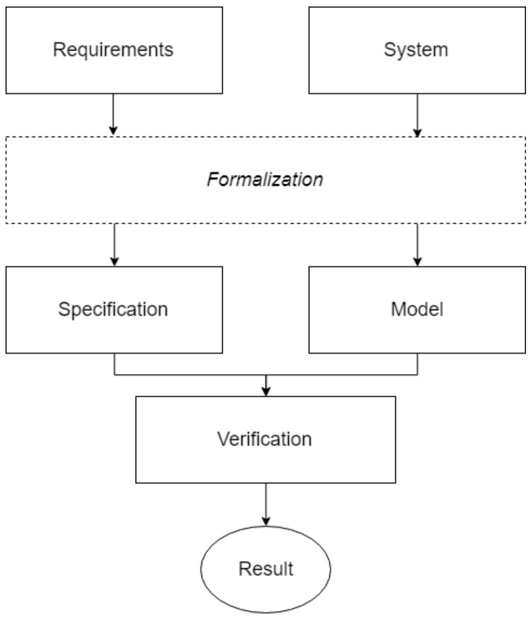

Fig. 1. Formal verification process

Formal verification methods differ in the types of program models (state machines, Petri nets, labeled transition systems), types of requirements models (software contracts, production rules, temporal statements), correspondence relations (equivalence, simulation, route inclusion) and compliance verification techniques (state space research, symbolic analysis, deductive inference). In this paper we briefly consider one of the most popular formal verification methods – model checking.

2.3 Model checking

In model checking, requirements for programs are represented by logical formulas of a certain type, and programs themselves are structures interpreting formulas of this type; the correspondence relation is the truth of a formula on a structure (a program satisfies the requirements if and only if the corresponding structure is a model of the corresponding formula).

Often, temporal logics are used to represent the requirements – logic that allows one to define the relationship of events in time, for example, Linear-time Temporal Logic or Computational Tree Logic. Accordingly, Kripke structures and related formalisms (marked transition systems) are used to represent programs.

Kripke structures are used to model reactive systems – systems operating in an “infinite loop” and interacting with their environment. As an example of such systems cyber-physical system can be seen. The behavior of the system is modeled by calculation in the Kripke structure [7].

Propositional temporal logic can be used to describe requirements, in particular LTL. The main method for establishing the correctness or incorrectness of a model is to search in the state space (bypassing the state graph).

3 Finite state machines

A finite state machine (FSM) is a computation model that can be implemented with hardware or software and can be used to simulate sequential logic (for example computer program) [8]. Finite state machine generates regular languages. Finite state machines can be used to model problems in many fields including mathematics, artificial intelligence, games, and linguistics.

3.1 Deterministic and non-deterministic finite state machines

There are two types of finite state machines: deterministic finite state machines (deterministic finite automaton) and non-deterministic finite state machines (nondeterministic finite automata).

A deterministic finite automaton (DFA) is described by a five-element tuple:

$$\langle Q, \Sigma, \delta, q_0, F \rangle . \tag{1}$$

Where Q is a finite set of states, Σ is a finite, nonempty input alphabet, δ is a series of transition functions, q_0 is the starting state, F is the set of accepting states.

There must be exactly one transition function for every input symbol in Σ from each state. DFAs can be represented by following diagrams:

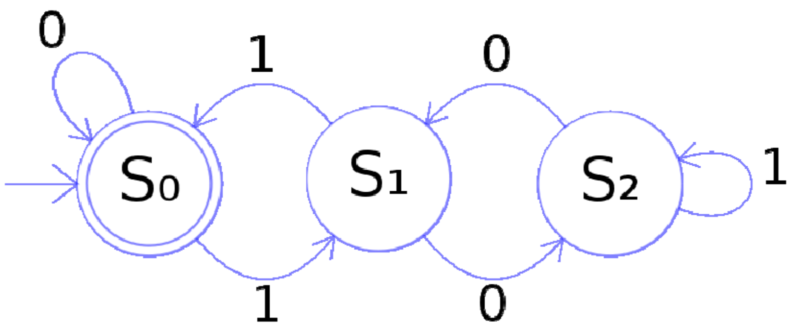

Fig. 2. Deterministic finite automaton diagram

Similar to a DFA, a nondeterministic finite automaton (NDFA) is described by an above five-element tuple. Unlike DFAs, NDFAs are not required to have transition functions for every symbol in Σ , and there can be multiple transition functions in the same state for the same symbol. Additionally, NDFAs can use null transitions, which are indicated by ε . Null transitions allow the machine to step from one state to another without having to read a symbol. NDFAs can be represented by diagrams of this form:

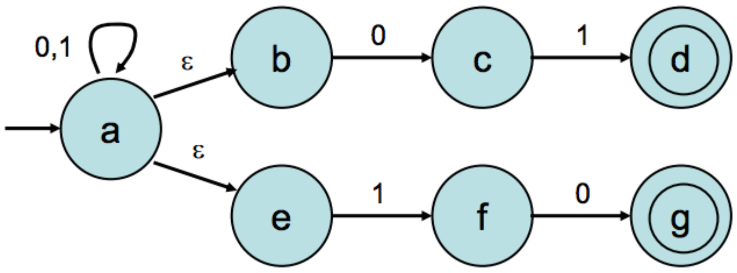

Fig. 3. Nondeterministic finite automata diagram

One might assume that NDFAs can solve problems that DFAs cannot, but NDFAs are just as powerful as DFAs. However, a DFA will require many more states and transitions than an NDFA would take to solve the same problem. Converting from DFA to NDFA and vice versa is possible which makes them equivalent.

3.2 Finite state machine example from the real world

A system where particular inputs cause particular changes in state can be represented using finite state machines. This example describes the various states of a turnstile. Inserting a coin into a turnstile will unlock it, and after the turnstile has been pushed, it locks again. Inserting a coin into an unlocked turnstile, or pushing against a locked turnstile will not change its state. Diagram of the turnstile FSM is following:

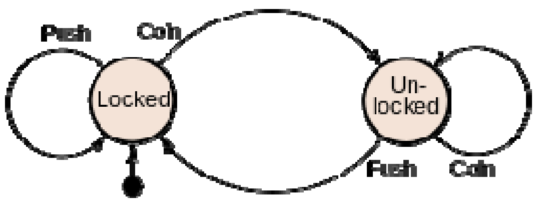

Fig. 4. Turnstile FSM diagram

4 Kripke structure

4.1 Definition of the Kripke structure

Saul Aaron Kripke introduced structures named after him to the logical and philosophical use in the late 1950s [9]. In general terms, the Kripke structure is a system of possible worlds and transitions between them: each world is static and interpreted in some traditional way. Its diagram can be seen at Fig. 5. The Kripke structure is the quadruple $\langle S, S_0, R, L\rangle$, where S is the set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation, $L : S \rightarrow 2^{AP}$ is a labeling function which marks each state of the structure with a set of atomic propositions.

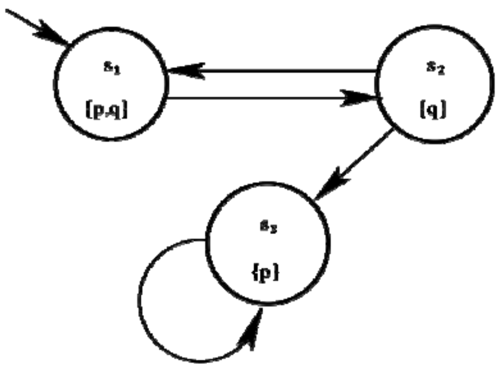

Fig. 5. Kripke structure diagram

A Kripke structure is an annotated finite-state transition graph.

4.2 Kripke structure example from the real world

Let a set of atomic propositions have the form $\{locked, lock, unlock\}$. Consider the Kripke structure $M = \langle \{S_0, S_1, S_2, S_3\}, \{S_0\}, R, L \rangle$, which simulates a door with a lock which can be seen at Fig. 6. The initial state is marked by an incoming arrow; next to each state there are statements which are true in it. The states of the structure are labeled as follows:

- $L(S_0) = \varnothing$ – the door is open; no action is taken on the lock;
- $L(S_1) = \{lock\}$ – the door is open; it is being closed;
- $L(S_2) = \{locked\}$ – the door is closed, no action is taken with the lock;
- $L(S_3) = \{locked, unlock\}$ – the door is closed; it is being opened.

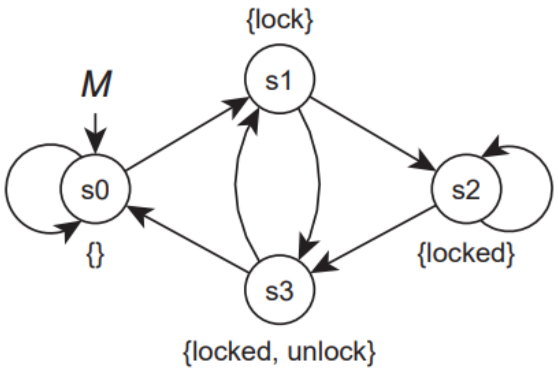

Fig. 6. Kripke structure that simulates a door lock

5 Temporal logic and its scope

5.1 Cyber-physical systems and their properties

A parallel program is a finite set of sequential programs over a common set of variables [10]. A separate program of this set is called a process. Programs running in the "infinite loop" are considered in this paper. These are the so-called reactive systems.

Such systems respond to environmental events by performing certain actions in response. This is an extensive class of programs, including cyber-physical systems, operating systems, device drivers, telecommunication environments, control systems, etc. [11].

Cyber-physical systems are the systems that provide the integration of computing, physical processes and networks, or as systems where software and physical subsystems are closely bounded, each of which works in a variety of temporal and spatial dimensions, demonstrating clear and multiple behavioral patterns, and interacts in a variety of ways [12]. Modern trends in productivity and complexity of requirements for systems use require fundamentally new design approaches in which cybernetic and physical components are integrated at different stages.

In general, the qualitative properties of cyber-physical systems can be classified into the following two broad categories:

- reachability or guarantee properties that raise the question of whether a system can achieve a configuration that satisfies a particular property;
- security properties that raise the question of whether the system can remain forever in configurations that satisfy a particular property.

The main properties of cyber-physical systems include following [13]:

- high degree of automation,
- reorganization / reconfiguration of the dynamics,
- cybernetic capabilities in each physical component,
- the ability of networks to work on multiple scales,
- integration on different time and spatial scales.

The behavior of cyber-physical systems is described in terms of sequences of events distributed in time. So-called temporal logics are often used for the specification of requirements for cyber-physical systems. Temporal logics are formal languages that allow to define the interrelationships of events in time: causal relationships, restrictions on the relative order, the magnitude of delays between events, etc. The following examples can be cited as temporal properties: the system always works without freezing; two users cannot simultaneously access shared data; a request with a higher weight will be processed before constipation with a lower weight.

5.2 Linear-time Temporal Logic

Among number of temporal logics, two have become particularly popular in computer science: Linear-time Temporal Logic (LTL) [14] and Computation Tree Logic (CTL) [15]. In this paper LTL is considered.

In LTL, two temporal operators are added to the syntax of classical propositional logic: the unary operator X (next time) and the binary operator U (until). These two operators form the LTL temporal basis.

Let a set of elementary statements AP (Atomic Propositions) be given. The LTL formula is given by the following grammar [16]:

$$\varphi := p \,|\, \varphi \vee \varphi \,|\, \neg\varphi \,|\, X\varphi \,|\, \varphi U \varphi \cdot \tag{2}$$

Here, p is an arbitrary elementary statement from the AP set. For convenience, in LTL formulas it is possible to use:

- derived logical connectives, for example, $\wedge$ and $\rightarrow$;
- logical constants, *true* and *false*;
- derived temporal operators, for example F (in the Future), G (Globally), W (Weak until) and R (Release).

Temporal operators can be interpreted as following.

- The formula φ is true at the next point in time – $X\varphi$.
- The formula ψ is true now or will surely become true in the future, but up to this point (not inclusive) the formula φ should be true – $\varphi U \psi$.
- The formula φ is true now or will become true sometime in the future – $true U \varphi$ which is also can be displayed as $F\varphi$.
- The formula $\neg\varphi$ is false now and will never become true in the future (always, from now on, the formula φ is true) – $\neg F \neg \varphi$ which is also represented as $G\varphi$.
- The formula φ is true until the formula ψ is not become true, without requiring the formula ψ to ever become true in the future – $(\varphi U \psi) \vee G\varphi$ or $\varphi W \psi$.
- The formula ψ is true until the moment (inclusive) when φ becomes true the first time; if such a moment never comes, the formula ψ is always true (it possible to say that φ frees ψ) – $\neg(\varphi U \neg\psi)$ or $\varphi R \psi$.

Thus, the requirements for the system, which simulates a door lock and was described above, can be expressed in LTL logic:
- $G\{\neg(lock \wedge unlock)\}$ – for system it is impossible to open and close door at the same time;
- $G\{\neg(locked \wedge lock)\}$ – system should never close the closed door;
- $G\{lock \rightarrow X(locked)\}$ – if the door is closed, at the next moment in time it will become closed.

6 Experimental modeling on the base of GOLDi

Integrated Communication Systems Group at the Ilmenau University of Technology has many years of experience in integrated hard- and software systems and over 10 years of experience in dealing with Internet-supported teaching in the field of digital system design [17]. Grid of Online Lab Devices Ilmenau (GOLDi) gives the students the possibility to work on real physical systems without the need to stand in line at a lab or the need to take care of opening hours and offers the students a working environment that is as close as possible to a real world laboratory. Under real laboratory conditions disturbances can appear and lead to failures of the control algorithm that cannot be detected under virtual lab conditions.

Online laboratories offer various features like visualization and animation, which allows to observe and to test all the properties of the design. In connection with formal design techniques, simulation and prototyping are used to establish a foundation for the development of a reliable system design. To check the functionality of the whole design, some special simulation and validation features are included as integral part of the GOLDI system. This offers various possibilities for the execution of simulations [18], such as:

- usage of simulation models of the physical system for visual prototyping,
- step by step and parallel execution of these prototypes,
- visualization of the simulation process with the tools also used for specification,
- features for test pattern generation and
- code generation for hardware and software synthesis.

GOLDI offers a Web-based environment supporting the above mentioned features to generate and execute a design by using simulation models.

As an example of modeling it was decided to create Kripke structure of the elevator which is located in the GOLDi. This elevator has ability to move upwards and downwards from floor to floor and open or close its door.

The atomic propositions for the Kripke structure representing the elevator are as follows:

- 1st – elevator is located at the 1st floor;
- 2nd – elevator is located at the 2nd floor;
- DO – door is open;
- MU – elevator is moving in the upward direction;
- MD – elevator is moving in the downward direction.

For clarity, each state is labeled with both the atomic propositions that are true in the state and the negations of the propositions that are false in the state. The labels on the arcs indicate the actions that cause transitions and are not part of the Kripke structure. Kripke structure of the elevator can be seen at Fig.7.

This model can be used for further formal verification. For example one might want to determine that "door of the elevator is closed and it is moving upwards". S_0, $p = \neg 1st \wedge \neg 2nd \wedge DO \wedge MU \wedge \neg MD$. Using Kripke structure this can be determined.

7 Conclusions

In the paper the authors considered the possibility of utilization of the Kripke structures for applying linear-time temporal logic in problems of verification of reactive systems. Review of the main characteristics of the cyber-physical systems, finite state machines, Kripke structures and temporal logics was carried out by the authors. Example of modeling on the base of GOLDi was provided in the paper.

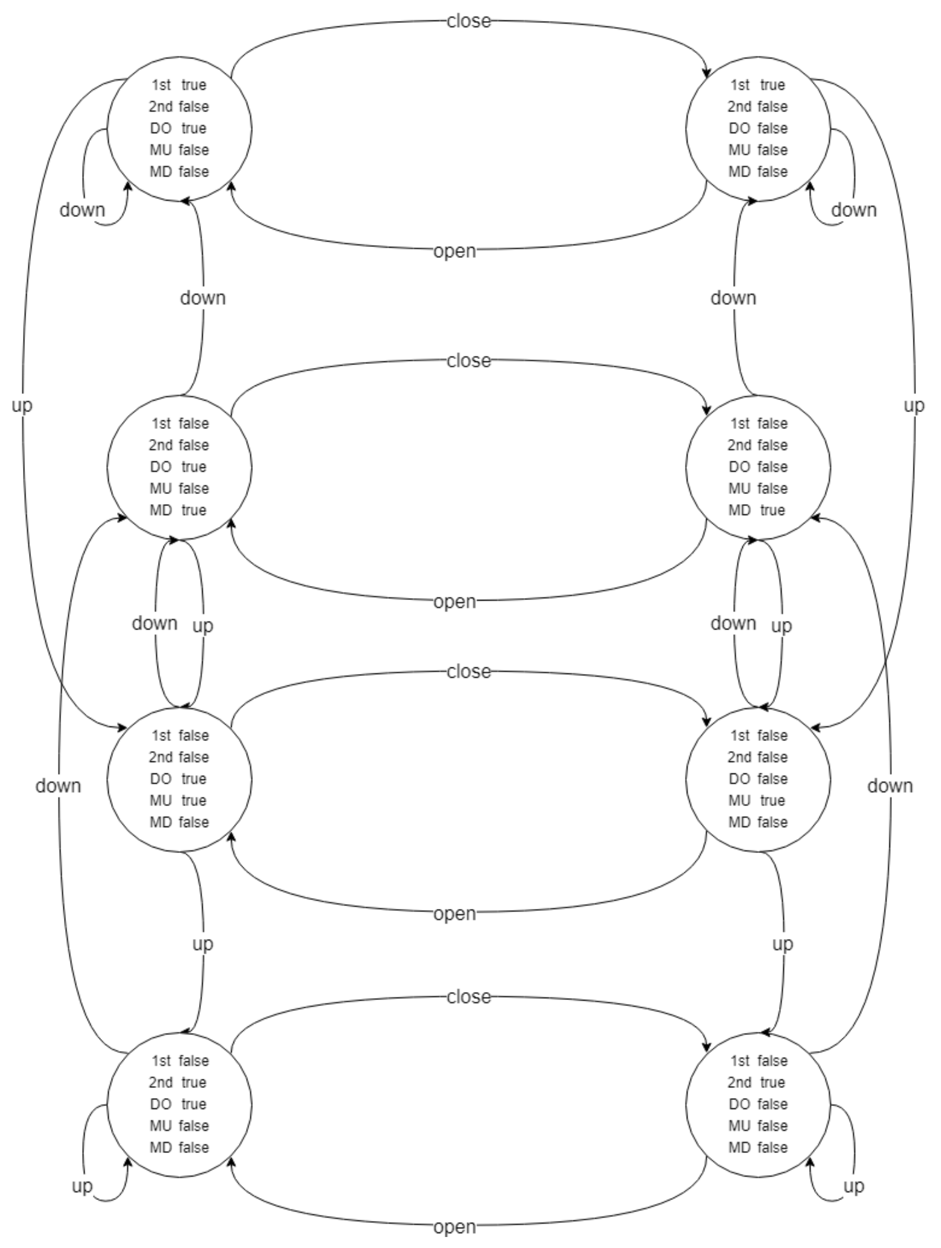

Fig. 7. Kripke structure of the elevator in GOLDi

8 References

1. Garfinkel, S., Goode, L., Barrett, B., Pardes, A.: History's Worst Software Bugs, https://www.wired.com/2005/11/historys-worst-software-bugs/?currentPage=2.

2. Leveson, N., Turner, C.: An investigation of the Therac-25 accidents. Computer. 26, 18-41 (1993).
3. McConnell, S.: Code complete, second edition. Microsoft Press, Redmond (Washington) (2004).
4. Lloyd, S.: Programming the universe. Knopf, New York (2006).
5. Habra, N., Abran, A., Lopez, M., Sellami, A.: A framework for the design and verification of software measurement methods. Journal of Systems and Software. 81, 633-648 (2008).
6. Tuch, H., Klein, G., Heiser, G.: OS Verification -- Now!, https://www.usenix.org/legacy/event/hotos05/final_papers_backup/tuch/tuch_html/index.html.
7. Subbotin, S.: Methods of data sample metrics evaluation based on fractal dimension for computational intelligence model buiding. 2017 4th International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T). (2017).
8. Grant, P.: Elementary Computability, Formal Languages and Automata. Software & Microsystems. 1, 171 (1982).
9. Gabbay, D.: Kripke Saul A.. Semantical considerations for modal logics. Proceedings of a Colloquium on Modal and Many-valued Logics, Helsinki, 23-26 August, 1962, Acta Philosophica Fennica 1963, pp. 83–94. The Journal of Symbolic Logic. 34, 501 (1969).
10. Oliinyk, A., Skrupsky, S., Subbotin, S.: Parallel Computer System Resource Planning for Synthesis of Neuro-Fuzzy Networks. Recent Advances in Systems, Control and Information Technology. 88-96 (2016).
11. Müller-Olm, M., Schmidt, D., Steffen, B.: Model-Checking. Static Analysis. 330-354 (1999).
12. Korotunov, S., Tabunshchyk, G., Wolff, C.: Cyber-Physical Systems Architectures and Modeling Methods Analysis for Smart Grids. 2018 IEEE 13th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT). (2018).
13. Miclea, L., Sanislav, T.: About dependability in cyber-physical systems. 2011 9th East-West Design & Test Symposium (EWDTS). (2011).
14. Pnueli, A.: The temporal logic of programs. 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). (1977).
15. Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems. 8, 244-263 (1986).
16. Tonetta, S.: Linear-time Temporal Logic with Event Freezing Functions. Electronic Proceedings in Theoretical Computer Science. 256, 195-209 (2017).
17. Henke, K., Fäth, T., Hutschenreuter, R., Wuttke, H.: Gift – An Integrated Development and Training System for Finite State Machine Based Approaches. International Journal of Online Engineering (iJOE). 13, 147 (2017).
18. Henke, K., Vietzke, T., Wuttke, H., Ostendorff, S.: Safety in Interactive Hybrid Online Labs. International Journal of Online Engineering (iJOE). 11, 56 (2015).